
# SOFE 3980U
# Software Quality
# Winter 2022

## Assignment-1: Choosing the right software process to build good-quality software, and ensuring quality using test automation (12.5 points)

Owais Quadri
100697281

# Software Quality Assignment 1

## Program

The program that I developed was a spinoff of the new viral internet game called Wordle. The name of my game is “Wordify”. The user must use 6 tries to guess a word that contains 6 letters

## Development Process

The development process that was used was the incremental process. This entails the initial stage of planning, where I defined the requirements for the software as well as designed the outline. Then, the implementation process is where the development happens and then the development progress is tested for functionality and any failures. if there are any missing functionalities, then the process loops back to the planning process or else the program can go to deployment.

### Planning

The planning process began by identifying the game that I would like to recreate. I decided for Wordle because I wanted to understand the complexity/simplicity of the program that got so popular for a simple idea. I extended the functionality by making the word length of guesses to be six letters instead of five. I also needed a word list to pick from so I used a list of 450 words and the word was randomized on each attempt instead of only selecting a certain word for a whole day. This encourages the user to make more than one attempt in a day, and enhances the replayability of the game.

### Requirements

List of functional requirements:

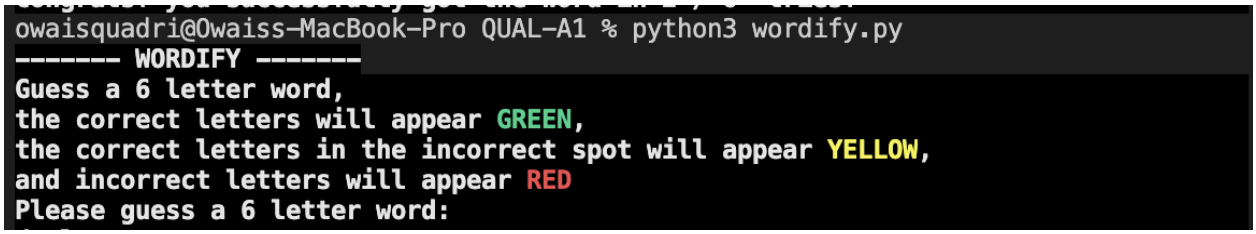
- R1: The user must be able to guess any six letter word but only a six letter word.
- R2: If the user guesses the correct word, a congratulations message will print and the program will exit.
- R3: The letters that are correctly guessed are made known to the user.
- R4: The letters that are in the incorrect position are made known to the user.
- R5: The letters that are not in the word are made known to the user.
- R6: The program will exit as an unsuccessful attempt when the user incorrectly guesses the word 6 times.

List of non-functional requirements:

- NR1: The status of the guessed letters have a visual cue (red for incorrect, green for correct, and yellow for incorrect position).
- NR2: On exit after an unsuccessful round, the program will reveal the correct word.

## Implementation

I implemented the game in python by using the command line as an interface. The user is informed of the rules of the game when the program starts:




This was the first  round of implementation, and basic completion of the user interface. Then, the next part of planning consisted of collecting a random word to be used for the correct word. This was implemented by using the 'random' library and selecting a random word from the list of words in 'wordlist.txt'. Once this was implemented, it was tested by printing the word on each run to make sure the words are different and randomized each time.

The next implementation is the implementation of the guess verification. First verification is whether the guess contains six letters. If the guess doesn't contain six letters, the attempt is invalidated and the user is able to guess again without spending a guess. Then, we iterate through the guessed word to check on the status of each letter. If the letter is correct, the letter is rendered green in the output. If the letter is in the incorrect position, the letter is rendered yellow. If the letter is incorrect, the letter is rendered as red.

In the end, if there is a correct guess within 6 guesses, a congratulations message is displayed or else the program exits and reveals the correct answer.

# Testing

Since some of the application returns void actions, manual and automated testing methods were implemented.

## Manual Testing

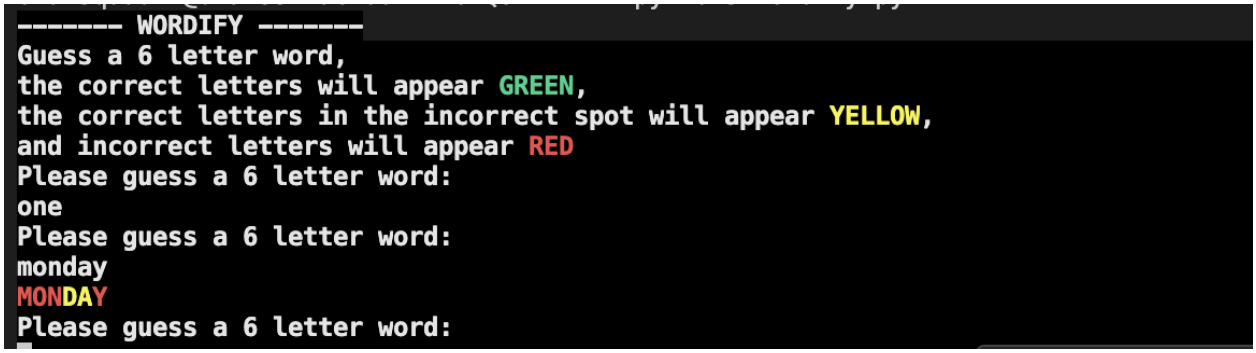
The manual testing was done by running the file to check for functionalities.

