CSC2034 Project Report

Science Of Programming

Name: Nitin Jacob
Student number: 220448307

Abstract

This report details the development and enhancement of a Python program implementing a simplified version of the popular word guessing game, Wordle. The tasks involved the creation of a new operation, *hint(),* which provides users with a single character hint to help in guessing the word. A new type called "Hint" was created to represent single-character hints. Subsequently, the *check_guess()* function was improved to correctly handle repeated letters in the word, utilising a counter object to track occurrences. To give users the full Wordle experience, the function was also updated to produce visual output using ANSI colour codes. Additionally, a "hard mode" feature, represented by the *hard_guess()* function, was introduced, enforcing constraints on subsequent guesses based on revealed hints from prior attempts. The report highlights the success of these implementations, ensuring adherence to pre-conditions and thorough testing.

In conclusion, the Wordle project served as an enriching learning experience, fostering skill development in software development, problem-solving, and effective time management. The project's successful completion highlights its contribution to the developer's growth and proficiency, indicating that it was a rewarding journey in software development.

Word Count: 1,989

What Was Done And How

Firstly, I was tasked with creating a new operation *hint()* that gives users an appropriate hint by returning a single character to help them solve the wordle. I did this by creating a function that takes a word and a list of revealed hints as inputs and returns the next available letter that has not already been revealed. I created a new type, '*Hint'*, which has been defined using the *'NewType'* function from the '*typing*' module. This type is used to represent single-character hints. The function starts by checking whether a letter is already in the list of revealed hints for every letter in the word. If not, the letter is displayed after being added to a list. The pre-conditions for this function are to ensure the word parameter should be a string representing the target word, consisting only of uppercase letters [A-Z] and the revealed hints parameter should be a list of hints, where each hint is a single uppercase letter [A-Z]. In cases where we assume all letters in the word are already revealed, the function does not return any hints. Finally, after completing the function, I tested it with valid and invalid inputs and verified whether the correct hint was displaying.

```python
from typing import NewType

# Define a type variable for the Hint type
Hint = NewType("Hint", str)

def hint(word: Word, revealed_hints: List[Hint]) -> Hint:
    """
    Returns a single character hint to help the player.
    """

    # pre-condition
    assert isinstance(word, str) and all(letter.isupper() and letter.isalpha() for letter in word), \
        "pre-condition failed: Word should be a string representing the target word, consisting only of uppercase letters [A-Z]."
    assert all(isinstance(hint, str) and len(hint) == 1 and hint.isupper() and hint.isalpha() for hint in revealed_hints), \
        "pre-condition failed: Revealed hints should be single-character strings, consisting only of uppercase letters [A-Z]."

    selected_hint = []

    for letter in word:
        if letter not in revealed_hints:
            selected_hint.append(letter)


    if selected_hint:
        revealed_hints.append(selected_hint[0])
        output = "Hint: " + selected_hint[0]
        return output # returns the next available letter that is not already a revealed hint
    else:
        return "No hints"
```

(Figure 1. hint() function)

My next task was to fix the implementation of *check_guess* combined with *check_letter* to correctly produce the colours of the letters when there are repeating letters in the word. This was successfully created by changing the operation body of the *check_guess* function to directly compute the clues without depending on the *check_letter* function. First, I initialised all letters in the word to grey by default and assigned a 'Counter' object from the 'collections' module to a variable called counter to count the occurrences of elements in a collection. Next, I created two for loops to search for the letters in the word. In the first loop, it checks if the guessed letter at position 'i' in the guess matches the letter at the same position in the word. If it does, it increments the count of that letter in the counter, and it sets the corresponding clue in the clues list to Clue.GREEN. This means the letter is correctly placed in the word. For the other for loop, it checks if the clue for the current letter is already marked as 'Clue.GREEN', and if the letter is placed correctly it moves to the next iteration. If the letter is not correctly positioned, it checks if the letter is present in the word and checks if the

