

Software Unit Testing Report

Hangman game using TDD



Name: Alvin Monsi Mammoottil Lecturer: Charles yeo

Student Id: s290871

Contents

[Preparation for Test Driven Development. 2](#_Toc49594259)

[Test Cases 3](#_Toc49594260)

[Test Case 1: Generating random word. 3](#_Toc49594261)

[Test Case 2: Printing blank spaces. 4](#_Toc49594262)

[Test Case 3: Replace blank spaces with correctly guessed letter. 6](#_Toc49594263)

[Test Case 4: Deduction of player’s lives 8](#_Toc49594264)

[Test Case 5: Game over when lives left is less than 0 10](#_Toc49594265)

[Test Case 6: Provide appropriate feedback for entering inputs that are not letters. 12](#_Toc49594266)

[Test Case 7: Provide appropriate feedback for entering more than one letter as input. 14](#_Toc49594267)

[Test Case 8: Displaying previously used letters. 16](#_Toc49594268)

[Test Case 9: Feedbacks for guessing wrong letter, finding the word and player runs out of lives. 18](#_Toc49594269)

[Test Case 10: Game should not end when player enters a non-letter while 0 lives left. 20](#_Toc49594270)

[Refactoring 22](#_Toc49594271)

[Refactoring 1 : Refactoring was done after completing test case 1. 22](#_Toc49594272)

[Refactoring 2: Refactoring was done after completing test case 2. 24](#_Toc49594273)

[Refactoring 3: Refactoring was done after passing test case 9. 26](#_Toc49594274)

[Refactoring 4: Refactoring was done after passing test case 10. 30](#_Toc49594275)

[Removing Test Case codes from the program. 36](#_Toc49594276)

# Preparation for Test Driven Development.

The task for this Test-Driven Development is to create a hangman game.

Hangman is an old school favourite, a word game where the goal is simply to find the missing word. Your program will be slightly modified in which the player’s life can be represented by numbers instead of the ‘hangman’ picture.

The basic game requirements for the project:

* + 1. One word will be generated randomly
    2. Player will be presented with several blank spaces representing the missing letters the player needs to find.
    3. If the player’s chosen letter exists in the answer, then all places in the answer where that letter appear will be revealed.
    4. Every time the player guesses a letter wrong, the player’s life will be deducted.
    5. The player must find the missing word before the player’s life becomes zero.

Additional requirements added on by me

1. Alert the player when the player input a non-letter character.
2. Alert the player when the player inputs a string of characters rather than a single letter as guess.
3. Every time a player guesses a letter, the used letter is displayed to them so that they can refer to it when in doubt.
4. Appropriate feedbacks are provided when the user finds the word, user runs out of lives and when user guesses a wrong letter.

Python was chosen as the programming language selected for creating the game.

GitHub repository link: <https://github.com/Alvin2810/PRT582_Assignment_1>

# Test Cases

## Test Case 1: Generating random word.

**Description:** Checking if the program can generate words and if the generated words are random words.

**Test Steps:**

1. Run the program
2. Check if the game generated a random word.
3. Repeat the above steps four more times to see if a random word is generated each time.

**Expected Result:** Five random words will be generated.

**Actual Result:**

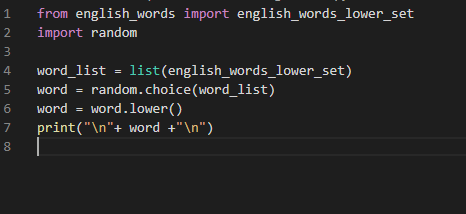
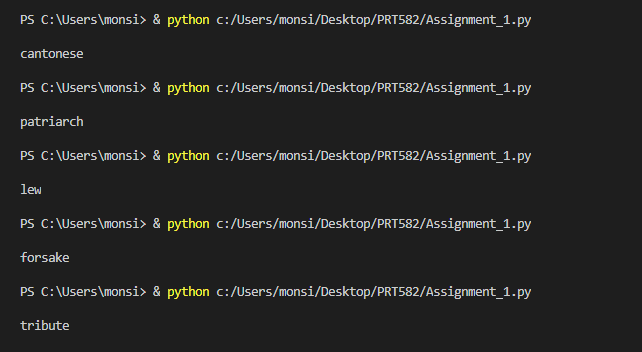


Figure 1: Code for random word generation.

Figure 2: The five random words created by the game.

According to the screenshots, one random word was generated each time program was run. Therefore, it can be said that the Actual result was same as the Expected result.

**Pass/Fail:** Pass

## Test Case 2: Printing blank spaces.

**Description:** Checking if the game presents the players with several blank spaces representing the missing letters the player needs to find.

**Test Steps:**

1. Run the program
2. Check if the number of blank spaces shown is the same number as the number of letters in the randomly generated word.
3. Repeat the steps 1 and 2 two more times to be certain.

**Expected Result:** Players will be presented with blank spaces representing the missing letters the player needs to find.

**Actual Result:**

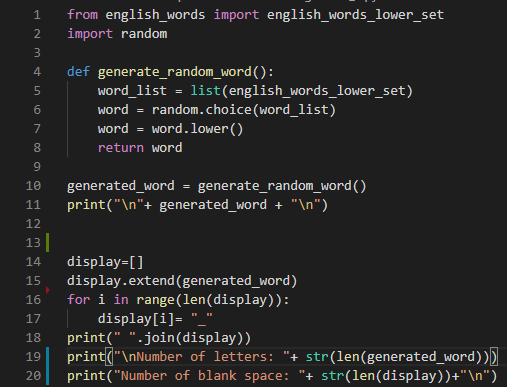


Figure 3: Code for random word generation and creating the blank spaces.

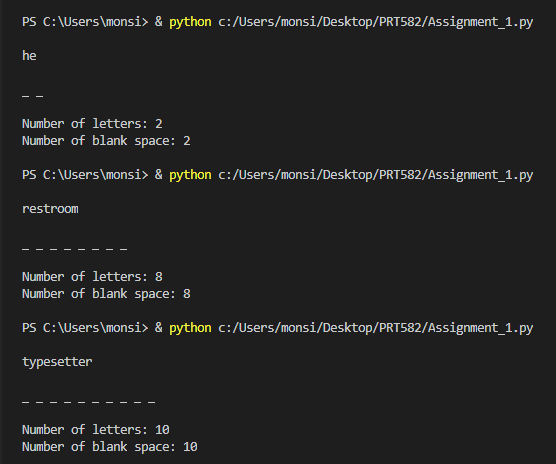


Figure 4: Proof that the game presents players with blank spaces representing the missing letters the player needs to find.

According to the screenshot above the number of blank spaces presented to the players has the same length as the randomly generated word. Therefore, the actual result is same as the expected result.

**Pass/Fail:** Pass

## Test Case 3: Replace blank spaces with correctly guessed letter.

**Description:** Checking if the game replaces all the blank spaces with the letter entered by the user if the user entered letter is part of the randomly generated word.

**Test Steps:**

1. Run the program
2. Input a letter into the given space.
3. Check if the blank space is replaced with the letter if the letter is a part of the generated word.
4. Repeat the steps 2 and 3 till all the blank spaces are replaced by letters.

**Expected Result:** If the player’s chosen letter exists in the answer, then all places in the answer where that letter appear will be revealed.

