Hangman Project

Table of Contents

[Test-driven Development (TDD) 1](#_Toc49613617)

[Development before beginning test cases 1](#_Toc49613618)

[Requirements 1](#_Toc49613619)

[Test Cases 1](#_Toc49613620)

[Refactoring 5](#_Toc49613621)

[Code Smells 5](#_Toc49613622)

# Test-driven Development (TDD)

This project has been built following a Test-Driven Development approach. This approach utilizes planned test cases that must be passed before continuing to the next test case. Throughout the project, code will be developed to pass each test case one after another, however it is likely that with the development of code for new test cases, older tests will have to be run again. When running older tests some issues may arise that indicate code smells. These will then be addressed, and solutions developed for them.

## Development before beginning test cases

I want this application to utilize a UI for display and input, therefore before I begin developing to pass test cases, I decided to construct a blank Tkinter window.

## Requirements

The project test cases are derived from the following requirements.

1. One word will be generated randomly
2. Player will be presented with a number of 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.

## Test Cases

The following is the series of test cases in order of their initial development. There is a likely chance that code will need to be refactored throughout development which will be documented within each test case.

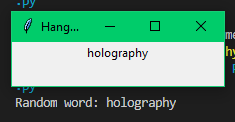
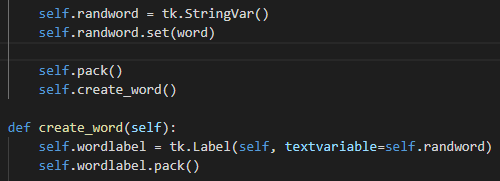
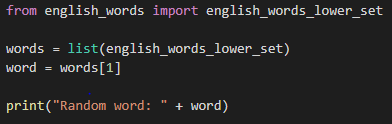
**Test Case: 1**

**Description:** Generates a random English word to the Tkinter window.

**Test Steps:**

1. Generates a random word
2. Check if the generated word is a correct English word in lowercase
3. Word is displayed in Tkinter window

**Expected Result:** The word will be printed in console and displayed to Tkinter window.



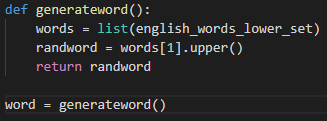
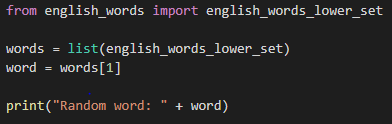
Figures 1. and 2. In the code images above, it shows that the random word is being generated from a python library and I am selecting 1 of these words. I am then printing the word to the console and also creating a wordlabel to hold the random word.. Figure 3. Shows the random word generated in the Tkinter window and in console.

**Actual Result:** The word is displayed in lowercase in the Tkinter window and printed to the console.

**Pass / Fail:** Pass – While the code passed on the first attempt, there is still refactoring to be done.

**Refactoring Test Case 1**

**Refactor Reason:** The random generation of the word should be within a defined function so that it is encapsulated and may be expanded on later, also I realized it would be better if the word were uppercase.



Before / After: We can see the code is now within a defined function which returns the word when called.

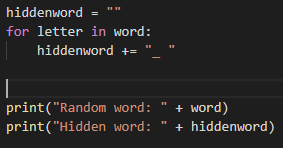
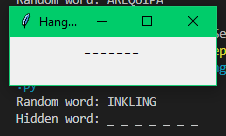
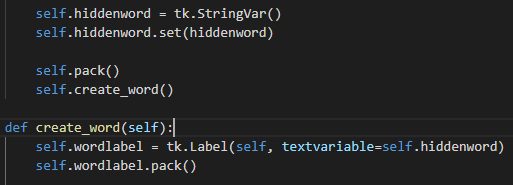
**Test Case: 2**

**Description:** Make letters display hidden and show only Underlines on screen.

**Test Steps:**

1. Make the word hidden
2. Display an underscore for each letter in the word
3. Display the hidden word to the screen

**Expected Result:** The word shall be hidden and displayed on screen as underscores only.

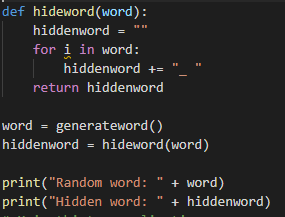
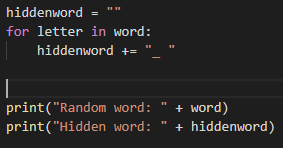
The code figures above show how the hidden word is generated by looping through the random word and placing underscores and spaces.