number of correctly guessed occurrences of this letter is less than the total occurrences of this letter in the word. If so, the letter should be marked to Clue.YELLOW. To colour code the letters in the word like how it is in the Wordle game, I created a dictionary that assigns ANSI colour codes to each possible clue status (Clue.GREEN, Clue.YELLOW, and Clue.GREY) and then created a list at the end, where each letter is presented in the defined colour associated with its clue status. The resulting coloured word is then printed to the console *(Print Colors in Python terminal, 2022).*

```
def check_guess(word: Word, guess: Word) -> List[Clue]:
    """
    Given the answer and a guess, compute the list of
    clues corresponding to each letter.
    """

    # ANSI color codes
    color_codes = {
        Clue.GREEN: '\033[92m',   # Green
        Clue.YELLOW: '\033[93m',  # Yellow
        Clue.GREY: '\033[90m',    # Grey
        'RESET': '\033[0m'        # Reset
    }

    # pre-condition
    assert len(word) == WORD_LENGTH and \
           len(guess) == WORD_LENGTH, "pre-check_guess failed"

    clues = [Clue.GREY] * len(word) # initializes all letters in the word to grey
    counter = Counter()

    # finding the if right letters are in their correct places
    for i, letter in enumerate(guess):
        if letter == word[i]:
            counter[letter] += 1
            clues[i] = Clue.GREEN

    # checking for existing letters out of correct places, after searching for green letters
    for i, letter in enumerate(guess):
        if clues[i] == Clue.GREEN:
            continue
        elif letter in word and counter[letter] < word.count(letter):
            counter[letter] += 1
            clues[i] = Clue.YELLOW

    # print colored output
    colored_output = [f'{color_codes[clue]}{letter}{color_codes["RESET"]}' for letter, clue in zip(guess, clues)]
    print("".join(colored_output))


    return clues
```

(Figure 2. check_guess() function)

For the last task, I had to implement a “hard mode” function into the wordle game which follows the rule that any revealed hints must be used in subsequent guesses further constraining what a user can guess with. I had first copied and pasted the game class onto a new code cell and added a new function to it called *hard_guess*. This function was taken from the already defined function, *make_guess*. The only difference is the pre-condition, that checks each position of the previous guesses and asserts that the current guessed letter matches the answer at the same position (green letter) or that the current guessed letter is present in the answer (yellow letter). If the pre-condition fails, the program will raise an assertion error saying that the guess was incorrect as it did not contain a revealed letter from the previous guesses. This was tested in the *test_check_letter* function where I included a commented-out test case which, when uncommented, attempts to make a guess with

letters that are not in the revealed hints list, and so, displays an assertion error because it violates the constraints of the “Hard mode” functionality.

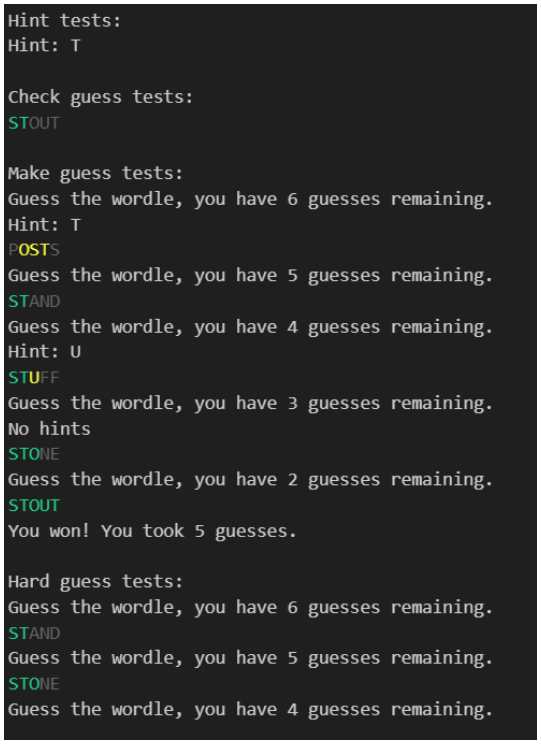
```
def hard_guess(self, word: Word):
    """
    Make a guess at the wordle in Hard mode.
    """
    # make guesses uppercase
    word = word.upper()

    # pre-condition
    assert self.gstate == Gamestate.PLAYING and \
        len(self.guesses) < MAX_GUESSES and \
        word in WORDS, "pre-hard_guess failed."

    # Check if there are previous guesses to constrain the current guess
    if self.guesses:
        prev_clues = self.guesses[-1].clues
        for i, clue in enumerate(prev_clues):
            if clue == Clue.GREEN:
                assert word[i] == self.answer[i], "pre-hard_guess failed: Incorrect guess based on previous GREEN clue."
            elif clue == Clue.YELLOW:
                assert word[i] in self.answer, "pre-hard_guess failed: Incorrect guess based on previous YELLOW clue."

    self.guesses.append(Guess(word, check_guess(self.answer, word)))
    if word == self.answer: self.gstate = Gamestate.WON
    elif len(self.guesses) == MAX_GUESSES: self.gstate = Gamestate.LOST
```

