# **PRT582**

# **SOFTWARE ENGINEERING: PROCESS AND TOOLS**

**Software Unit Testing Report**

**Name: Simin Li**

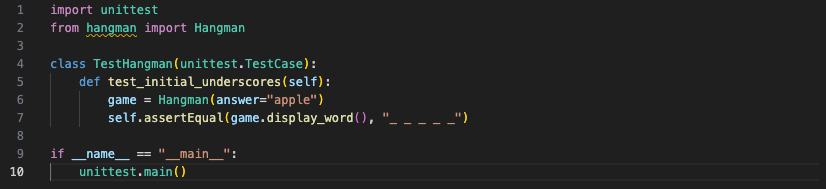
**ID: S390115**

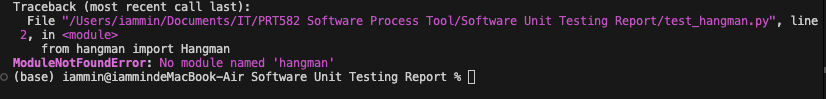
**Darwin Danala Campus**

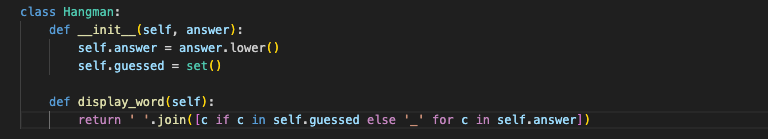
**Introduction**  
This report focuses on the development of a Hangman game using Python. The project adopts a Test Driven Development (TDD) approach, where tests are written before implementing the actual features. This method ensures that each function is developed according to its expected behavior, reducing errors and improving code reliability. To support this approach, unittest, Python's built-in unit testing framework, will be used for automated testing. By combining TDD with unittest, the development will proceed step by step, gradually fulfilling all the functional requirements specified in this assessment.

**Process**  
Step 1: Create test file and minimal Hangman Class

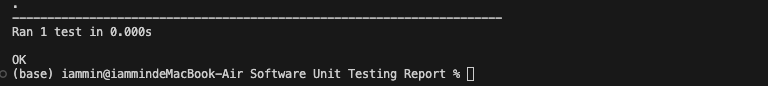
Create a file named test\_hangman.py (hereafter referred to as the test file), import the unittest module, and write a minimal Hangman class interface along with a failing test.



Run this code, and the output is as follows:  


The error occurs because hangman.py has not been created yet and the Hangman class is undefined. Next, create hangman.py and write the minimal code that passes the test.

Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets this requirement.

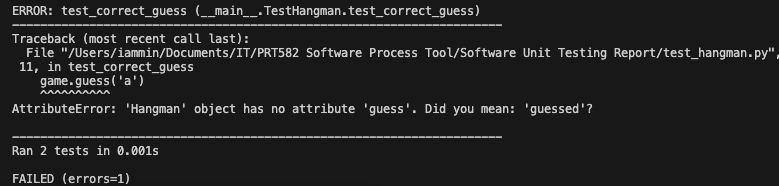


Step 2: Implement guess functionality and display correct letters

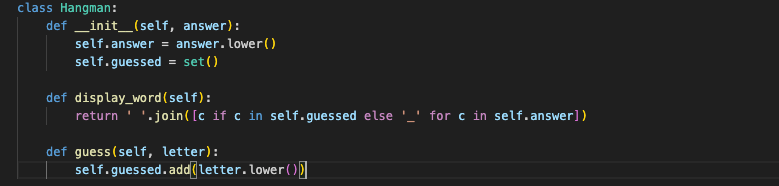
Add code in test\_hangman.py to check whether correctly guessed letters are displayed.



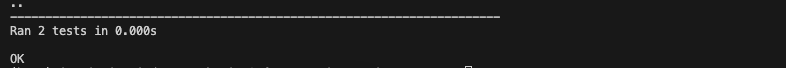
Run the test file, and the output is as follows:



Update hangman.py to implement the guess( ) method.



Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets this requirement.

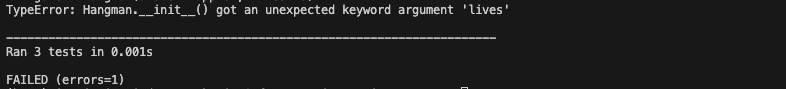


Step 3: Deduct life on wrong guess

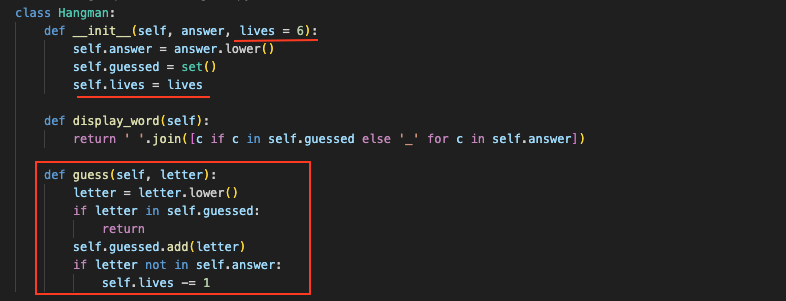
Add a test in the test file for “Every time the player guesses a letter wrong, the player’s life will be deducted.”



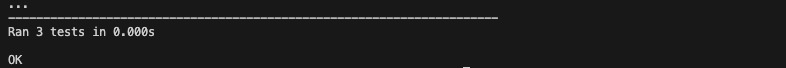
Run the test file, and the output is as follows:



Update the \_\_init\_\_( ) and guess( ) methods in Hangman.py to define the player's lives and implement life deduction for each wrong guess.

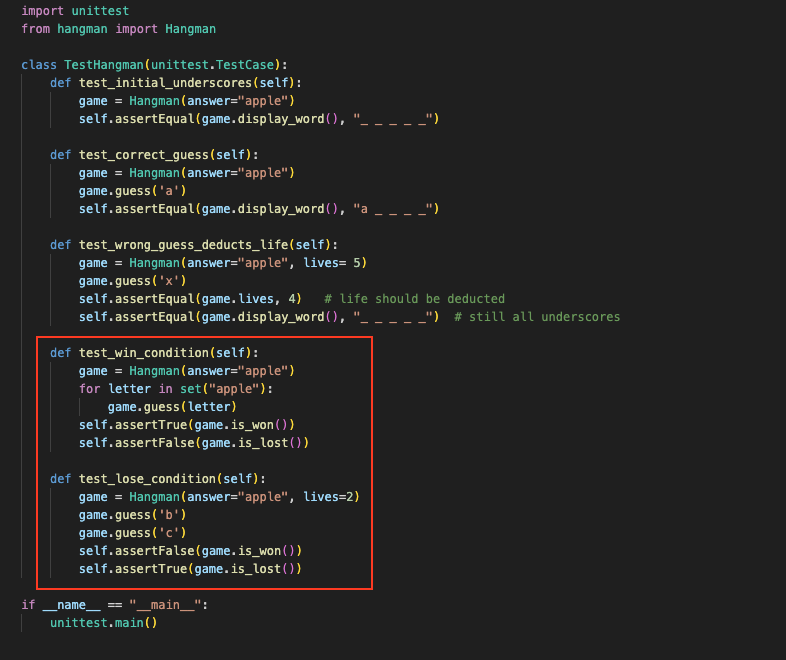


Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets this requirement.

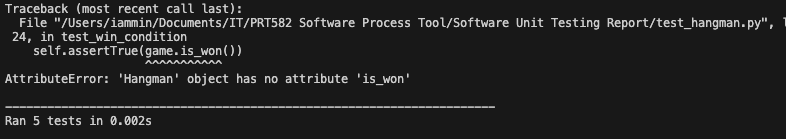


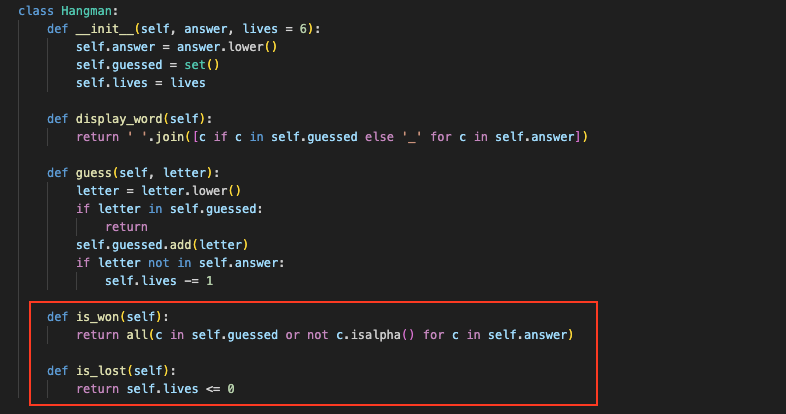
Step 4: Check win and lose Conditions

Add tests code in the test file to detect win (all letters guessed) and lose (lives reach 0) conditions.