**Actual Result:** The word is displayed as underscores to the Tkinter window.

**Pay / Fail:** Pass – Although this passes for now, in the future it will need to be majorly refactored

**Refactoring Test Case 2**

**Refactor Reason:** The hiddenword code for this test case should also be encapsulated.



**Before / After:** The hiddenword code is now encapsulated, allowing for ease of change in the future.

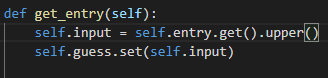
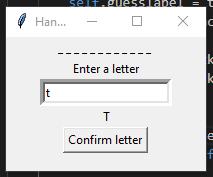
**Test Case: 3**

**Description:** User can enter a guess letter. This letter will be displayed to the screen to show their entry.

**Test Steps:**

1. User types in their letter
2. User presses confirm
3. The entered letter displays on the screen in uppercase

**Expected Result:** After the user confirms their letter entry, it will be displayed to the screen as uppercase.

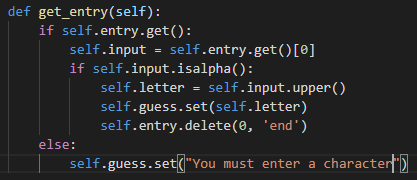
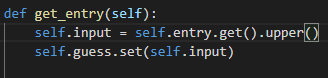
Figures 1. Shows the code for function get\_entry. When the confirm button is pressed, the code grabs the input from the user from the entry box (text box) and converts it to upper then sets the text of guess to equal the input. Figure 2. Shows the entered letter T on screen.

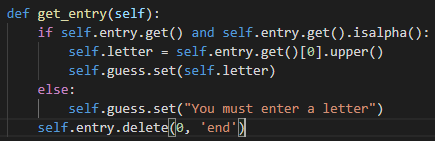
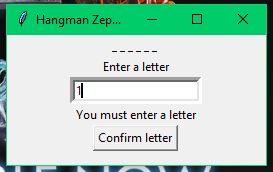
**Actual Result:** The entered letter is shown to the screen as uppercase.

**Pass / Fail:** Pass – While the code passed on the first attempt, there is still refactoring to be done.

**Refactoring Test Case 3**

**Refactor Reason:** Currently the user’s full input is taken rather than 1 letter and it does not check if the entry exists or if it is a letter. Also, for better usability I want to remove the previous text from the box on confirmation and show error checking for wrong input.

First refactoring

  Second refactoring

**Before / After:** The code has been refactored twice, the code after refactor 1 had an additional if condition so I combined them, now refactor 2 includes all the features mentioned above.

**Test Case: 4**

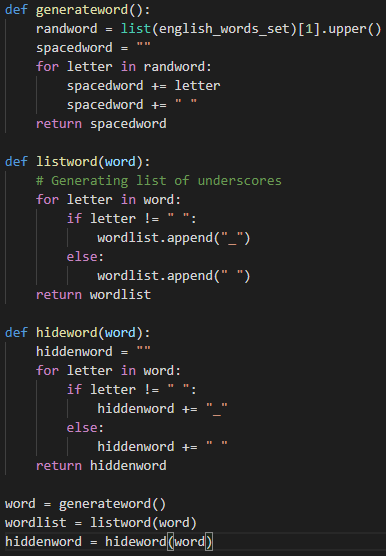
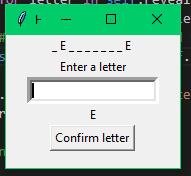
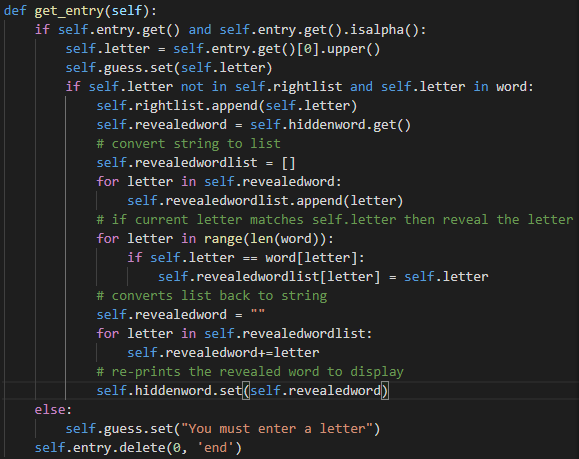
**Description:** If user’s guessed letter is in the word it will reveal all instances of that letter in the word.

**Test Steps:**

1. User types in their letter and confirms
2. Check if hidden word is in word
3. If letter in hidden word update display with revealed letters
4. Otherwise do nothing, wrong guess will be handled in next test case

**Expected Result:** After the user confirms their letter entry, if letter in hidden word, the display will be updated with the revealed letters.