(Figure 3. make_guess() function)

To make sure the test cases run repeatedly, I wanted to combine them all into a single code cell at the end of the notebook and separate them with comments. In it, I created a test function for each operation in the program which accurately tests the game’s functionality.

```
# Test the hint function
print("Hint tests:")
def hint_test():
    Word = "STOUT"
    revealedLetters = ["S", "O"]
    print(hint(Word, revealedLetters))
hint_test()

# Test the check_guess operation
print("\nCheck guess tests:")
def test_check_guess():
    check_guess("STAND", "STOUT")
    pass
test_check_guess()

# Test the make_guess operation
print("\nMake guess tests: ")
def test_make_guess():
    Word = "STOUT"
    revealedLetters = ["S", "O"]

    game1 = Game()
    game1.print_state()
    print(hint(Word, revealedLetters))
    game1.make_guess("posts")
    game1.print_state()
    game1.make_guess("stand")
    game1.print_state()
    print(hint(Word, revealedLetters))
    game1.make_guess("stuff")
    game1.print_state()
    print(hint(Word, revealedLetters))
    game1.make_guess("stone")
    game1.print_state()
    game1.make_guess("stout")
    game1.print_state()
test_make_guess()

# Test the hard_guess operation
print("\nHard guess tests: ")
def test_check_letter():
    game = Game()
    game.print_state()
    game.hard_guess("stand")
    game.print_state()
    game.hard_guess("stone")
    game.print_state()
    # game.hard_guess("stink") # This should raise an AssertionError since "stink" doesn't follow the constraints
test_check_letter()
```

(Figure 4. Test cases for each of the operations)



(Figure 5. Output of the program)

Results and Evaluation

The final artifact is a Python program that implements a simplified version of the popular word guessing game, Wordle. The code consists of several functions allowing users to test the game and make guesses. The important functions of this artifact are:

1. check_letter():
    - Description: Given a clue (letter), an index (position of letter) and the answer, this function computes the colour of the clue based on the placement of the letter in the answer. The colour green indicates that the letter is in the correct position; yellow indicates that the letter is in the answer word but not in the correct position; and grey indicates that the letter is not in the word.

2. check_guess():
    - Description: Given the answer word and the guessed word, this function computes a list of clues for each letter. In addition, the output text is printed using colour codes that correspond to the word's letter positions.
    - Evaluation: The *check_guess()* function has been updated to handle the repeated letters and the modifications of *check_guess* do not impact the overall correctness of the function, as it still accurately computes the clues without *check_letter*. The inclusion of ANSI colour codes serves as a form of visualisation for the task. They use different colours (green, yellow, and grey), which match the typical colour scheme used in Wordle, to add a visual element and make it appear more engaging.