Run the test file, and the output is as follows:

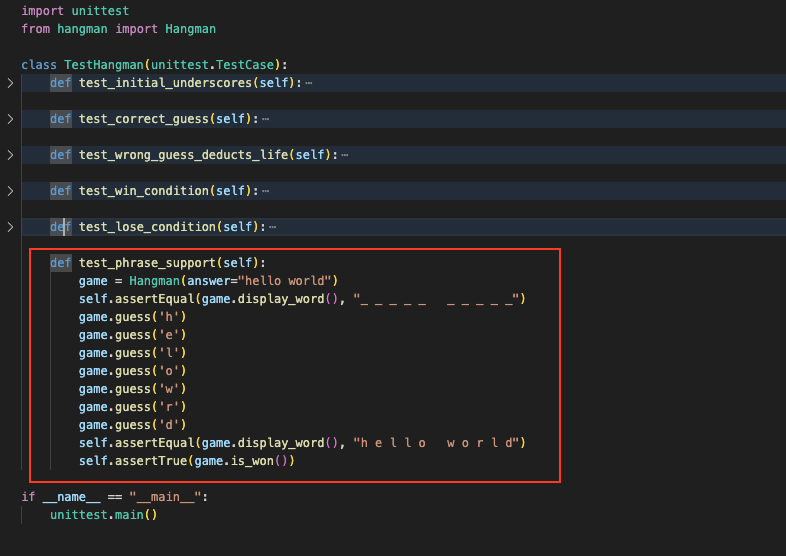


Update hangman.py, add is\_won( ) method to verify winning conditions and is\_lost( ) method to verify losing conditions.  


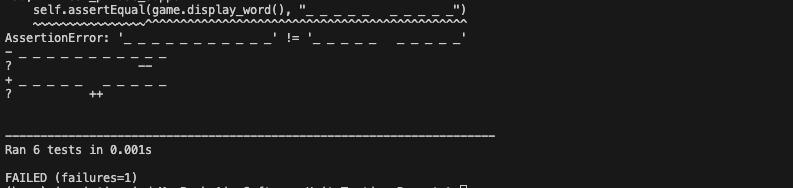
Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets the requirements.



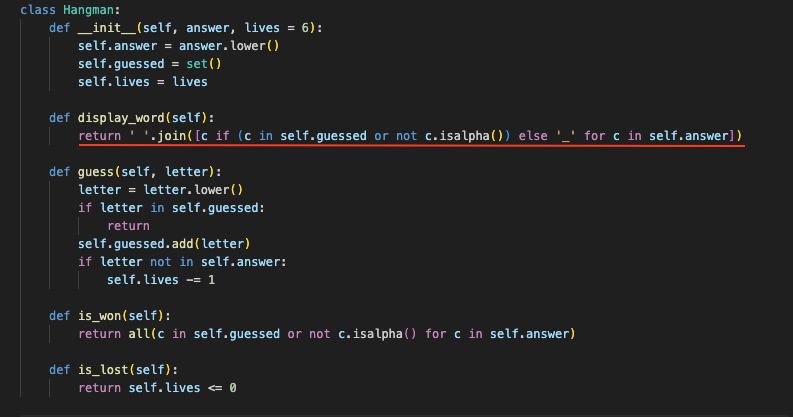
Step 5: Support the input of phrases with spaces (e.g.hello world)  
Add code to test phrases like "hello world" in test\_hangman.py.



Run the test file, and the output is as follows:



Update display\_word() in hangman.py to handle spaces.

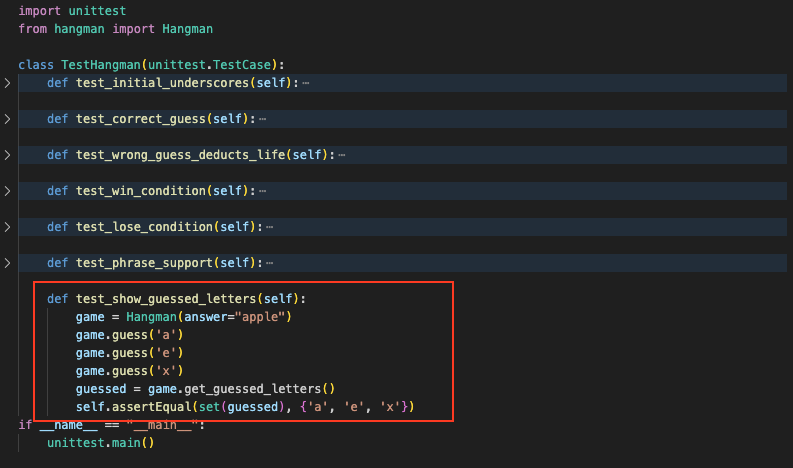


Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets the requirements.

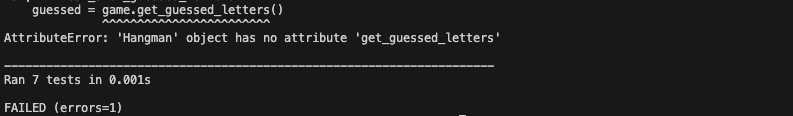


Step 6: Store guessed letters in a list

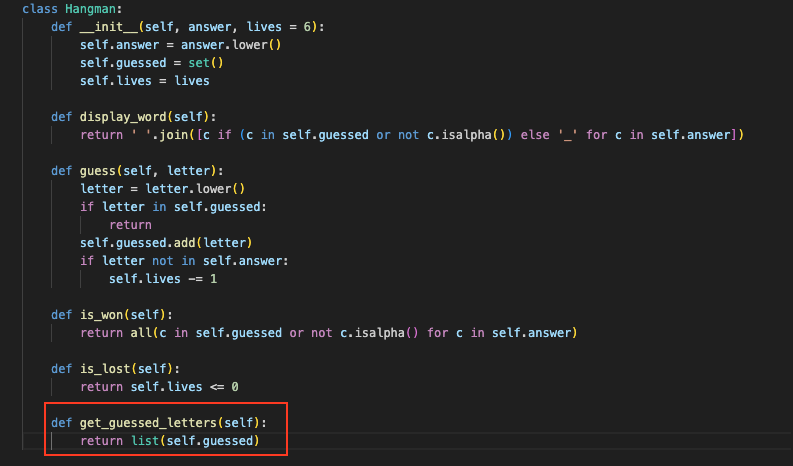
Add code in the test file, to test whether guessed letters can be stored in a list, which provides game information and helps players track their guesses.

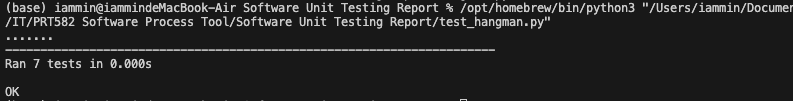


Run the test file, and the output is as follows:



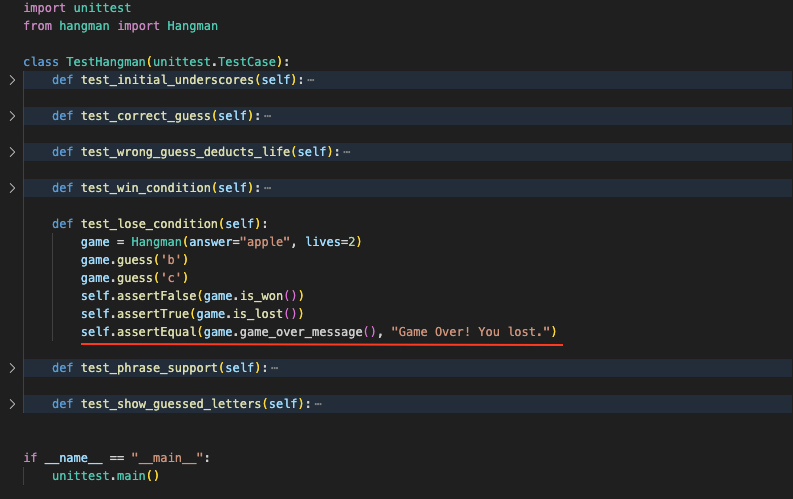
Add get\_guessed\_letters( ) method in the Hangman class in hangman.py. It aims to store the guessed letter in a list.



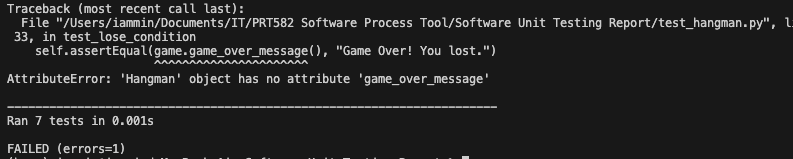
Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets the requirements.  


Step 7: Display “Game Over” message

Update the test\_lose\_condition( ) in the test file for testing “Game Over” messages.



Run the test file, and it shows that game\_over\_message( ) is missing.



Add the game\_over\_message() method in hangman.py.