**Actual Result:**

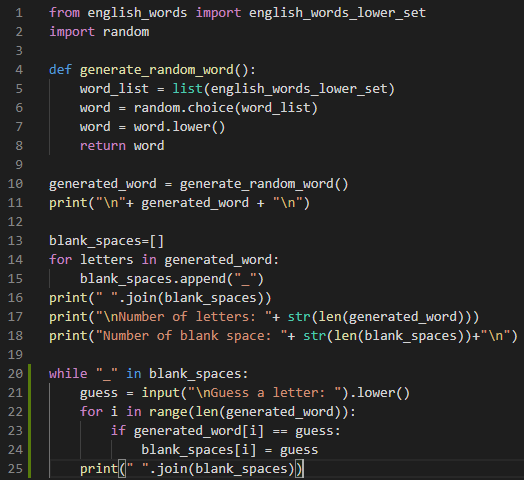


Figure 5: Code for the game with the newly added code highlighted.

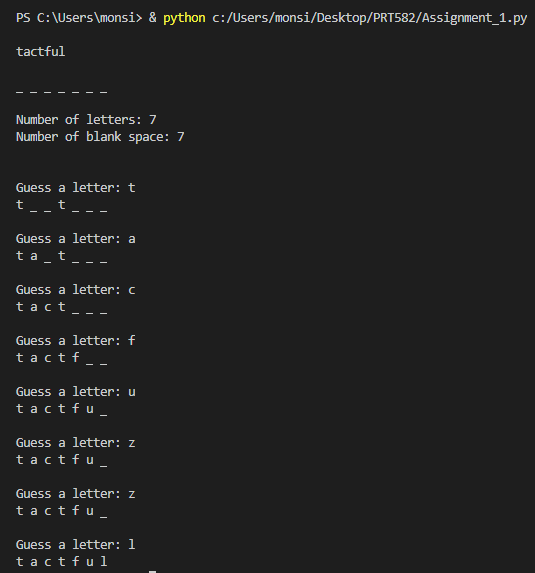


Figure 6: Proof that the blank spaces are replaced with user entered letters if they exist in the word.

According to the screenshot it can be understood that the blank spaces presented to the users will be replaced with the user entered guesses if they exist in the word. It also shows that no change is made to the blank spaces if the letters do not exist in the word.

**Pass/Fail:** Pass

## Test Case 4: Deduction of player’s lives

**Description:** Checking if the game deducts the players life every time a wrong letter is entered.

**Test Steps:**

1. Run the program
2. Check if the game shows the number of lives players have at the start of the game.
3. Input a letter that is not in the generated word and check if the players life is deducted by 1.
4. If it is deducted, then input a letter that is present in the word to make sure that the life is only deducted when a wrong letter is inputted.

**Expected Result:** The player’s life will be deducted by 1 every time a wrong letter is inputted by the player.

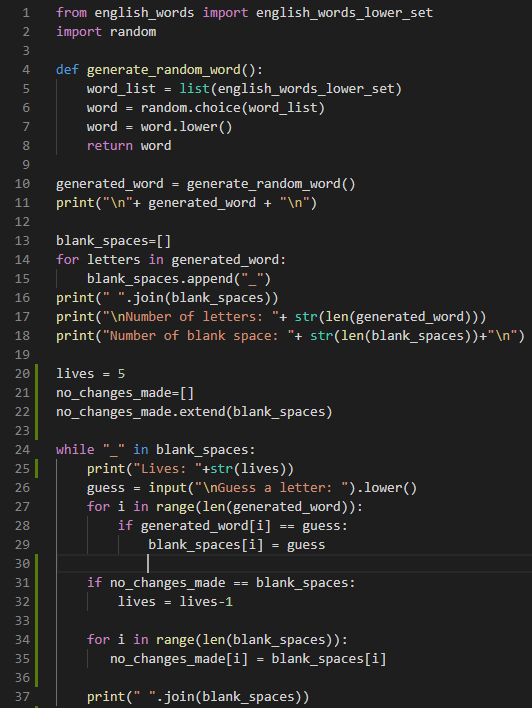
**Actual Result:** 

Figure 7: Code for the game with the newly added code highlighted.

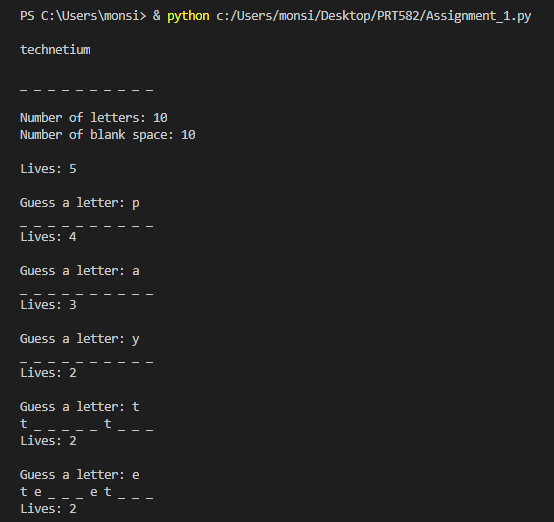


Figure 8: Proof that a life is deducted only when the player guesses the wrong letter.

According to the screenshots, it is seen that when a letter that is not in the word “technetium” is entered by the player then the player’s life is deducted by 1. It also shows that the life is only deducted when the wrong letter is entered as shown in the last three entries in the screenshot. Therefore, it can be said that the actual result was as expected.

**Pass/Fail:** Pass

## Test Case 5: Game over when lives left is less than 0

**Description:** Checking if the game is over when the player does not find the missing word before the lives left is zero.

**Test Steps:**

1. Run the program
2. Check if the game shows the number of lives players have at the start of the game.
3. Input a letter that is not in the generated word and check if the players life is deducted by 1.
4. Keep inputting letters not in the generated word till the lives is 0.
5. Check if the game ends if a wrong letter is inputted, when the player has no lives left.

**Expected Result:** Game will be over if the player does not find the missing word before the player’s life becomes zero.

**Actual Result:**

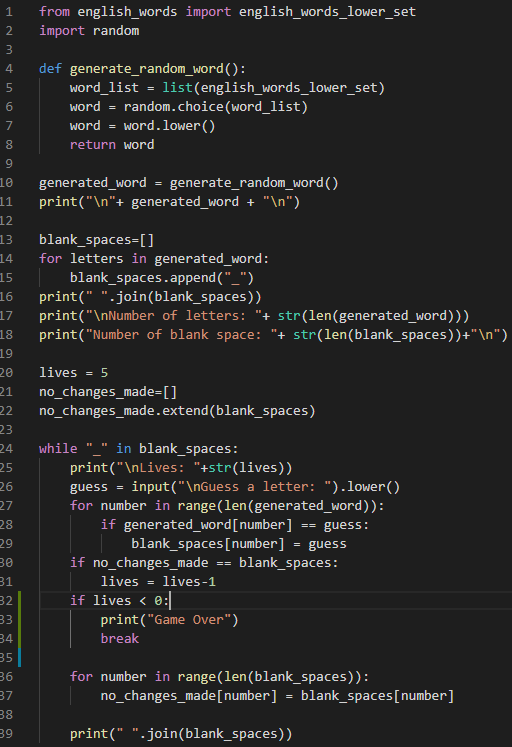


Figure 9: Code for the game with the newly added code highlighted.

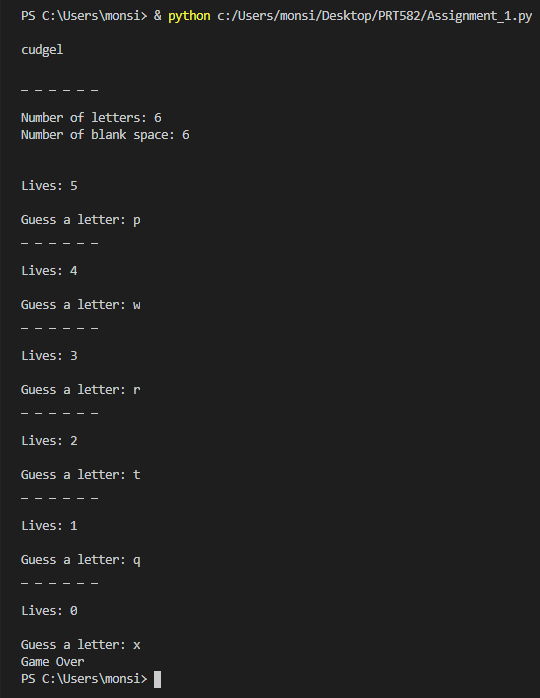


Figure 10: Proof that the game is over when the player runs out of lives.