3. hint():
    - Description: Given the answer word and a list of revealed hints, this function computes a single character hint from the answer, that has not already been unveiled, to help the player guess the word. The function returns the next unrevealed letter as a single-character hint if there are still unrevealed letters. It returns "No hints" to indicate that there are no more hints available if every letter has already been revealed.
    - Evaluation: The implementation of the *hint()* operation follows the task description by defining the 'Hint' type as a new type, and the invariants ensures that the hint is a single uppercase letter. The function returns the next available unrevealed letter as a hint.

4. make_guess():
    - Description: Given the guessed word, this function will make a normal guess at the Wordle. It first checks the validity based on the pre-conditions and computes the corresponding clues using *check_guess*. If the guessed word matches the target word, the game is won. If the maximum number of guesses reaches its limit without a correct answer, the game is lost.

5. hard_guess():
    - Description: Given the guessed word, this function will make a guess at the Wordle but will first check if there are revealed clues from the previous guess in the current guess. The function makes sure that the guess word follows the known correct positions and contains the correct letters by enforcing constraints based on the previously obtained clues. If the guessed word does not follow the rules of hard mode, the program raises an AssertionError.
    - Evaluation: the *hard_guess()* operation was implemented to meet the requirement of using revealed hints in subsequent guesses in hard mode. The pre-condition constrains the guesses the players can make. In the test cell, *test_check_letter()* checks the correctness of the *'hard_guess'* operation, ensuring that the constraints are met.

Conclusions

In summary, the provided code underwent several additions and enhancements to meet the specific tasks. The subsequent tasks included:

1. Defining a new operation *hint()* which provides users with a single character hint for their guesses. The function checks for unrevealed letters in the answer word and returns the next available letter that has not been revealed yet.

2. The *check_guess()* function was enhanced to handle situations with duplicate letters in the word. The clues are now calculated using a combination of loops and conditions to handle repeated letters correctly. To visually represent the clues in the output, ANSI colour codes were added to the function.

3. To replicate "Hard mode" in the Wordle game, a new function called *hard_guess()* was added to the "Game" class. The function imposes constraints on the guessed word based on

revealed hints from previous guesses. This was done by implementing pre-conditions that the current guess adheres to the rules of Hard mode.

Throughout these tasks, the code was iteratively tested to ensure correct functionality with the specified requirements.

Reflections and Future Work

My time spent on this project has become enjoyable because it helped me in acquiring an extensive knowledge on creating projects using Jupyter Notebook, as this was a platform that was introduced to me for the first time. I made sure to break down the specifications on the first day of working on this project in order to simplify my objectives by writing down the pseudocodes *(BBC, no date)* of the functions that were required (see Figure 6 below). Subsequently, I executed every function individually and tested them through trial and error. When updating the *check_guess()* function to correctly colour duplicate letters, I struggled to get it to work correctly. Sometimes, when searching for the duplicate letters in the guesses, it would incorrectly colour the entire guess. After many approaches, I fixed this by creating a variable called "counter" which initialises a *Counter()* object. I used this approach to keep track of how many letters have been correctly guessed including repeated letters. Upon successful completion of these tasks, I decided to add an extra extension to the project by visualising the console output with coloured texts. I did this because I wanted to ensure the project closely resembled the actual game. At first, I tried to include a pygame package in the artefact so that it would show the whole game in a different window. Unfortunately, because of the short submission time, I had trouble properly incorporating this feature into the finished product. The process of setting up and configuring the pygame package proved to be more intricate and time-consuming than anticipated, adding an element of frustration to the endeavour. Given the tight deadline, I had to prioritise essential functionalities, which made me decide not to include pygame in the project's final version. This experience made clear how crucial it is to manage your time properly and how important it is to maintain a balance between project goals and practical constraints.