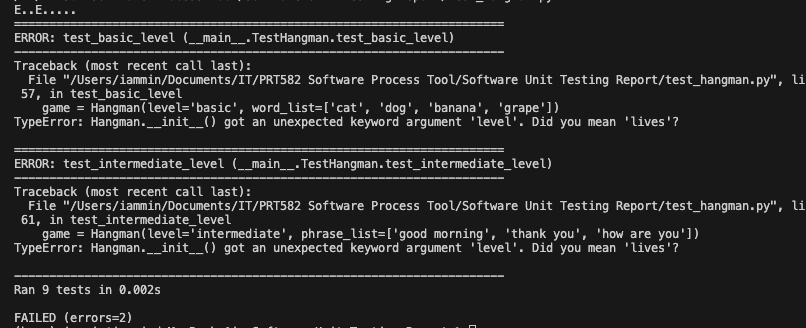
Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets the requirements.



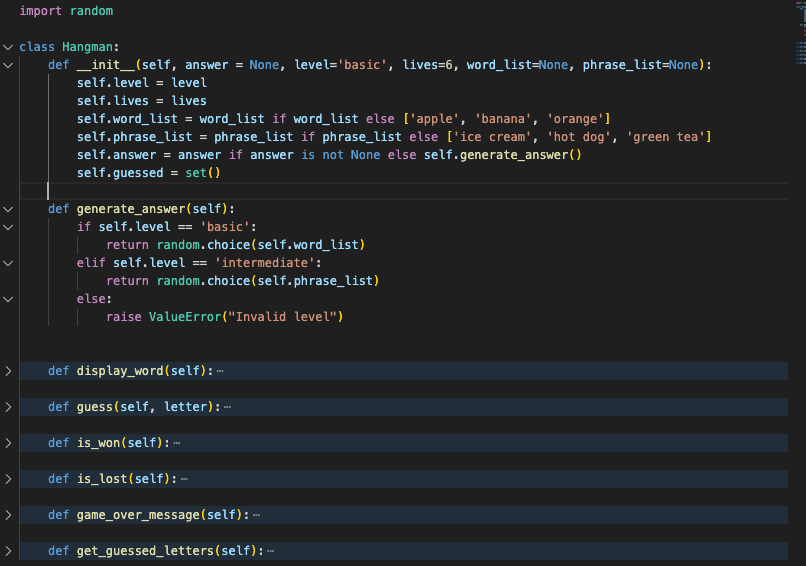
Step 8: Implement Level Selection



Run the test file, and the output shows that Hangman needs to define two levels.



Update hangman.py to include random, modify \_\_init\_\_() to accept levels, and add generate\_answer() in the Hangman class.

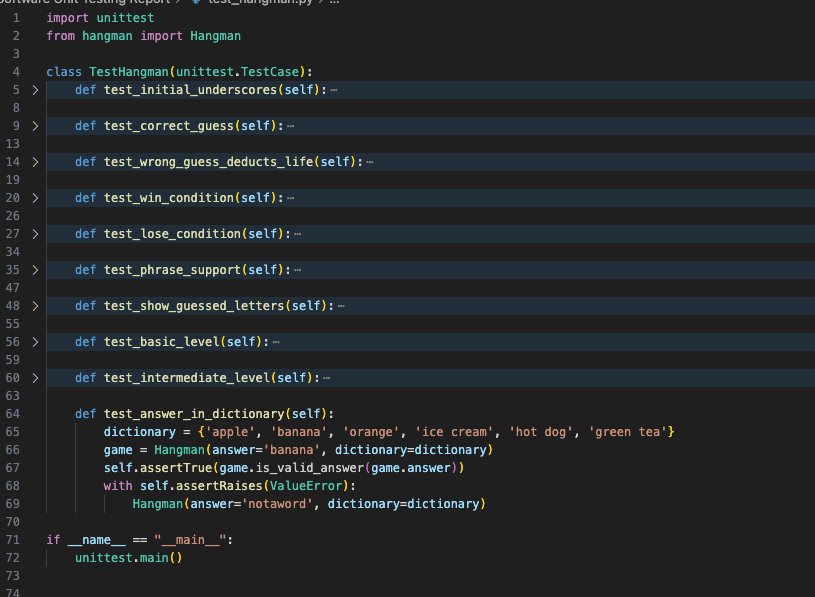


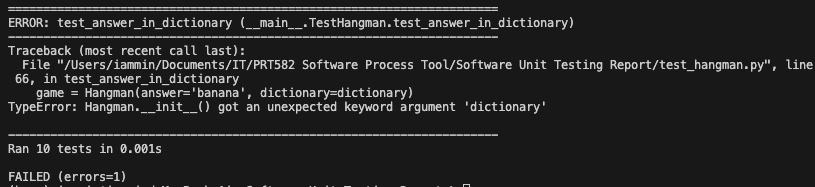
Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets the requirements.



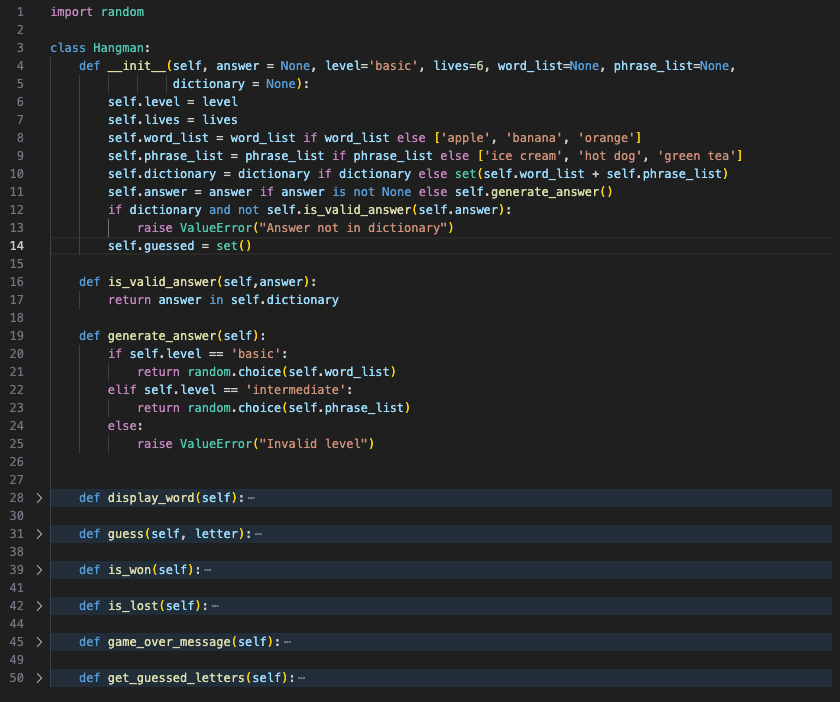
Step 9: Validate Words/Phrases from Dictionary

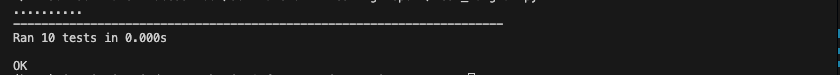
Add tests to check that the generated word/phrase is valid according to a dictionary.



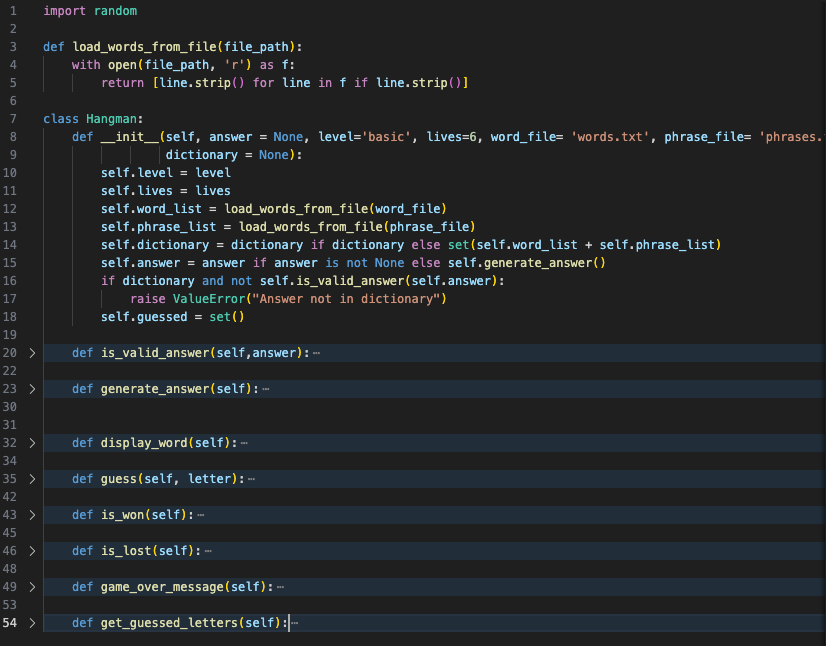
Run the test file, and the output is as follows:  


Update Hangman class to include a dictionary parameter, add is\_valid\_answer() and update generate\_answer( ).