According to the screenshot provided the player’s lives are deducted each time a wrong letter is guessed and when the player has 0 lives any wrong guess inputted afterwards results in game over. Therefore, the actual result is same as the expected result.

**Pass/Fail: Pass**

## Test Case 6: Provide appropriate feedback for entering inputs that are not letters.

**Description:** Checking if the game provides players with appropriate feedbacks for entering inputs that are not letters.

**Test steps:**

1. Run the program
2. Input a non-letter character as a guess.
3. Check that the players life is not deducted and that an appropriate feedback is displayed showing that the player should only enter letters and no other characters.

**Expected Result:** A feedback is displayed which instructs the player to only input letters as guess and not non-letter characters. The players life is also not deducted.

**Actual Result:**

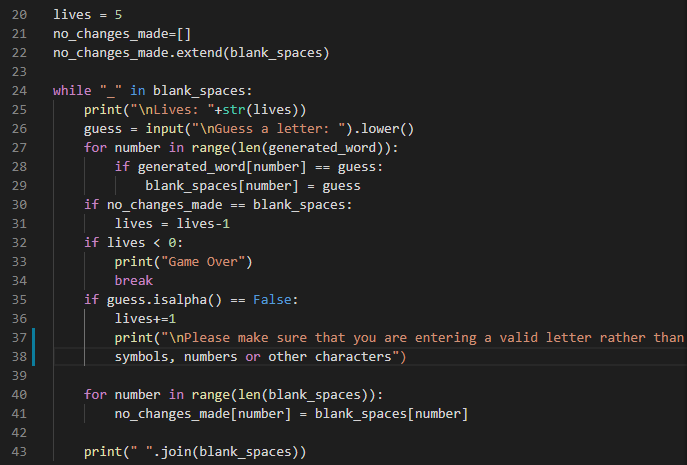


Figure 11: Code for the game with the newly added code highlighted.

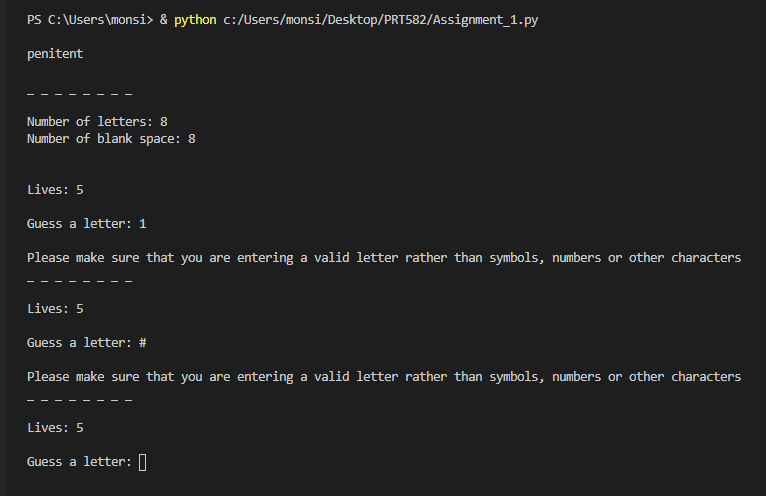


Figure 12: Proof that an appropriate feedback is provided to the players when a non-letter character is inputted as a guess.

The screenshot above shows that the player is given an appropriate feedback when the guess inputted by the player is not a letter. The player’s life also stays 5 and does not get deducted, therefore, the actual result is same as the expected result.

**Pass/Fail: Pass**

## Test Case 7: Provide appropriate feedback for entering more than one letter as input.

**Description:** Checking if the game provides players with appropriate feedbacks for entering more than one letter or characters as inputs.

**Test steps:**

1. Run the program
2. Input a string of characters or letters as a guess.
3. Check that the players life is not deducted and that an appropriate feedback is displayed showing that the player should only enter one letter as a guess.

**Expected Result:** A feedback is displayed which instructs the player to only input one letter as guess and not a string of letters. The players life is also not deducted.

**Actual Result:**

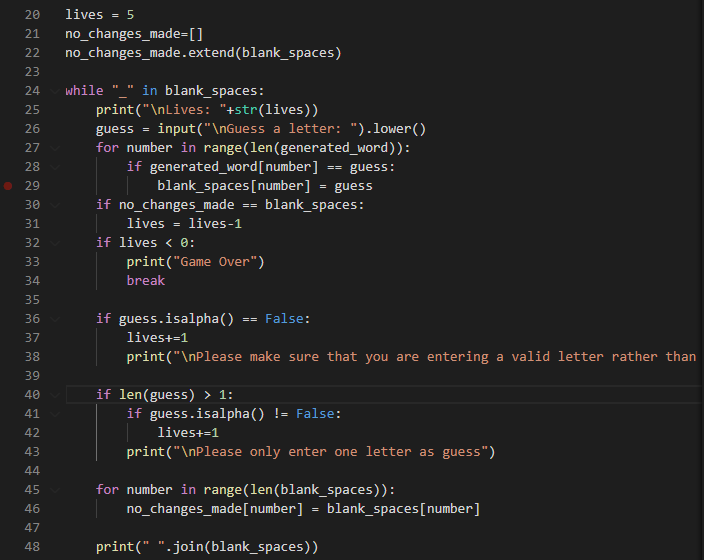


Figure 13: Code for the game with the newly added code highlighted.

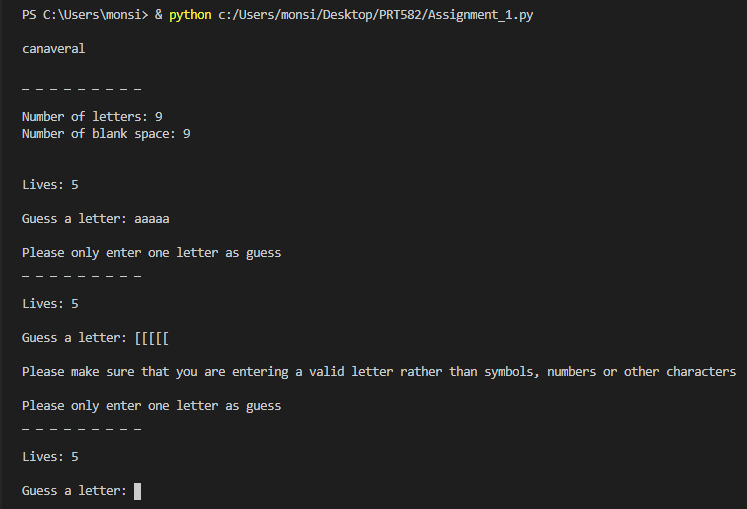


Figure 14: Proof that an appropriate feedback is provided to the players when a string of letters is inputted rather than a single letter.

The screenshot above shows that the player is given an appropriate feedback when the player inputs a string of characters as a guess instead of a single letter. The player’s life also stays 5 and does not get deducted, therefore, the actual result is same as the expected result.

**Pass/Fail: Pass**

## Test Case 8: Displaying previously used letters.

**Description:** Checking if the game displays a list of previously used letters to the player, so that the player knows what all letters were already inputted by them.