**\*\*In order to complete this test case I had to refactor how the hidden word works – See** [**code smell 1**](#codesmell1)**.**

 **- Code refactored in code smell 1 now allows the following code to work.**

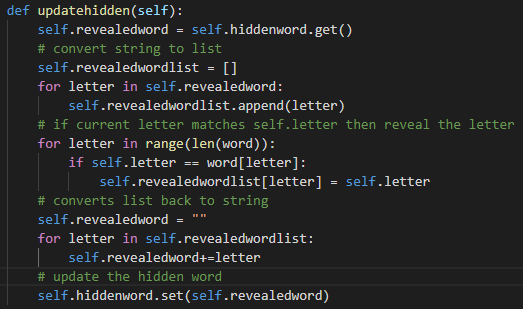
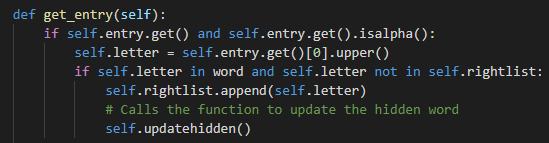
Figures 1. Shows the newly developed code that takes the user’s input and if the guessed letter is in the word and not in a list of correct letters then it performs the following: Appends letter to rightlist, converts the displayed word from string to list, a for loop that adds the current letter to the list for each position it appears, then converts the list back to string then lastly updates the hidden word with the new revealed letters. Figure 2. Shows that by entering E the hidden word has been reveled with all occurrences of E.

**Actual Result:** The program reveals all occurrences of correct letters in the hidden word.

**Pass / Fail:** Pass

**Refactoring Test Case 4**

**Refactor Reason:** The function for updating the hidden word was cluttering the get\_entry function so I extracted the code into a new function.



**Before / After:** (“Before” image is the code in the red box above) The code has now been extracted into the function thus making the get\_entry much cleaner.

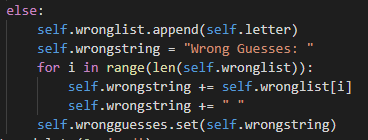
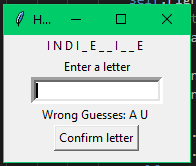
**Test Case: 5**

**Description:** Displays wrong guesses to the screen.

**Test Steps:**

1. User enters a letter
2. Check if letter is in word and is already guessed,
3. If not in word and not guessed, then add it to the wrong guesses list
4. display wrong guesses list on screen

**Expected Result:** The wrong guesses are displayed to the screen.

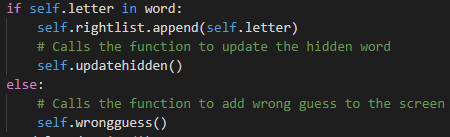
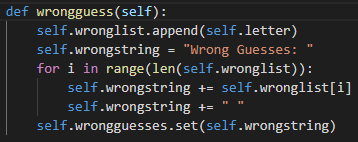
Figures 1. and 2. Show how the wrong guesses are being added to a list which is displayed to the screen.

**Actual Result:** The wrong guesses are displayed to the screen.

**Pass / Fail:** Pass

**Refactoring Test Case 5**

**Refactor Reason:** The function for displaying wrong guesses was also cluttering the get\_entry function so I extracted the code into a new function.

Before / After: We can see the code is now within a defined function which displays wrong guesses.

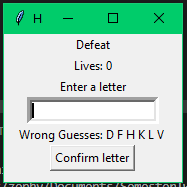
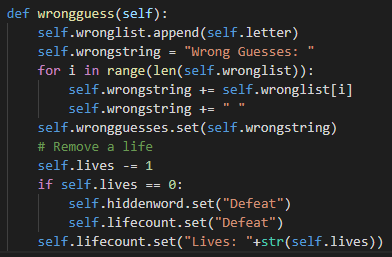
**Test Case: 6**

**Description:** Windows shows life count and lives reduced by 1 for each wrong guess.

**Test Steps:**

1. User enters a letter
2. Check if letter is in word and is already guessed
3. If not, then reduces the total life by 1
4. If zero is reached, replace with defeat text

**Expected Result:** The lives will be reduced by 1 per wrong letter entered and printing life count to the screen. When lives reach zero, the game shows defeat.

Figures 1. and 2. Shows the code for setting and removing lives. Figure 3. Shows the game reaching a defeat state when lives reach zero.

**Actual Result:** When the player enters the wrong letter, the lives reduce by one and when they reach 0 the window displays defeat.

**Pass / Fail:** Pass – No refactoring was required as I placed the code inside of the already defined wrongguess function.