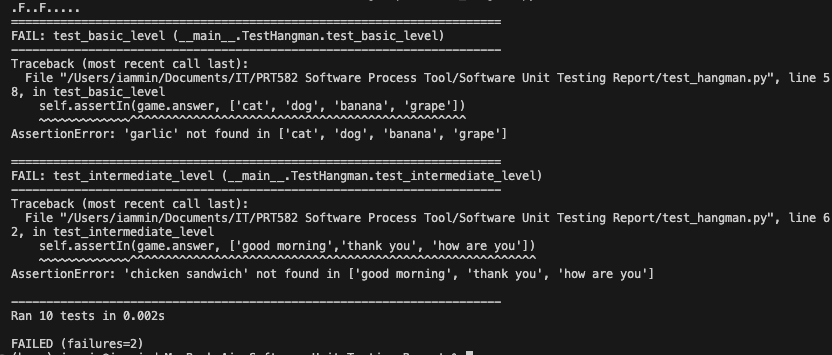


Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets the requirements.

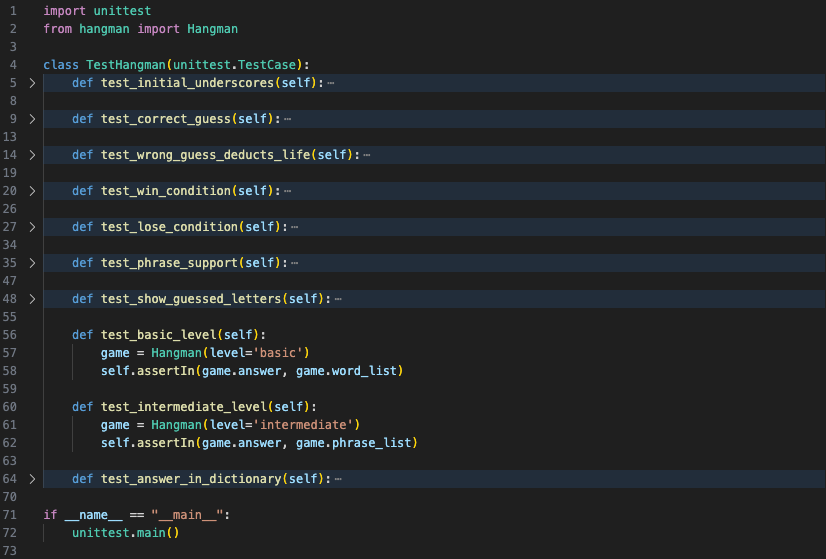
However, the default dictionary is too small, so in hangman.py, two files—words.txt and phrases.txt—each containing 100 randomly generated words or phrases, are loaded. The modifications are as follows:



Run the test file again, the results show that the tests fail.



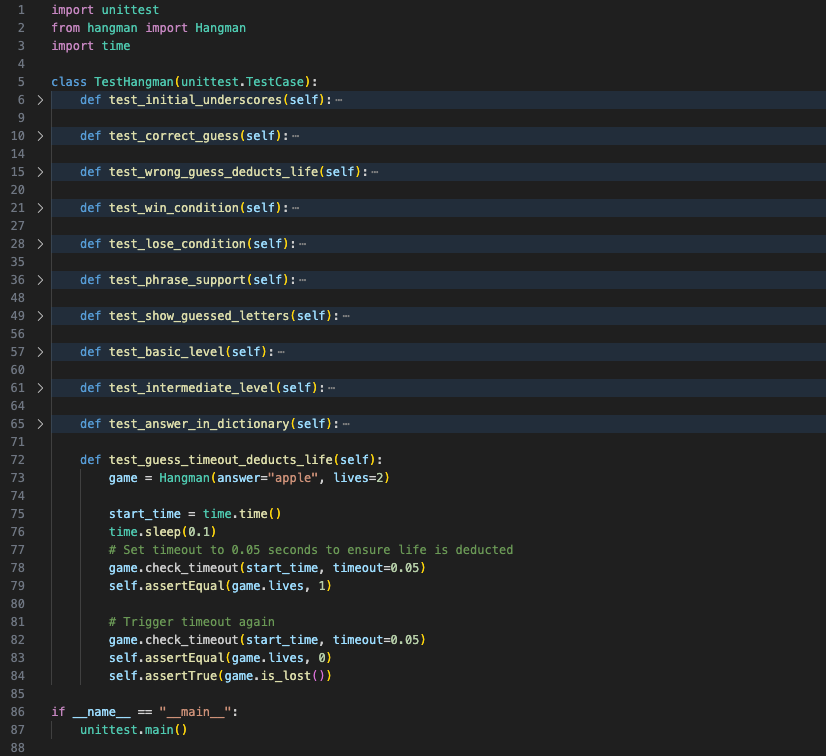
This is because the assertions in test\_basic\_level() and test\_intermediate\_level() in the test file use a fixed small list, while the game has updated the answer list. The answers generated randomly from the txt files. As a result, some words or phrases fall outside the fixed list, so the test file needs to be modified as follows:



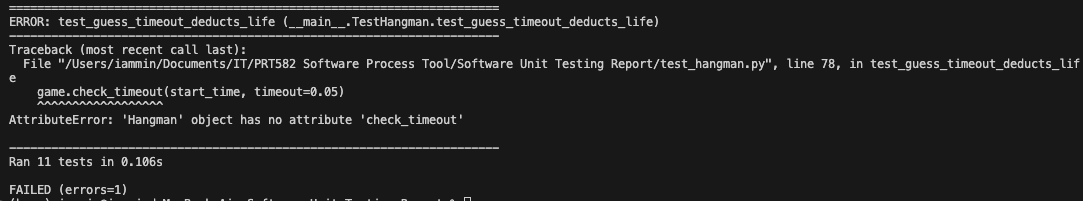
Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets the requirements.

Step 10: Timer for 15 Seconds per Guess

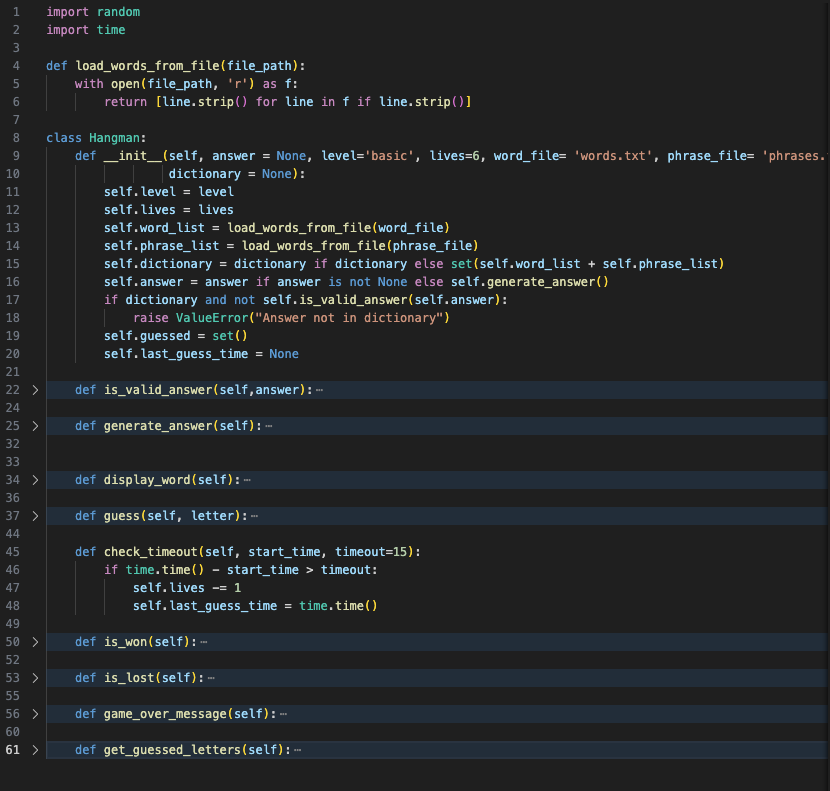
Import time in the test file, ddd test\_guess\_timeout\_deducts\_life( ) :



Run the test file, and the output is as follows:



Update Hangman class with self.last\_guess\_time = None and check\_timeout( ) method:



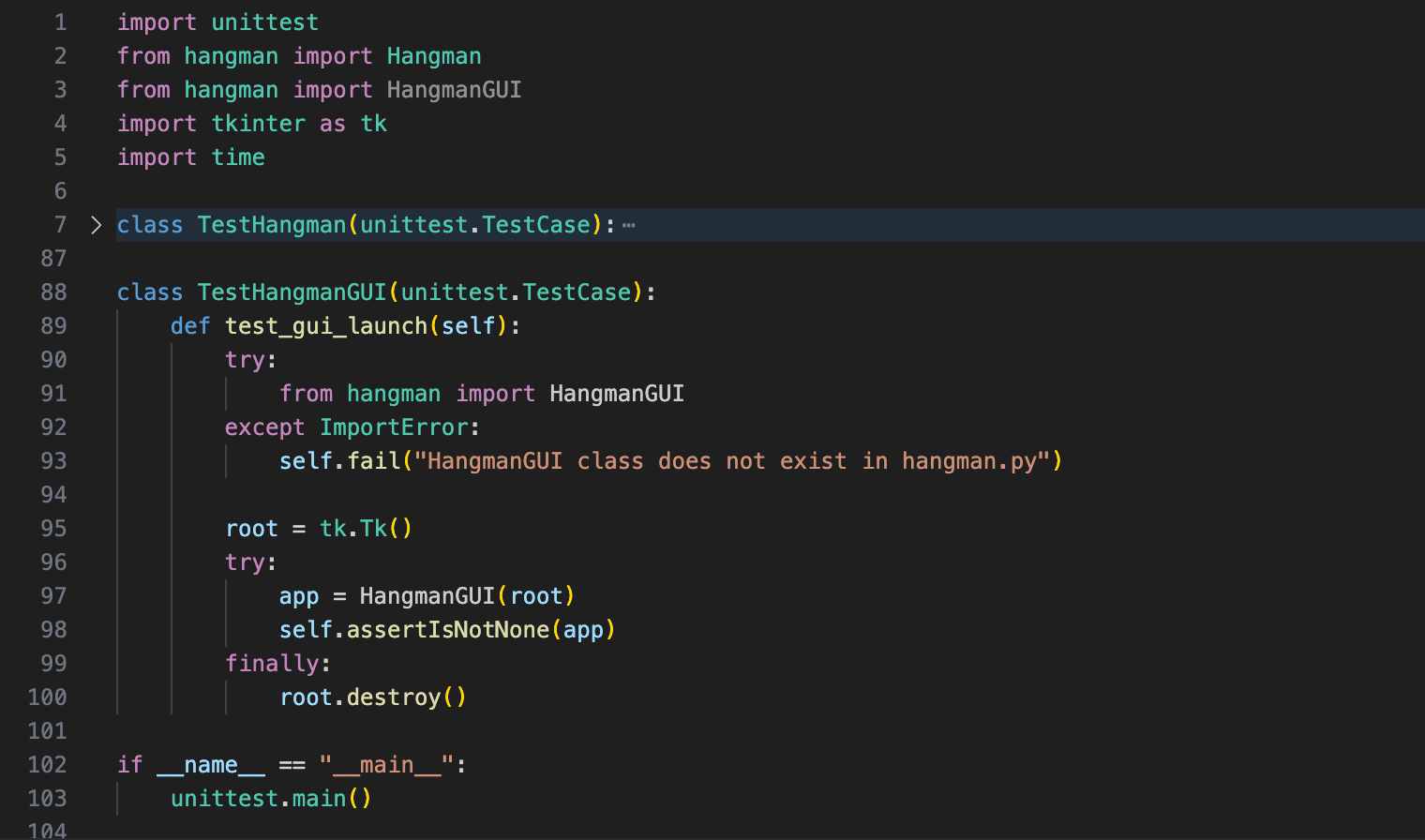
Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes, indicating that the function meets the requirements.



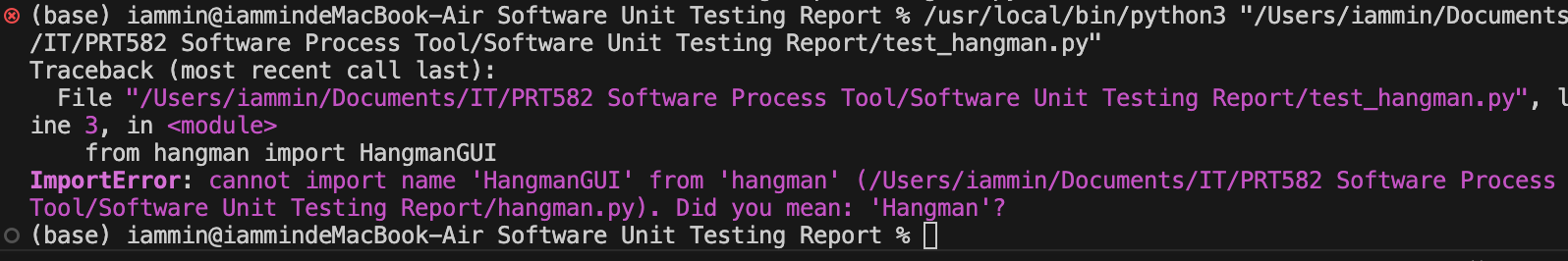
At this point, the basic requirement for the Hangman game has been mostly implemented. However, to make the game run smoothly and meet the requirement that “the timer must be shown,” a GUI can be implemented.

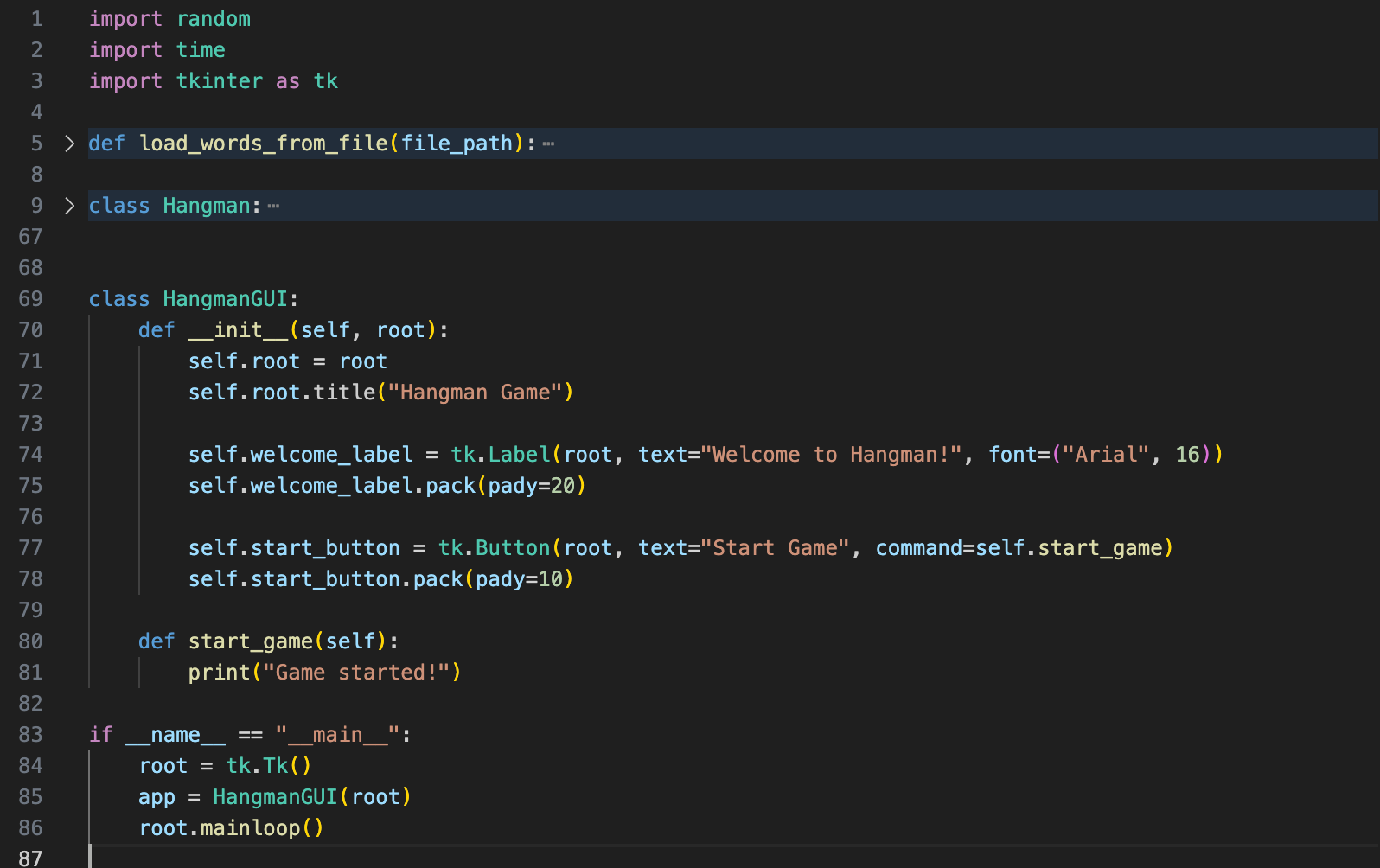
Step 11: Minimal GUI Test

Add a simple GUI test in the test file, importing tkinter and HangmanGUI.

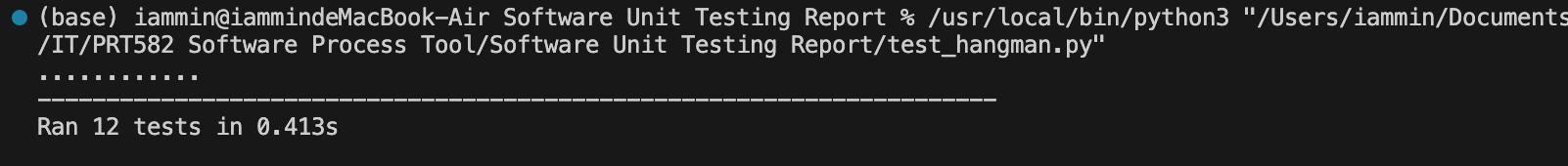


Run the test file, and an error occurs.



Add a minimal HangmanGUI class in hangman.py to pass the test.