**Test steps:**

1. Run the program
2. Input a letter as a guess
3. Check if the game displays the used word to the player.

**Expected Result:** A list of letters that was previously inputted by the player is displayed on the screen.

**Actual Result:**

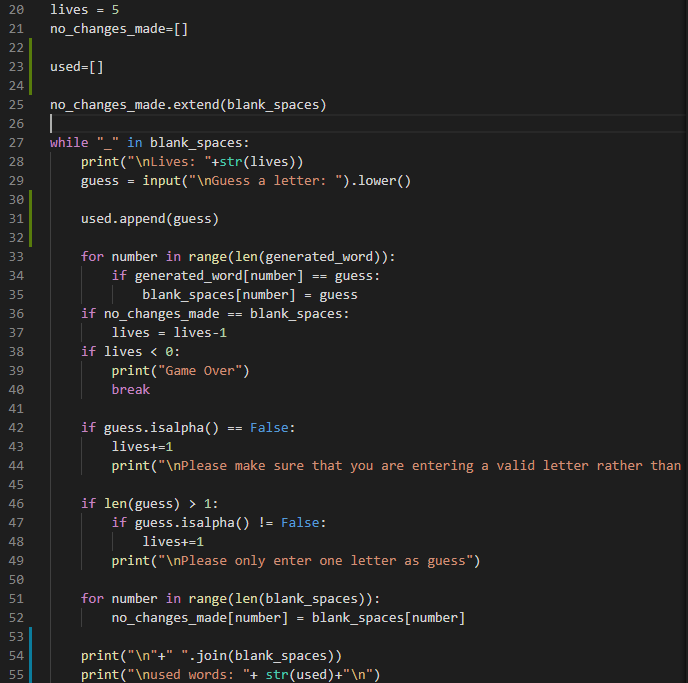


Figure 15: Code for the game with the newly added code highlighted.

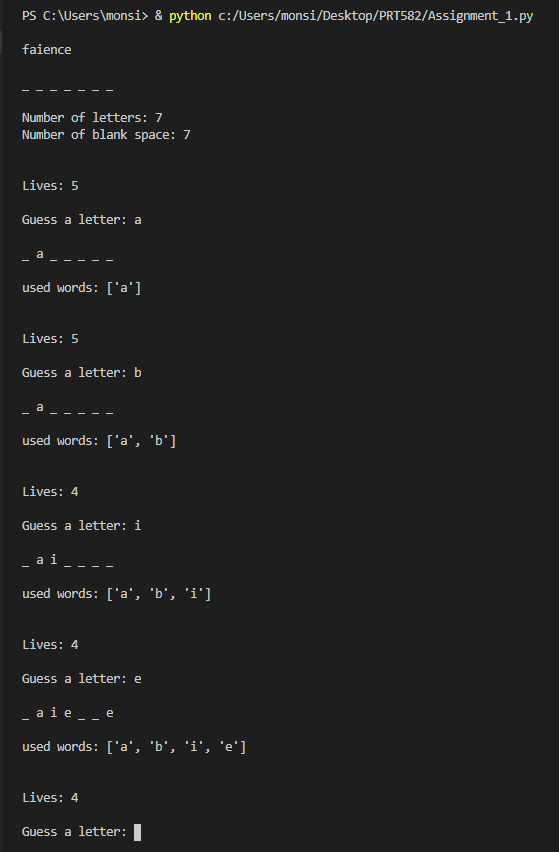


Figure 16: Proof that the previously used letters are displayed to the players.

According to the screenshot the letters previously inputted are displayed back to the players. Therefore, the actual is same as the expected result.

**Pass/Fail: Pass**

## Test Case 9: Feedbacks for guessing wrong letter, finding the word and player runs out of lives.

**Description:** Checking if the games provides players with appropriate feedbacks after:

1. Guessing a wrong letter
2. Finding the word
3. When players run out of lives.

**Test Steps:**

1. Run the program
2. Guess a wrong letter to see the feedback from the game.
3. Continue guessing wrong letters till the player runs out of lives to see the feedback provided when the game is over
4. Run the program again
5. Make the correct guesses and find the word to see what feedback is received by the player when they find the word.

Expected Result: An appropriate feedback is provided when the player guesses the wrong letter, runs out of lives and when they find the word.

Actual Result:

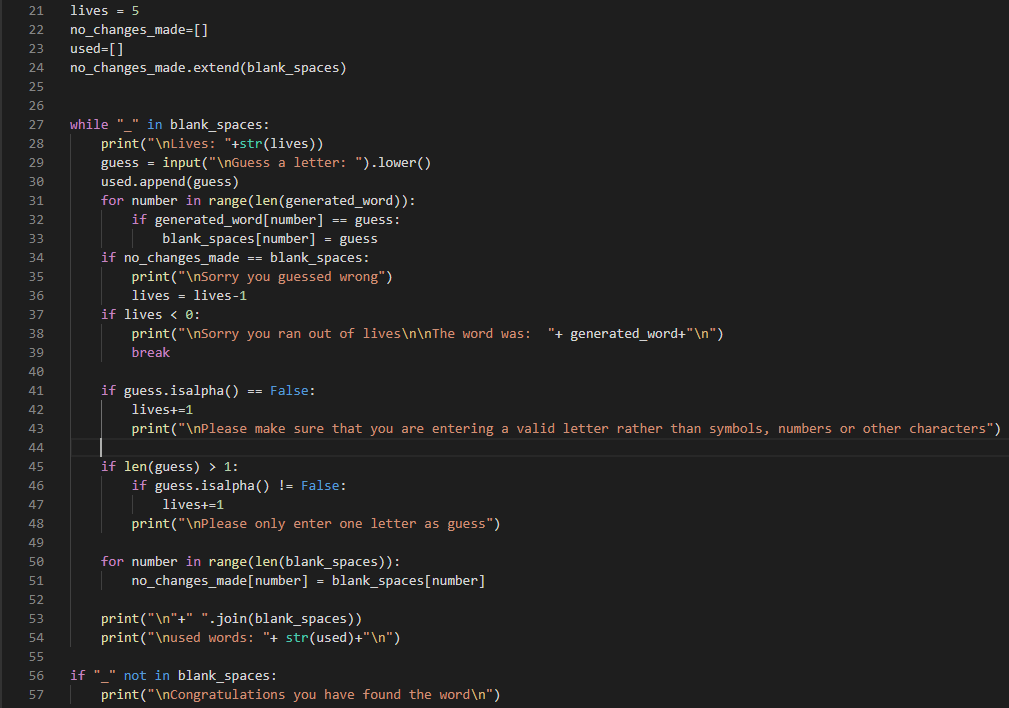


Figure 17: Code for the game with the newly added code highlighted.

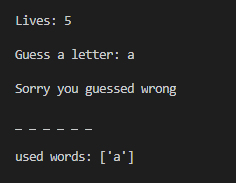


Figure 18: Proof that an appropriate feedback is provided when a player guesses the wrong letter.

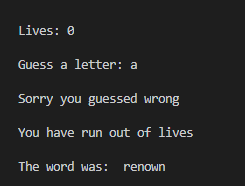


Figure 19: Proof that an appropriate feedback is provided when a player runs out of lives.

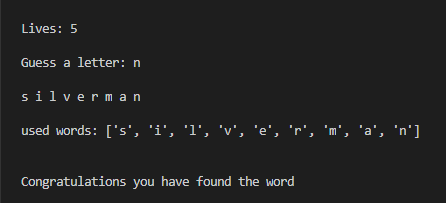


Figure 20: Proof that an appropriate feedback is provided when a player finds the word.

According to the screenshots an appropriate feedback is provided when the player guesses the wrong letter, runs out of lives and when they find the word. Hence the actual result is same as the expected result.

