Word Count: 2056

Science of Programming Wordle

CSC2034 Artefact Report

Abstract

We were asked by New York Times to complete the Python model of Wordle for them to understand how it works and to allow for extending functionality. Starting with a basic, incomplete model, we made a variety of improvements and even implemented a GUI using PyGame to replicate the current iteration of Wordle that is available on the New York Times website. Overall, we were quite successful: the basic game logic has been correctly applied and the game runs as intended. We also added the popular "hard mode" rules that the community likes to implement themselves into a button that can automatically enforce it for the player, along with the option to provide a hint for those who tend to struggle more with the basic rules, so that all differing skill levels can enjoy the game.

Sakir Azimkar

200823588

We started off with a simple tutorial, where we had to finish writing the logic for some half-written methods given to us, which finished the basic gameplay for wordle in a text-based format. This was quite simple as most of the difficult parts and the main setup was already completed. Although our solution was functional and relatively clean, we opted to proceed with the sample solution provided afterwards. As a result, we had more time to test the newer features, rather than testing our own version of the wordle logic in addition to anything added later.

Once the basic gameplay mechanics were completed, we had to think about how we wanted to implement a hint. It was a difficult decision: we previously wanted to go for a system that checked if the user had any yellow guesses and told them the index of one of those if there were, or to give the existence of a letter if they did not have a yellow clue at the time. This turned out to be quite convoluted and difficult to provide an explanation of (it was difficult to explain why the hint sometimes gave an index or a character without overwhelming the user with a lot of text). After looking for a solution online, we found an article about obtaining wordle hints without spoiling the entire word. Intrigued by this, the following line gave us our answer: "You could just stop there—if you know there's a G in it somewhere, seeing that headline might help you place it without taking an extra guess" (Skwarecki B., 2022). It is a fair assumption that other wordle players would agree that this is a good way to implement a hint, the confirmation of a single letter's presence. However, some players may not want any hints at all, and some may want multiple hints, based on their skill or enjoyment of challenge. As such, the decision was to provide the option for either a single hint or multiple, depending on how many times the player opts to receive a hint.

```python
import re

Hint = str

def hint(guesses: List[Guess], answer: Word) -> Hint:

    # pre-condition
    assert re.match('^[A-Z]{%d}$' % WORD_LENGTH, answer) and \
        answer in WORDS and \
        0 < len(guesses) < MAX_GUESSES, "pre-hint failed."

    # array of all letter indices
    grey = list(range(0, WORD_LENGTH))

    for j in range(0, WORD_LENGTH):
        if guesses[-1].clues[j] == Clue.GREEN:
            # remove indices with a green clue already
            grey.remove(j)
        elif guesses[-1].clues[j] == Clue.YELLOW:
            # removes indices with a yellow clue already
            index = answer.index(guesses[-1].word[j])
            grey.remove(index)
            answer = answer[:index] + '*' + answer[index + 1:]

    # set hint to a random currently grey letter
    hint = answer[random.choice(grey) if grey else '*']

    assert re.match('^[A-Z]$', hint), "all letters found"

    return hint
```

Figure 1: Hint Implementation Code

Our implementation was quite successful, the user is given a letter they didn't have in their previous guess. Figure One above shows the implementation. We first ensure that at least one guess has been made prior – and the maximum allowed guesses has not been exceeded – as well as the answer is of the correct length and is a valid word. To ensure that the hint provided is useful (not a green or yellow letter they have already figured out), we must remove these letters from the pool of possible hint options. We create an array of indices that are available called "grey" and iterate through the most recent guess. If there are green clues, the exact index is removed from the array as that is the correct location of the letter. We previously had the same logic for yellow clues, which was very quickly corrected for obvious reasons. Instead, the first instance of the letter is found, and its index is removed from grey. In the event of duplicate letters, we replace this index with an asterisk so that it is not found again. Then we check that a hint has been found; if the hint variable is set to an asterisk, it means that all the letters in the word have already been used. However, since this only uses the most recent guess, it may show the user a letter they already know if it were found in an earlier guess and not reused in the most recent. This is not a problem for hard mode, due to its rules, but can be a bit annoying for players not using this difficulty as they would have to press the hint button again.