Test Cases (screenshots will follow):

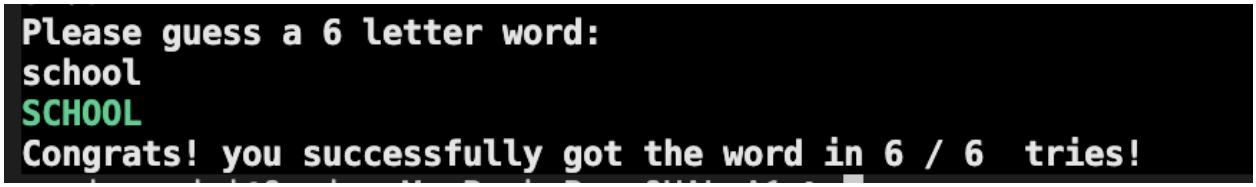
| Test Case | Description | Corresponding Requirements | Result |
|---|---|---|---|
| T1 | Ensure only 6 letters can be entered in each guess. | R1 | PASS |
| T2 | Ensure a congratulations message is displayed after successful completion. | R2 | PASS |
| T3 | ensure that the user knows of the correctly guessed letters | R3 | PASS |
| T4 | ensure that the user knows of the correctly guessed letters in the wrong position | R4 | PASS |
| T5 | ensure that the user knows of the incorrectly guessed letters | R5 | PASS |
| T6 | Ensure that the program will exit as an unsuccessful attempt when the user incorrectly guesses the word 6 times. | R6 | PASS |
| T7 | Verify that the status of the guessed letters have a visual cue | NR1 | PASS |
| T8 | ensure the program will reveal the correct word after an incorrect attempt. | NR2 | PASS |

Screenshots:

T1:

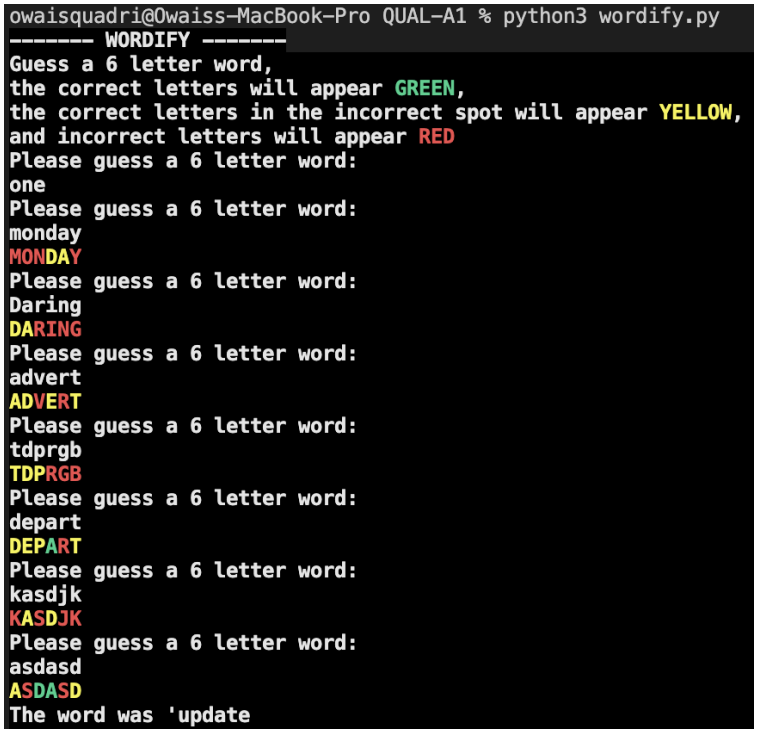



T2:




T3, T4, T5, T7:


T6, T8:



## Automated Tests

I integrated automated tests into my code using a ‘tests.py’ file which was inside a ‘test’ folder. The tests automatically run as each unit is executed and will exit with errors if it has any errors or if any of the assertEqual statements are violated. There are five automated test cases that were carried out.