**Pass/Fail: Pass**

## Test Case 10: Game should not end when player enters a non-letter while 0 lives left.

It was seen that since the “if isnotaletter(guess)” condition is below the “if lives<0” condition when the users have no lives left and enter a non – letter or a string of characters that will result in end of game rather than a feedback saying that the players should enter one letter per guess.

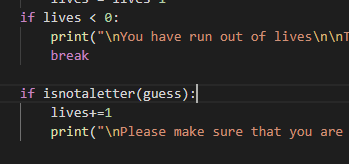


Figure 21: Code before making any changes to make it pass the test.

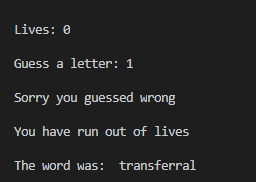


Figure 22: Proof that the game did not pass the test first time through.

**Test Description:** Check if the game ends when the player enters a non-letter character or a string of characters when the player has 0 lives left.

**Expected Result:** The Game does not end and prompts the player to enter a new guess which is not a non-letter character.

**Actual Result:**

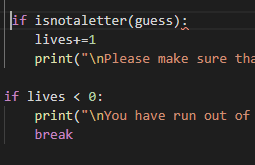


Figure 23: Code after making changes to make it pass the test.

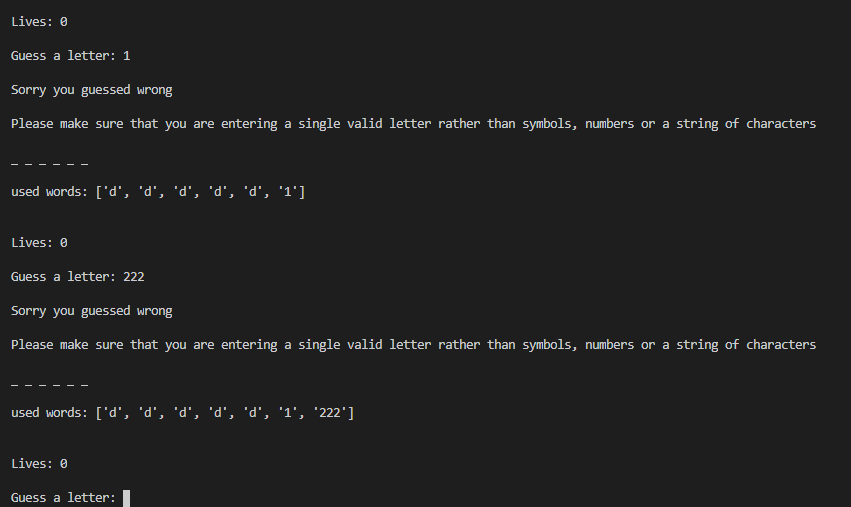


Figure 24: Proof that the game passes the test.

According to the above screenshot, when the players lives left is 0 and if they enter a non-letter character or several characters the game will not end instead will prompt them to enter an actual letter. Therefore, the actual result is same as the expected result.

**Pass/Fail: Pass**

# Refactoring

## Refactoring 1 : Refactoring was done after completing test case 1.

**Code Smell:** The code for creating a random word was long and involved several functions therefore, making it difficult to understand.

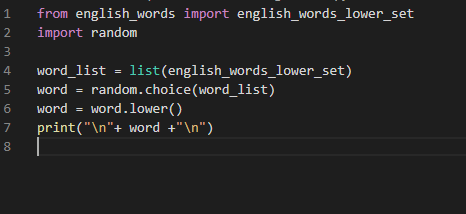


Figure 25: Before Refactoring 1

Changes made to implement the solution:

* Defined a new function called “generate\_random\_word()”. **(Highlighted by Yellow)**
* Extracted code and moved into “generate\_random\_word()” function. **(Highlighted by Green)**
* A new variable called “generated\_ word” was created to carry the value of the generated random word. **(Highlighted by Blue)**

This helps to have one function called “generate\_random\_word()” for creation of the random word rather than the long method used before.

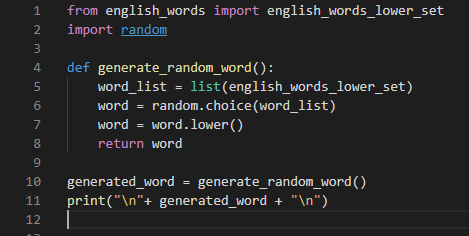


Figure 26: After Refactoring

**Running Test Case 1 Again:**

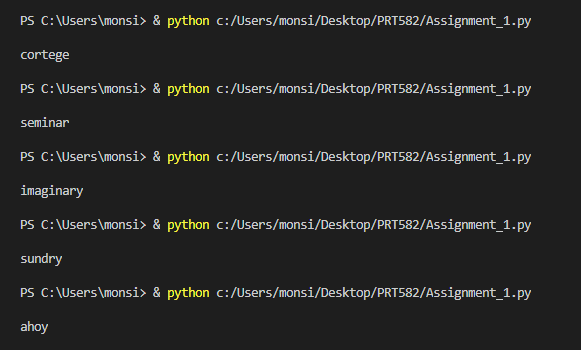


Figure 27: The five random words created after refactoring.

**Pass/Fail:** Pass

## Refactoring 2: Refactoring was done after completing test case 2.

**Code Smell:** Some variable names used were not easily understandable and function “display.extend(generated\_word)” was deleted since a better working function was available.

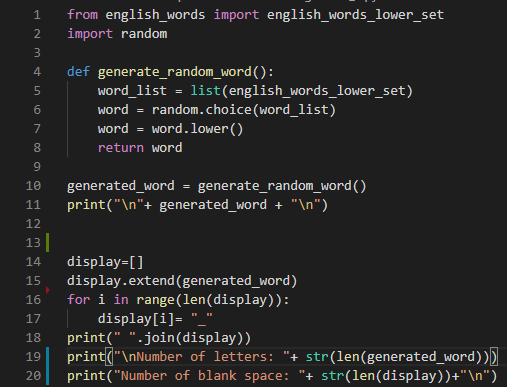


Figure 28: Before Refactoring 2

Changes made to implement the solution:

* Changed name of the list from “display” to “blank\_spaces” to make it more understandable for anyone reading the code. **(Highlighted in yellow).**
* Deleted the “display.extend(generated\_word)” and used “blank\_spaces.append()” function instead of it.
* The for loop was slightly modified to make it much more understandable and enhance readability. **(Highlighted in Green).**

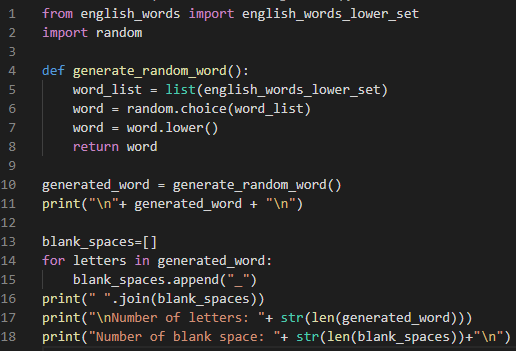


Figure 29: After Refactoring 2

**Running Test Case 2 Again:**

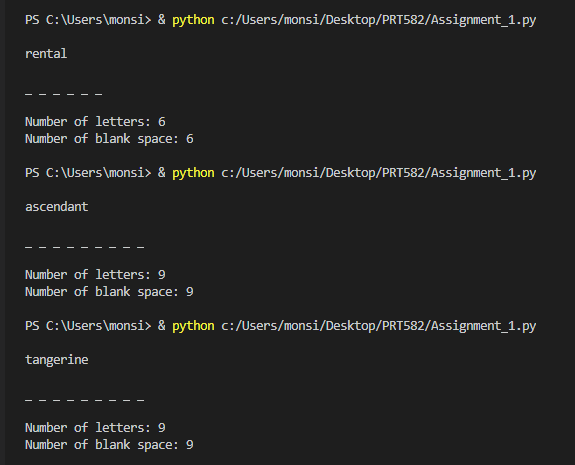


Figure 30: Proof that the game presents players with blank spaces representing the missing letters the player needs to find.