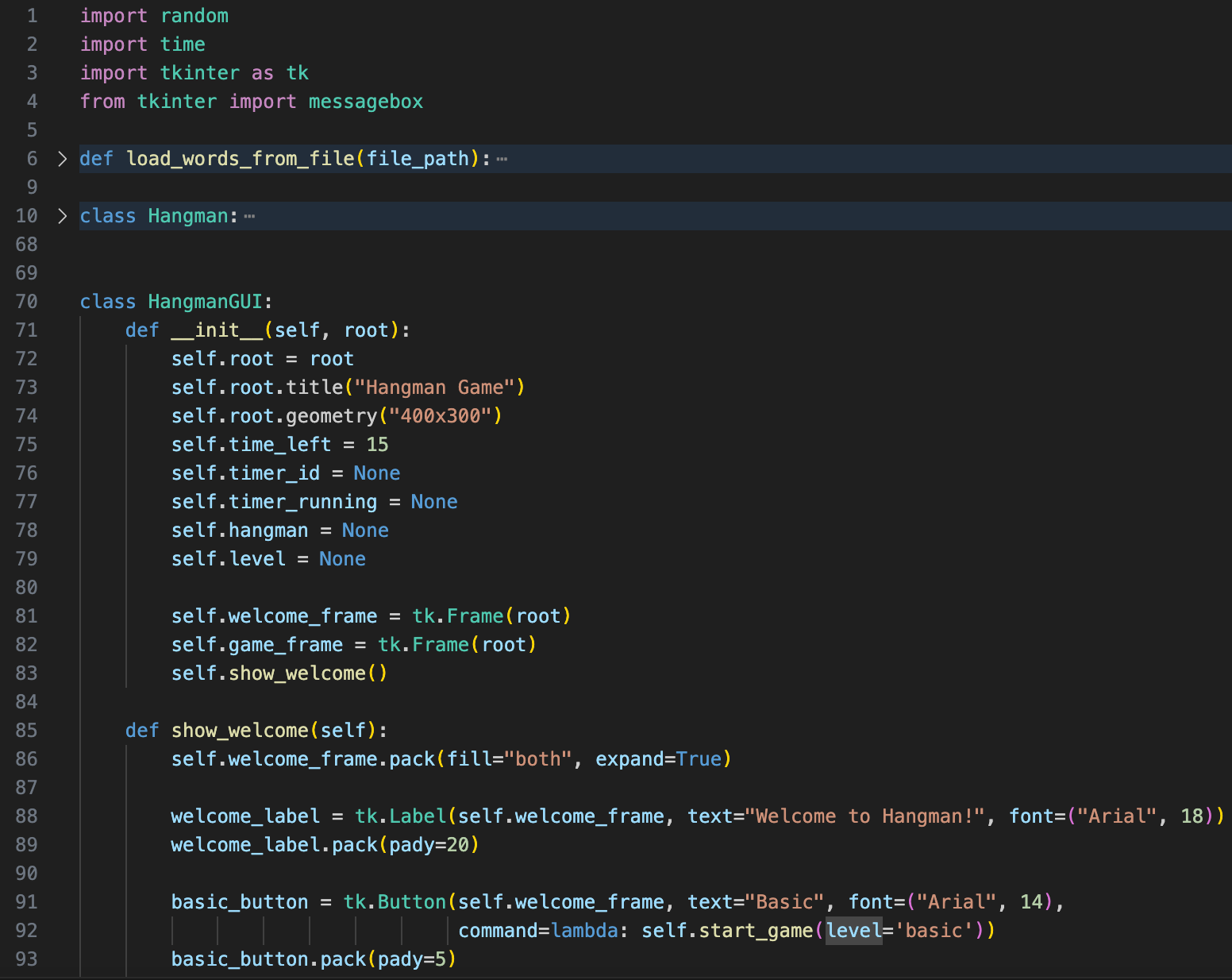
Execute hangman.py. It runs successfully without errors. Then run the test file, and the output shows the test passes.

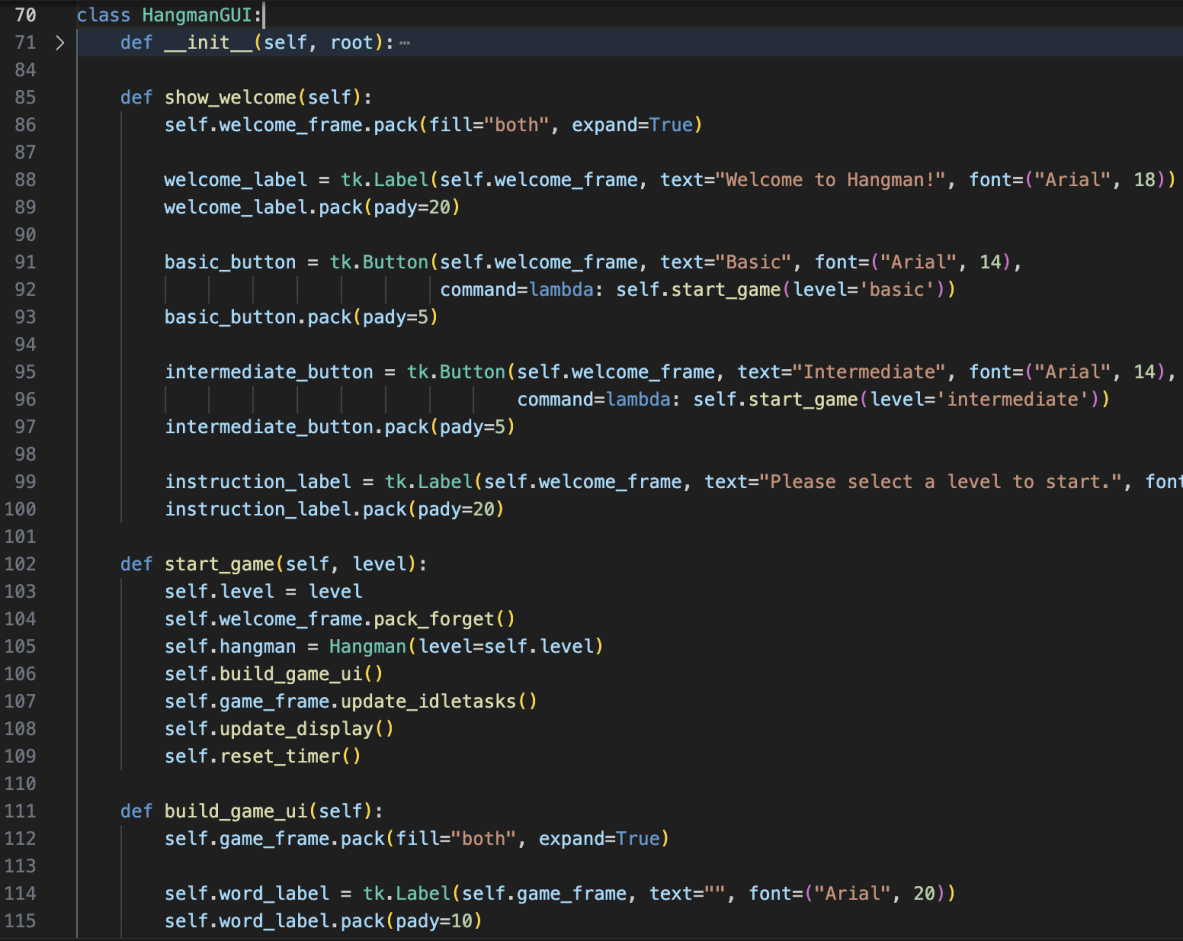


Step 12: Full GUI implementation with game loop and timer

Since the Hangman class already handles core logic, reuse existing tests.

Next, implement the full HangmanGUI in hangman.py including: Level selection; Timer countdown, Information prompts, Game loop.