Screenshots and Explanations:

```
# test 1: making sure that a game exists
def test_init(self, game=None):
    # init a wordify
    game = Wordify()
    # real, desired, failmessage
    self.assertEqual((game != None), True, "The game does not exist")
```

The first test case was determining whether there is an instance of the game that is available. First, I did not include the ‘game=Wordify()’ so that this would surely fail. Then I added the line of code to complete the test case.

```
15      # test 2: rig the game
16      #         (make everything lowercase)
17      def test_rig(self):
18          # start a game
19          game = Wordify()
20          #rig to any word
21          game.rigged_to("WoRdLe")
22          self.assertEqual((game.word), "wordle", "Cannot manually set word")
```

The second test is to ‘rig’ the game so that the testing can continue without the issue of not knowing the correct word. I also had to ensure that the input was always converted to the same case of letters (lower-case). I did this for the program’s ease so that the comparisons could ignore the case of each letter.

```
23      # test 3: take the correct guess
24      def test_guess(self):
25          # start a game
26          game = Wordify()
27          #rig
28          game.rigged_to("wordle")
29          #guess a valid word
30          self.assertEqual(game.guess(0,"wordle"),"Congrats! you successfully got the word in 1/6 tries!","Did not display congrats message")
```

The third test is regarding the correct answer. The application should send the user a congratulations message and exit the program right afterwards. The reason that I could not test any of the middle cases (1st guess moving to the next etc) is because these do not have return statements until the recursion loop is exited at a correct guess or at the last guess (6 incorrect guesses).

```
31      # test 4 : make sure the colours are correctly corresponding to the character status'
32      def test_render(self):
33          # start a game
34          game = Wordify()
35          #rig
36          game.rigged_to("wordle")
37          #render a guess
38          print(game.render("wallet"))
39          self.assertEqual(game.render('wallet')[1],"greenredyellowyellowyellowred","incorrect colors")
```

This test is focused on the rendering of the correct colour of a certain word that is guessed. When an incorrect word is rendered as guessed, the program will colour each letter based on its status. The possible statuses are: green for the correct letter in the correct position, yellow for wrong position of correct letter and red for a letter that is not in the word entirely. Furthermore, I had an issue with the testing of this unit due to the escape characters used in colouring letters in the command line, and therefore I had to come up with different ways to show the order of colours in a render.

```
23      def render(self, wordify):
24          # portray words wrt correct word
25          color_order=""
26          output = ""
27          for i in range(WORD_LEN):
28              if wordify[i] == self.word[i]:
29                  # correct
30                  output += colour.GREEN+wordify[i].upper()+colour.DEFAULT
31                  color_order+="green"
32              elif wordify[i] in self.word:
33                  # incorrect spot
34                  output += colour.YELLOW+wordify[i].upper()+colour.DEFAULT
35                  color_order+="yellow"
36              else:
37                  # incorrect letter
38                  output += colour.RED+wordify[i].upper()+colour.DEFAULT
39                  color_order+="red"
40          return (output,color_order)
41
```

As shown in the code pictured above, the render function outputs a tuple that includes the output that is to be printed, as well as the order of the colours in the output. This is purely a feature added for testing and it will always be consistent with the other value in the tuple because the two variables are always modified in the same locations, meaning that they cannot be accessed in other places.

```
40      #test 5: testing whether a loss will reveal the true word
41      def test_loss(self):
42          # start a game
43          game = Wordify()
44          #rig
45          game.rigged_to("wordle")
46          #incorrect last guess
47          self.assertEqual(game.guess(5,"walter"),"The word was: wordle", "Did not mention the correct word on the way out")
```

The final test requires the automated tests to suffer a loss and confirm that the game will reveal the correct word at the end of an unsuccessful game.

The output of this testing file is as follows:

```
C:\Users\owais\Documents\QUAL_A1>python test/tests.py
WORDLE
..WALTER
.('\x1b[1;32;40mW\x1b[1;37;40m\x1b[1;31;40mA\x1b[1;37;40m\x1b[1;33;40mL\x1b[1;37;40m\x1b[1;33;40mL\x1b[1;37;40m\x1b[1;33;40mE\x1b[1;37;40m\x1b[1;31;40mT\x1b[1;37;40m', 'greenredyellowyellowyellowred')
..
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
```

# Challenges

One challenge that I faced during development was when I was moving from my desktop windows PC to a Mac. I ran into an issue where the runtime could not interpret inputs from the user. I realised that I needed to use the python3 command instead of just using the python command from windows.

```
owaisquadri@Owaiss-MacBook-Pro QUAL-A1 % python wordify.py
------- WORDIFY -------
Guess a 6 letter word,
the correct letters will appear as 'C',
the correct letters in the incorrect spot will appear as '/', and incorrect letters will appear as '_'
Please guess a 6 letter word: murder
Traceback (most recent call last):
  File "wordify.py", line 63, in <module>
    start()
  File "wordify.py", line 59, in start
    guess(0)
  File "wordify.py", line 41, in guess
    guessed_word = input("Please guess a 6 letter word: ").lower()
  File "<string>", line 1, in <module>
NameError: name 'murder' is not defined
```

# Conclusion

In summary, this game parody was developed using the incremental process, and was tested using a variety of methods. The manual testing was thorough where the automated testing could not be used, and the automated testing was well integrated into the package, yet detached from the main application enough to test each unit separately. Therefore, this high-quality software is thoroughly tested for each scenario as described in the previous sections, and follows a development process that is proven to successfully produce quality software.