**Pass/Fail:** Pass

## Refactoring 3: Refactoring was done after passing test case 9.

**Code Smell:** Variables were declared in different areas of the code, making it harder to spot while wanting to make changes to the variables.

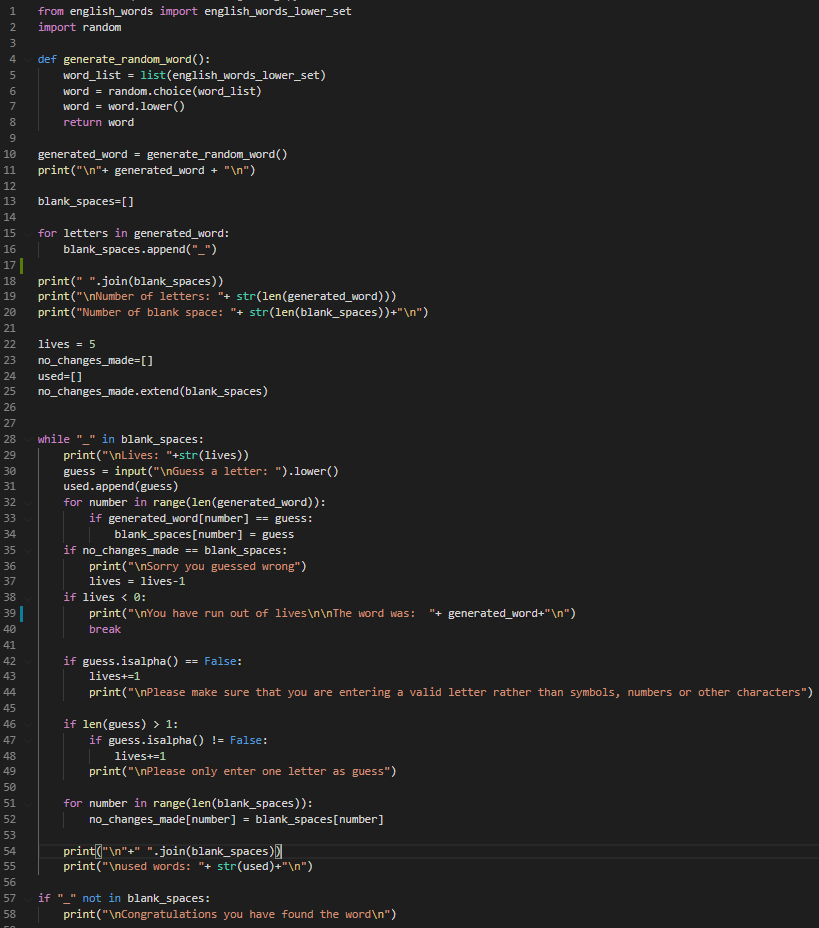


Figure 31: Before Refactoring 3 (variables are declared in different areas)

Changes made to implement the solution:

* Extract all the declaration of variables and move them into one area.

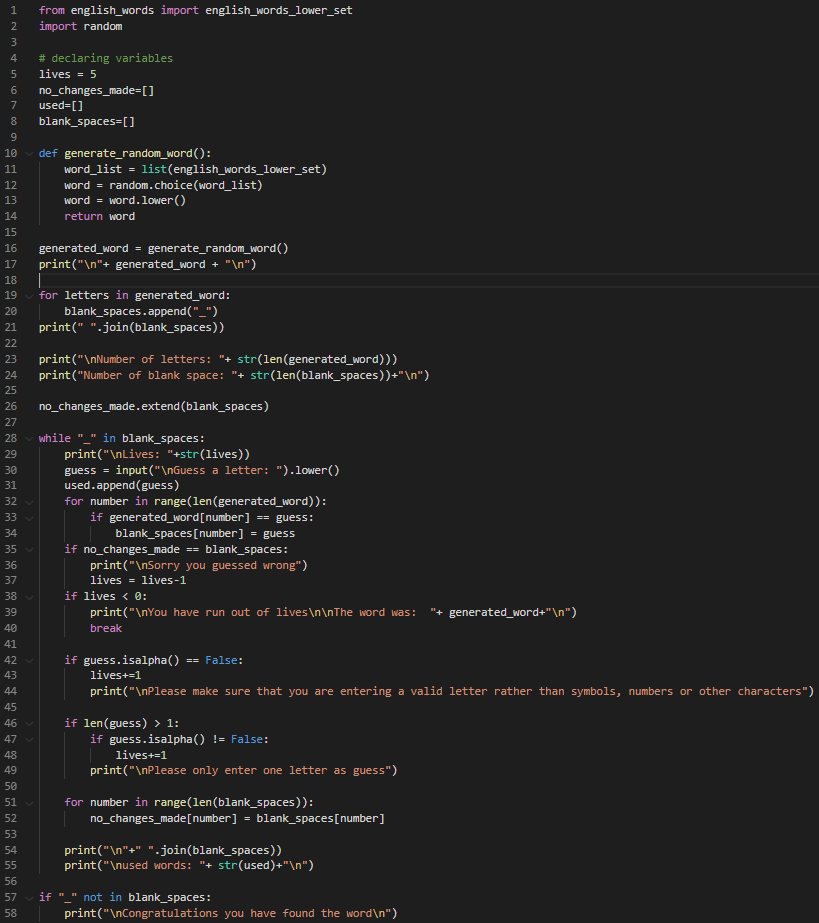


Figure 32: After Refactoring 3 (variables are declared in one area)

**Code Smell:** Since the conditionals “if guess.isalpha()==False:” and “if len(guess) > 1:” returns similar results when true they both can be consolidated into a single method.

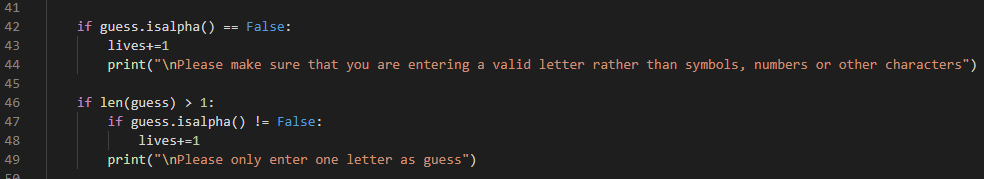


Figure 33: Before Refactoring 3 (Both if conditions)

Changes made to implement the solution:

* The two conditionals were extracted and moved into one function called isnotaletter(guess)”.

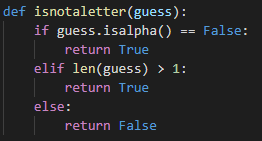


Figure 34: After refactoring 3 (The function to which the if conditions were moved to.)

* The two if conditions were then replaced with a single if condition “if isnotaletter(guess)”.