**Test Case: 7**

**Description:** Displays victory when the user completely reveals the word.

**Test Steps:**

1. User enters a letter
2. Check if letter is in word
3. If yes, then reveals the word on screen
4. When the word is completely revealed it congratulates the player for winning.

**Expected Result:** When the word is completely revealed the player wins.

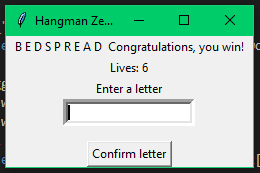


Figure 1. shows that when there are no blank spaces, the player wins. The second figure shows how it displays on the screen.

**Actual Result:** When the word is completely revealed, the player wins.

**Pass / Fail:** Pass – No refactoring was required as I placed the code inside of the already defined updatehidden function.

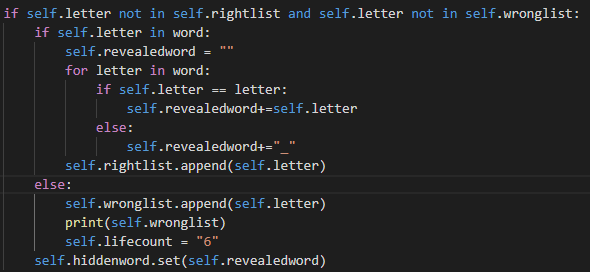
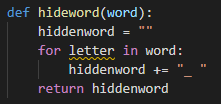
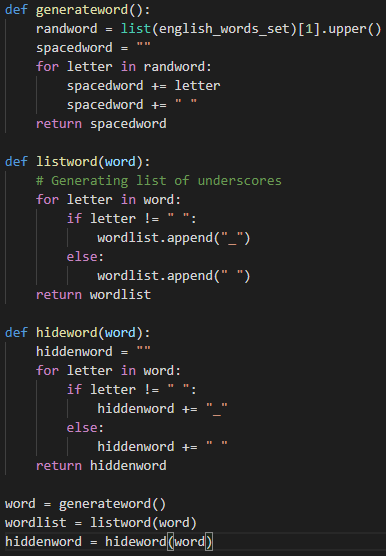
# Refactoring

## Code Smells

A code smell is a front-end / surface indication that usually corresponds to a deeper problem in the system. This section will clearly detail found code smells and the implementation of their fix.

**Code Smell 1**

**Code Smell reason:** Hiddenword is a string and is immutable in python, so when it comes to revealing the hiddenword string, the letters cannot be altered. Therefore, hiddenword must become a list but tkinter can only display string variables, therefore we must convert the string to a list to allow for modifying, then back to a string to display it on Tkinter.

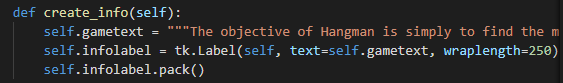
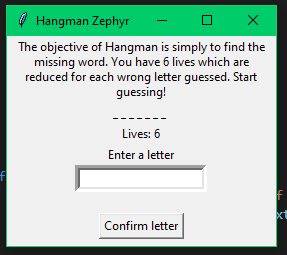
 Attempted solution

Firgure 1. Shows the code before implementing the string to list idea. Figure 2. Shows the attempted solution at trying to update the hidden word with occurrences of the letter but following this method I was never able to successfully update the word. Therefore, I implemented the idea of converting to list then back to string. Figure 3. Shows the updated code that now allows the

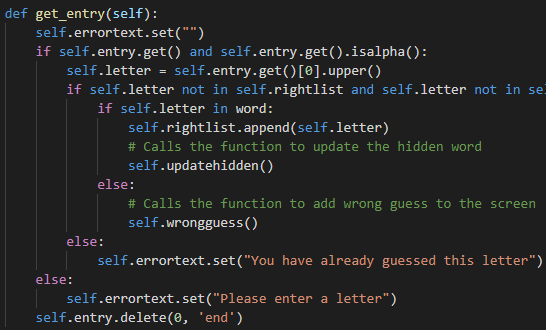
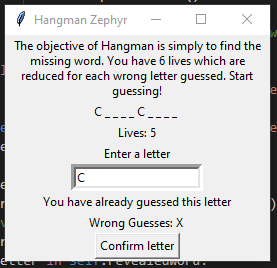
**Refactoring Before / After:** The refactor now allows for the random word to be generated and spaced. A list of the word is generated then lastly, the hiddenword is created the same way as the list. With these three in place, the program can now modify the list which will then be converted to the hidden word.

**Code Smell 2**

**Code Smell reason:** Error checking and usability go hand in hand in software creation therefore, I must implement advanced error handling and user feedback for cases that require it.

The figures above show the inclusion of game instructions.

The figures above show intrdouced error checking and user feedback. E.g. if they enter a previously entered character.