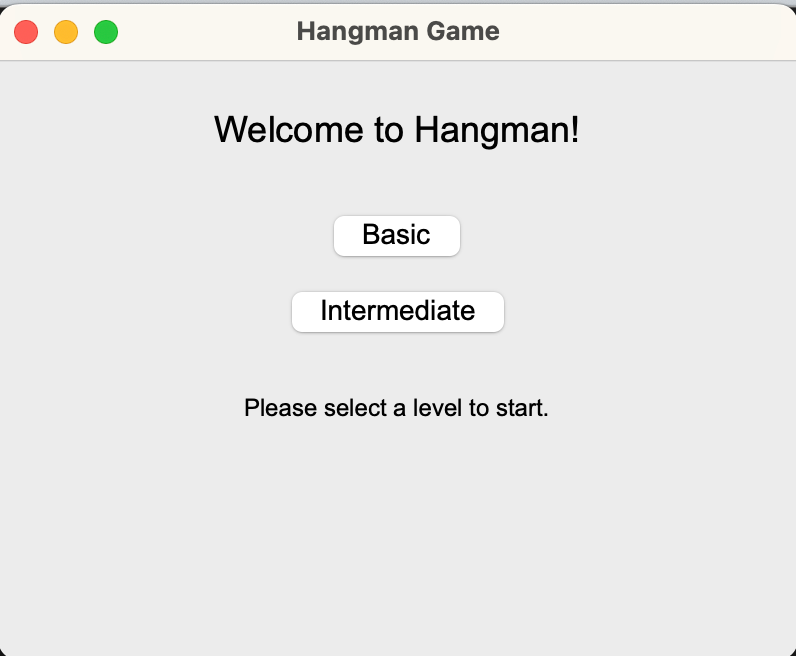




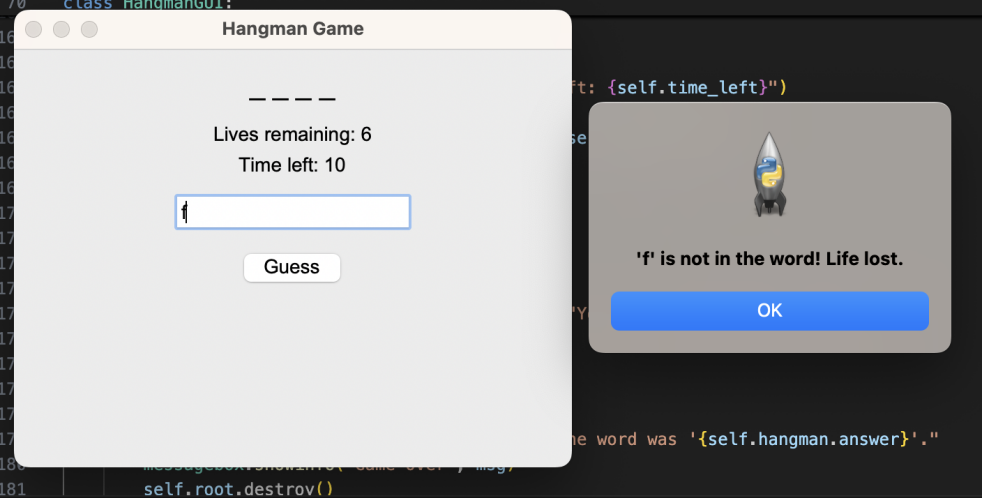


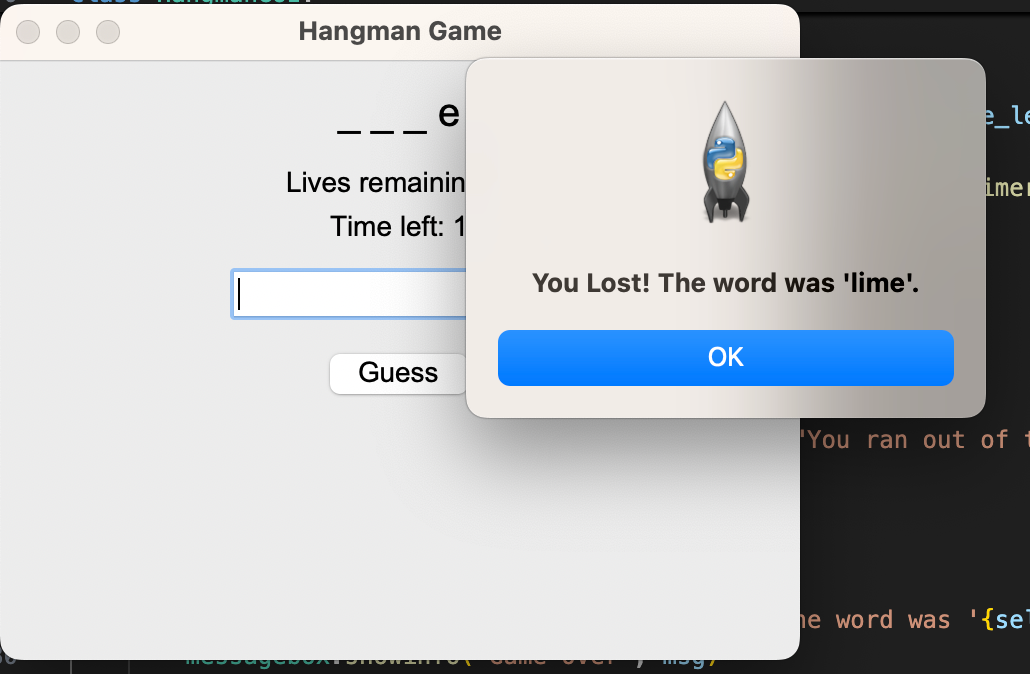


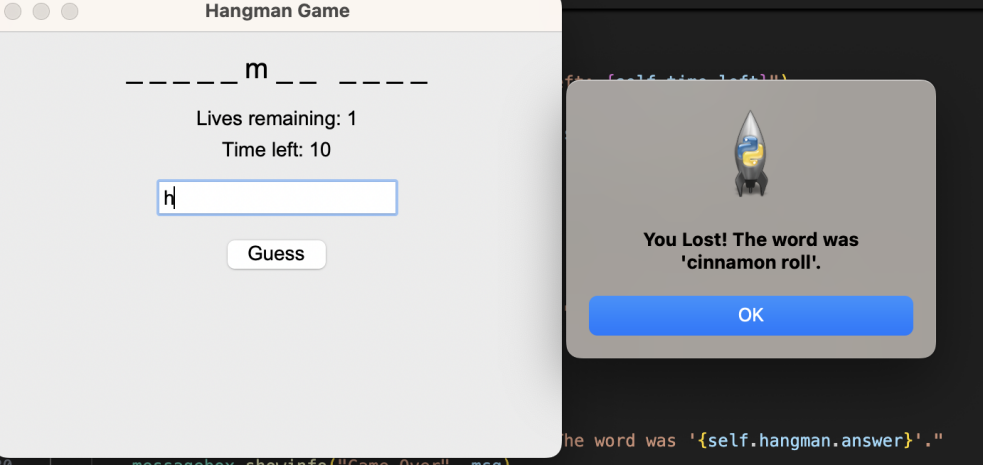
Running the code will display the following GUI window:

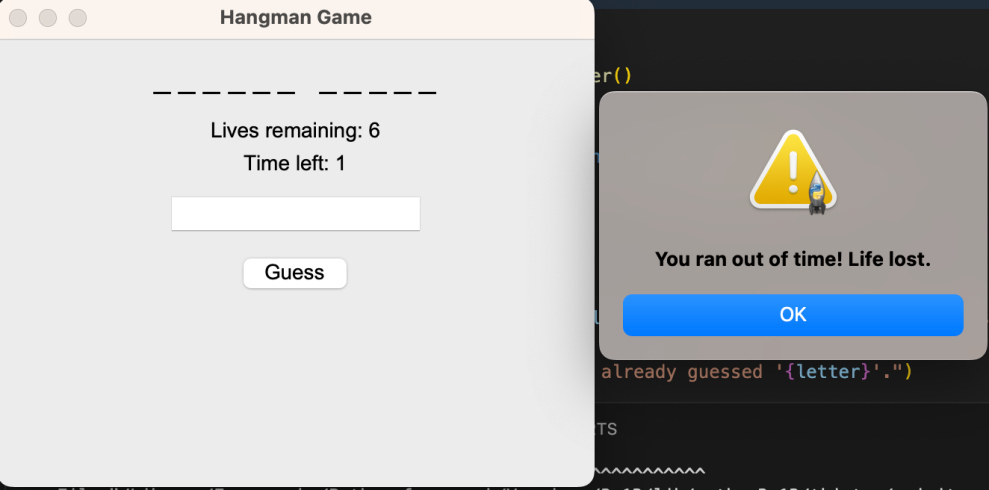


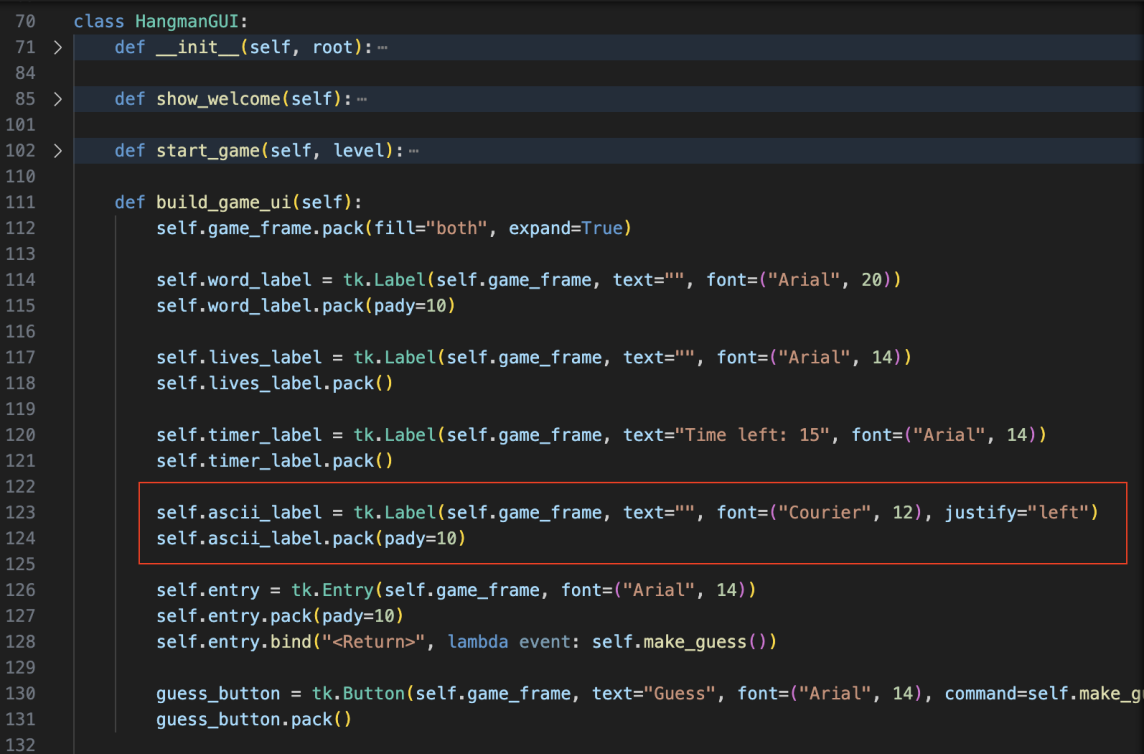
Run the program and test “Basic” and “Intermediate” levels. Wrong guesses, life lost ,correct guesses, timeout, and game over messages, timer and remaining life are all displayed correctly.