Following the completion of the Wordle project, some suggestions for future endeavours aim to broaden my skill sets and facilitate compelling opportunities in the dynamic field of software development. This involves the implementation of comprehensive software solutions designed for web applications, demonstrating proficiency in both front-end and back-end development. To properly carry out my responsibilities, I plan to keep decomposing the current task into simpler parts and come up with solutions swiftly *(Lcom Team, 2022)*. Alternatively, I could specialise in game or mobile app development, explore AI and machine learning, allowing me to use algorithms and models to solve challenging problems that will ultimately improve user experience. Finally, in order to minimise time management complications, I will make sure that next time I prioritise improving all task requirements by creating a more detailed project plan.

Working on the Wordle game project provided me with the opportunity to implement and fix core game functionalities and incorporate test cases, which was an invaluable learning opportunity. Implementing the *hint()* method and improving the *check_guess()* function, for example, required the use of critical thinking and problem-solving strategies, which helped me improve my skills in handling challenging situations. Overall, this project has been a rewarding journey that has significantly contributed to my growth as a developer and problem solver.

```
# HINT PSEUDOCODE
hint(word: Word, revealed_hints: List[Hint]) -> Hint:
    # Preconditions
    Verify word is a string containing only uppercase letters [A-Z]
    Verify revealed_hints is a list of single-character strings containing only uppercase letters [A-Z]

    Initialize selected_hint as an empty list

    For each letter in word:
        If the letter is not in revealed_hints:
            Add the letter to selected_hint

    If selected_hint is not empty:
        Add the first letter from selected_hint to revealed_hints
        Return the first letter from selected_hinttput
    Else:
        Return "No hints"



# CHECK GUESS PSEUDOCODE
check_guess(word: Word, guess: Word) -> List[Clue]:
    # Preconditions
    Verify length of word is equal to WORD_LENGTH
    Verify length of guess is equal to WORD_LENGTH

    Initialize clues as a list of length equal to the word, with all elements set to Clue.GREY
    Initialize a counter to keep track of the count of correct letters in correct positions

    # Finding if right letters are in their correct places
    For each index i and letter in guess:
        If the letter at index i in guess matches the letter in the same position in word:
            Increment the count of that letter in the counter
            Set the clue at index i to Clue.GREEN

    # Checking for existing letters out of correct places
    For each index i and letter in guess:
        If the clue at index i is Clue.GREEN:
            Continue to the next iteration
        Else if the letter is in the word and the count of that letter in the counter is less than the count of that letter in the word:
            Increment the count of that letter in the counter
            Set the clue at index i to Clue.YELLOW

    Return the list of clues



# HARD GUESS PSEUDOCODE
hard_guess(word: Word):
    # Make the guess uppercase
    Convert word to uppercase

    # Preconditions
    Verify game state is PLAYING
    Verify the number of guesses is less than MAX_GUESSES
    Verify word is in the list of possible words WORDS

    # Check if there are previous guesses to constrain the current guess
    If there are previous guesses:
        Get the clues from the last guess
        For each index i and clue in the previous clues:
            If the clue is GREEN:
                Verify the letter at index i in the current guess matches the letter at index i in the answer. If not, raise an error
            Else if the clue is YELLOW:
                Verify the letter at index i in the current guess is in the answer. If not, raise an error

    # Add the current guess to the list of guesses and check the game state
    Append a new Guess object with the word and clues to the list of guesses
    If the word matches the answer, set the game state to WON
    Else if the number of guesses is equal to MAX_GUESSES, set the game state to LOST
```

(Figure 6. Pseudocode of the operations)

References

[1] Print Colors in Python terminal (2022) Available at: https://www.geeksforgeeks.org/print-colors-python-terminal/ (Accessed: 5th March 2024).

[2] Lcom Team (2022) What does Decomposition Mean in Computational Thinking? - Decomposition in Computational Thinking: Solving Problems More Effectively. Available at: https://www.learning.com/blog/decomposition-in-computational-thinking/ (Accessed: 9th March 2024).

[3] BBC (no date) Designing algorithms with pseudocode. Available at: https://www.bbc.co.uk/bitesize/guides/znv3rwx/revision/1 (Accessed: 9th March 2024).