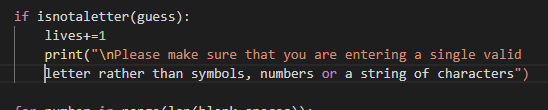


Figure 35: After refactoring 3 (The single if condition that replaces both if conditions)

**Running test case 6 and 7 again:**

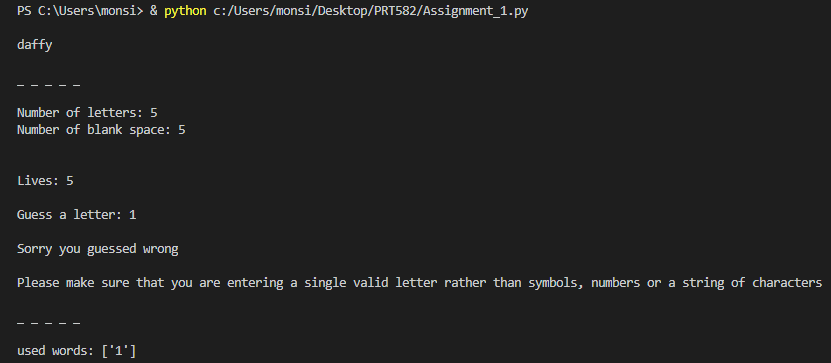


Figure 36: Proof that an appropriate feedback is provided to the players when a non-letter character is inputted as a guess.

**Test Case 6: Pass**

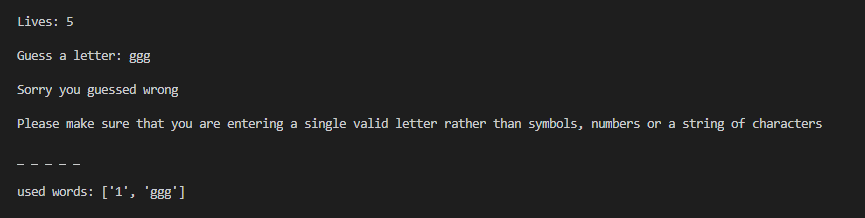


Figure 37: Proof that an appropriate feedback is provided to the players when a string of letters is inputted rather than a single letter.

**Test Case 7: Pass**

## Refactoring 4: Refactoring was done after passing test case 10.

**Code Smells:** Variables “lives” and “used” are not easily understandable.

Changes made to implement the solution:

* Renamed the variables “used” and “lives” to “used\_letters\_list” and “lives left” respectively.

Figure 38: Before refactoring 4

Figure 39: After refactoring 4

Figure 41: After refactoring 4

Figure 40: Before refactoring 4

**Code Smells:** Had “guess = input(“\nGuess a letter: “).lower()” in one line this could make it confusing for someone reading the code.



Figure 42: Before refactoring 4

Changes made to implement the solution:

* The code was split into two to make it less confusing:
  + “guess = input(“\nGuess a letter: “)”
  + guess.lower()

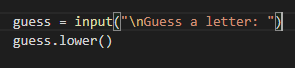
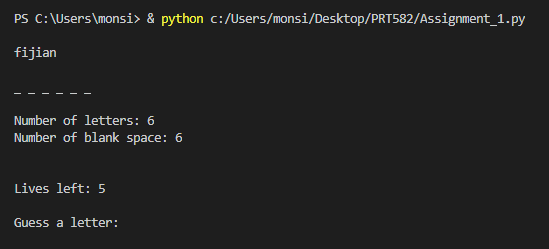
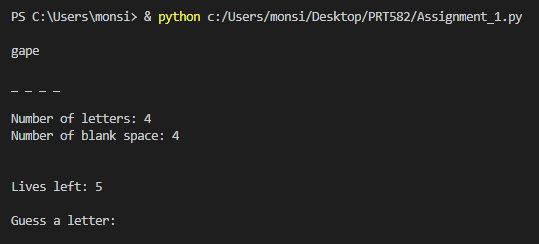


Figure 43: After refactoring 4

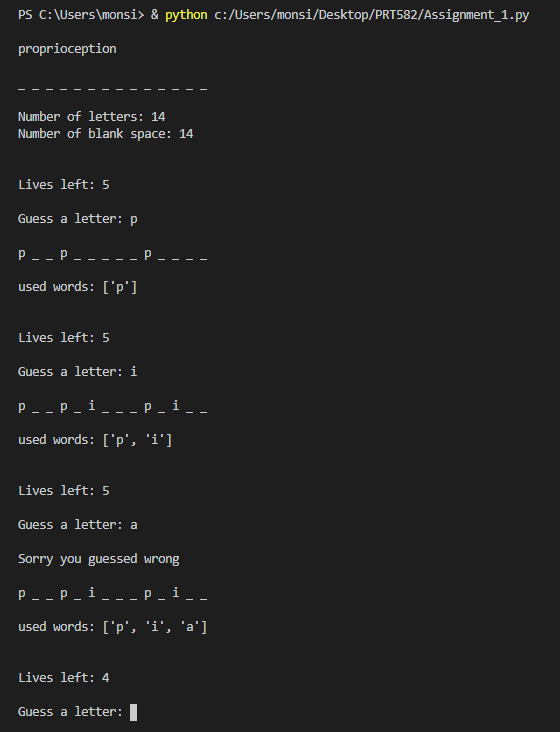
**Running tests 1 to 10 again:**





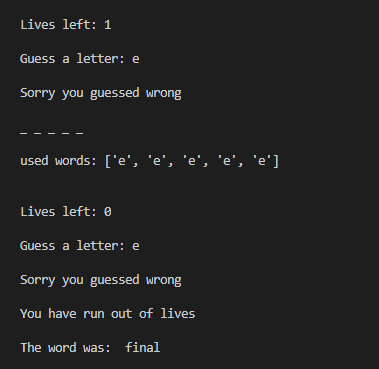
**Test Case 1 Passed:** The above screenshots show that each time the program is run a random word is generated.

**Test Case 2 Passed:** The blank spaces displayed matches the number of letters in the word.

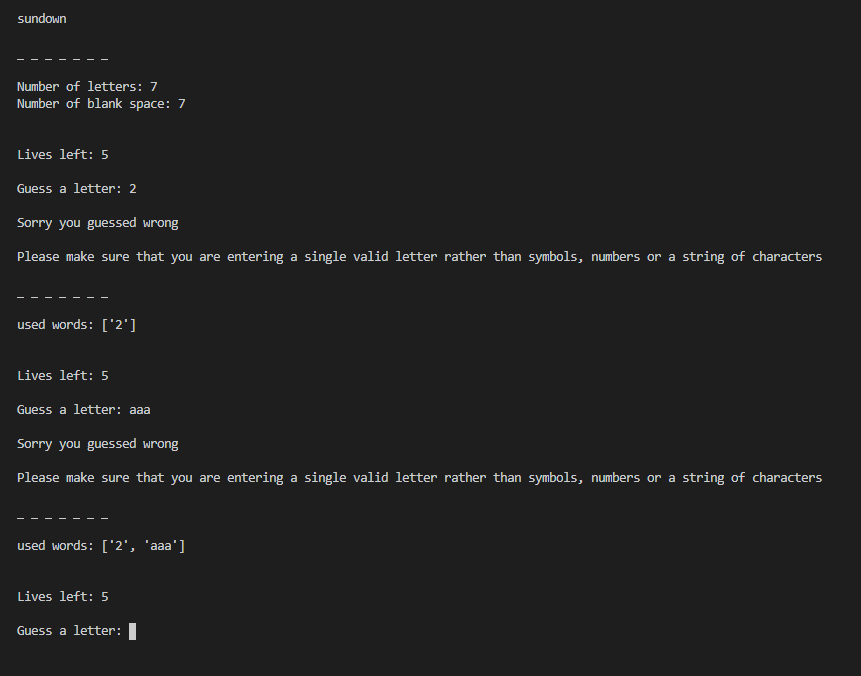


**Test Case 3 Passed:** The above screenshot shows that when the player enters a letter that is present in the word the blank space is replaced by the letter.