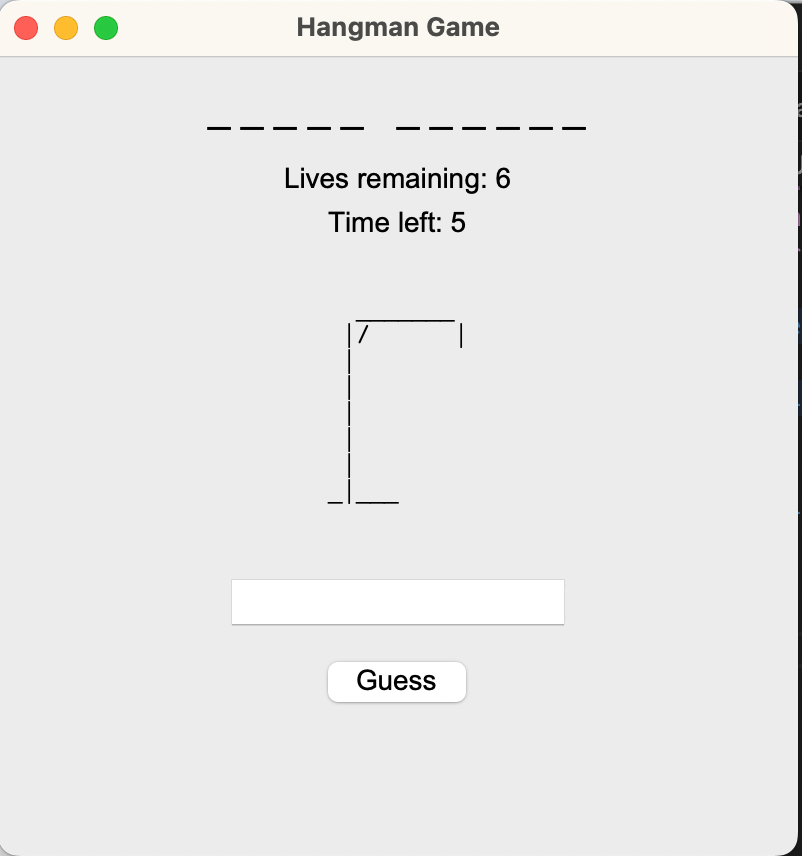
Since the interface looks somewhat empty and plain, Hangman ASCII art can be added to enhance the visual appeal. First, add the following code in the build\_game method of HangmanGUI:

Next, modify the update\_display method. Since the game starts with 6 lives, set 7 stages to gradually complete the Hangman figure as lives are lost:

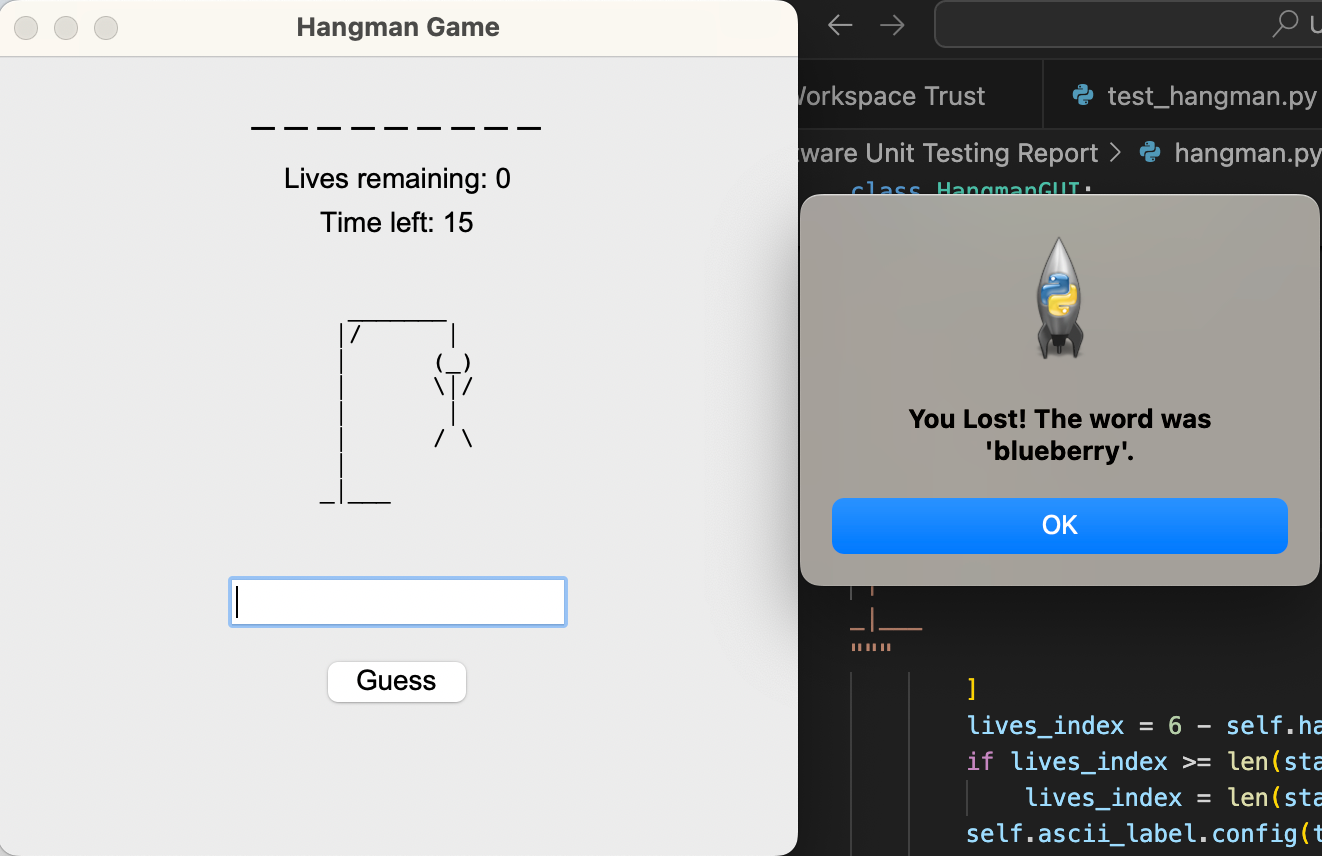




Finally, call self.update\_display() in the end\_game() method, and adjust the GUI window size in \_\_init\_\_. After making these changes, run the program; the game initially stage appears as follows:



When all lives are lost, it displays as follows:



The above completes the development of a simple Hangman game implemented using Test-Driven Development (TDD).

**Conclusion**

Through developing this Hangman game, I learned how to apply Test-Driven Development (TDD) to incrementally build a program, which is highly valuable for real-world software development. Firstly, TDD helps improve code quality by promoting modular, maintainable, and well-structured code, which is easier to debug and extend. Secondly, it provides developers with continuous feedback through automated tests, boosting confidence that the software behaves as expected. Fixing bugs early reduces later debugging effort, saves development time, and enhances overall efficiency. TDD also encourages thinking from the end-user perspective, helping to create a product that better meets user needs.

Although TDD may initially slow down development due to the time spent writing tests and learning the process, it can accelerate progress in the long term and improve test coverage, code cohesion, and maintainability. The experience of using TDD in this project demonstrated that it also acts as living documentation, supports future refactoring, and ensures that design decisions are verified through tests. These findings are consistent with previous research, which suggests that TDD improves internal and external software quality, enhances code cohesion, reduces complexity, and helps developers maintain a clear focus during incremental development [1].

In summary, the Hangman project showed that adopting TDD not only improves software reliability and maintainability but also strengthens a developer’s ability to anticipate and address potential problems, providing both technical and practical benefits in software development. The code for this project is available on GitHub at:

**Reference**  
[1] Z. Khanam and M. N. Ahsan, "Evaluating the Effectiveness of Test Driven Development: Advantages and Pitfalls," \*International Journal of Applied Engineering Research\*, vol. 12, no. 18, pp. 7705–7716, 2017. [Online]. Available: http://www.ripublication.com