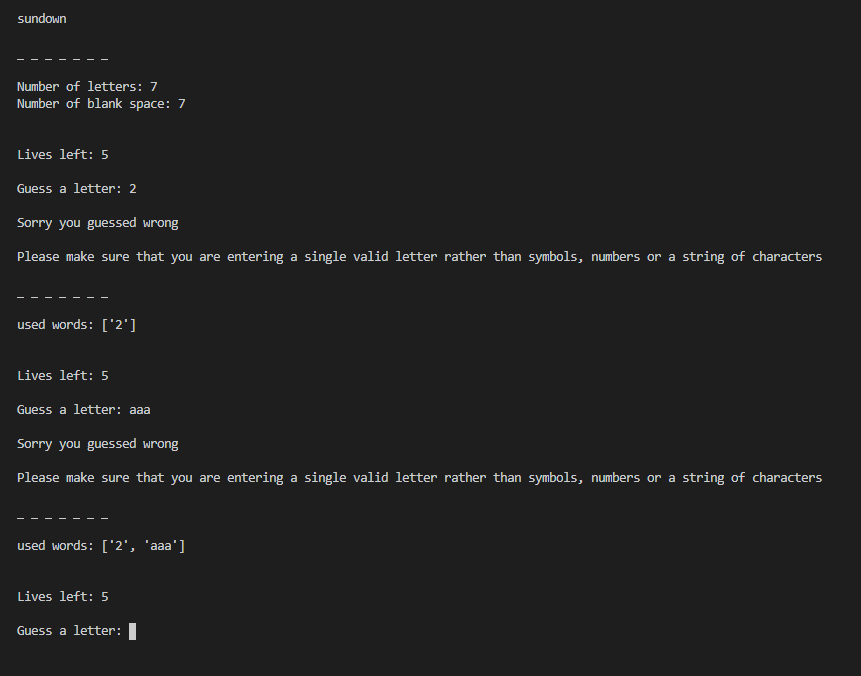
**Test Case 4 Passed:** It also shows that when a letter not in the word was entered in this case letter “a” 1 life is deducted from the players total life.



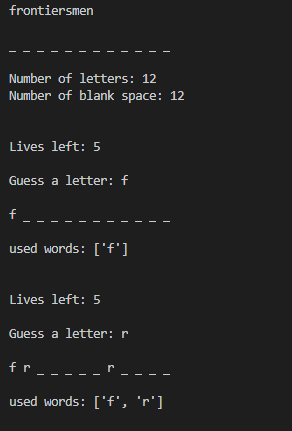
**Test Case 5 Passed:** The above screenshot above clearly shows that when the player runs out of lives the game is over.



**Test Case 6 Passed:** The above screenshot above clearly shows that when the player enters a non-letter character as a guess an appropriate feedback is provided.

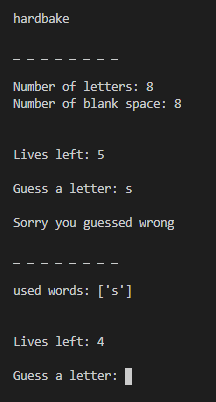


**Test Case 7 Passed:** The above screenshot above clearly shows that when the player enters more than one letter as a guess an appropriate feedback is provided.

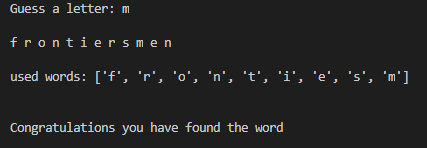


**Test Case 8 Passed:** The above screenshot above clearly shows that the program displays a list of used letters for the player to see.

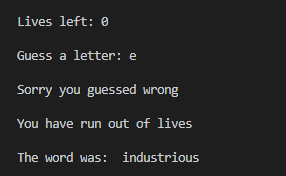
**Test Case 9 Passed:**



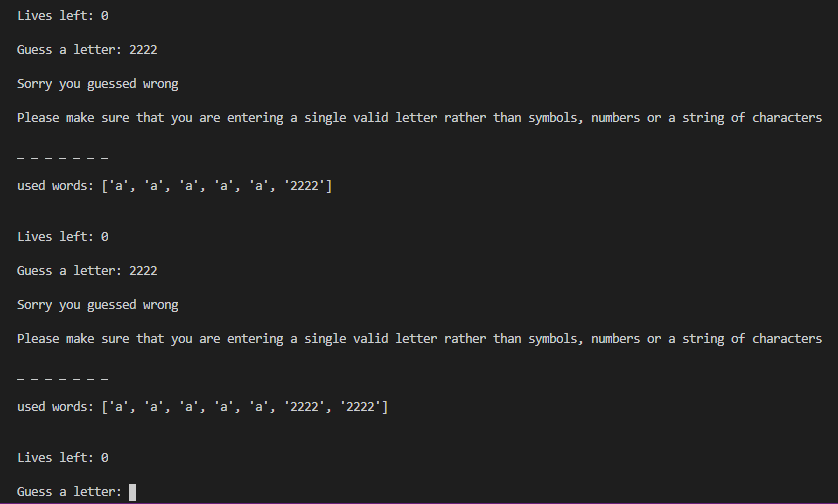
* The above screenshot above clearly shows that the program provides the user an appropriate feedback when entering a wrong guess



* An appropriate feedback was provided when the player found the word.



* An appropriate feedback was provided when the player ran out of lives.

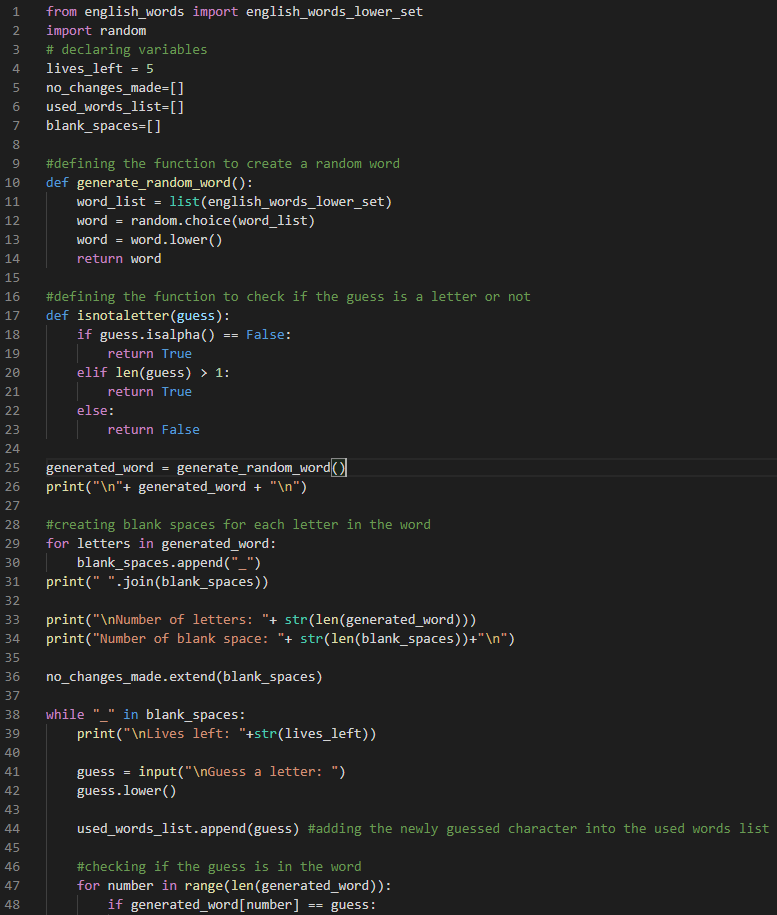


**Test Case 10 Passed:** The above screenshot above clearly shows that the game does not end if the player enters a non-letter character or a string of characters when the player lives left is 0.

# Removing Test Case codes from the program.

Since the game passed all the test cases it is possible to remove all the code in the game that was used for checking if the program was able to pass the test cases. For example, printing the generated random word.

The Codes being removed are highlighted below.



Once the codes are removed the coding process is completed and the game is ready to play.