There was a problem with the given implementation of the "check_guess" method: it did not function adequately for repeated letters. This was due to the "check_letter" function called by "check_guess" not being given enough information about the full word. We decided the best way to rectify this was by removing the second function entirely and doing the entire word check in "check_guess".

```python
def check_guess(word: Word, guess: Word) -> List[Clue]:
    """
    Given the answer and a guess, compute the list of
    clues corresponding to each letter.
    """
    # pre-condition, keeping same as before
    assert len(word) == WORD_LENGTH and \
           len(guess) == WORD_LENGTH and \
            guess in WORDS and word in WORDS, "pre-check_guess failed"

    # placeholders
    clues = [Clue.GREY for x in range(WORD_LENGTH)]


    # greens first
    for i, letter in enumerate(guess):
        if word[i] == letter:
            clues[i] = Clue.GREEN
            word = word[:i] + '*' + word[i + 1:]

    # yellows next, greys remaining are grey
    for i, letter in enumerate(guess):
        # either make the comparison of not green or replace it in guess as well, but a bit more work to do that so just this check
        if letter in word and clues[i] != Clue.GREEN:
            clues[i] = Clue.YELLOW
            # replace one instance of the letter in the answer, in case of duplicates
            word = word.replace(letter, '*', 1)


    return clues
```

Figure 2: check_guess Function

We kept the preconditions the same: the answer and guess must both be valid words, and they must be of the correct length. We then created placeholder values so that we didn't need to cycle through non-green and non-yellow values at the end. In the

event of duplicate letters, we needed to make sure that the correctly placed ones were highlighted green first, and that they did not cause the duplicate to be incorrectly labelled as yellow by seeing the correctly placed letter. To do this, we decided to obtain all the green clues first, replace them in the answer with an asterisk (so that they cannot be seen again), and then finish with the yellow clues. If there were correct duplicates, they would still be present in the word and labelled correctly. We then replace the first instance of the letter with an asterisk (in the event of the same letter appearing thrice), as the exact index does not matter for yellow clues. Our testing shows that this method is functional and handles duplicates as intended (see appendices below).

Now that the basic logic was working, we could continue with the addition of hard mode. This is already well-defined in the wordle community as "Any revealed hints must be used in subsequent guesses" (K. Pierce, 2024). In other words, if the player guesses a word and the result contains green letters, those letters must be in that exact index for their next guess, and yellow letters must also be included in the next guess also. We decided to use RegEx for the precondition at first, but then realised that it does not work for inclusion of a letter (pattern matching is very specific with placement). As a result, we used RegEx for green letters, and a similar method to the hint for the yellow letters. Once the precondition was satisfied and all letters that needed to be presented were accounted for, the rest of the method follows the same logic as a regular guess, so we just call “make_guess” afterwards. This method works as intended.

```python
def hard_guess(self, guess: Word):

    # edge case for first guess
    if len(self.guesses) == 0:
        self.make_guess(guess)
        return None


    recent_clues = self.guesses[-1] # obtain most recent guess info (all previous guesses will have incorporated the hints before)
    required = "*" * WORD_LENGTH
    included = []

    for i, letter in enumerate(recent_clues.word):
        if recent_clues.clues[i] == Clue.GREEN:
            required = required[:i] + letter + required[i + 1:]
        elif recent_clues.clues[i] == Clue.YELLOW:
            included.append(letter)

    # regex statement
    pattern_1 = re.compile("^%s$" % required.replace('*', "[A-Z]"))

    # check all yellow letters included
    for letter in guess:
        if letter in included:
            included.remove(letter)

    assert pattern_1.match(guess), "pre-hard_guess failed"
    assert not included

    self.make_guess(guess)

# Attach the hard_guess method to Game class
Game.hard_guess = hard_guess
```

Figure 3: Hard Guess Implementation

Once these additional features were fully implemented, we wanted to make the gameplay experience as similar to the real wordle as possible, with an GUI. Initially, we were going to use Tkinter, as it was a simple GUI tool that could do everything that we

needed for a game with such a simple UI. However, during research we found a video (Harsit, 2022) that used PyGame, a library designed for creation of games in Python. Considering this was a new module we hadn't used before, the idea was exciting, and we got to work straight away.

In the basic implementation, the number of maximum allowed guesses and the length of the words are defined as constants at the top. Although wordle is usually words of length five and have a maximum of six guesses, we wondered what would happen if someone wanted to change these values. As such, we implemented the GUI to adapt to these constant values, assuming they are within reason and the person changing these values provides words of valid length themselves. As a result, a lot of the drawing calculations use these constants, making the code seem quite daunting. For example, see below the calculation for the size of the boxes that the letters are written in.

```
BOX_SIZE = ((WIDTH - SPACE - WORD_LENGTH * OUTLINE_WIDTH * 2)//WORD_LENGTH,
            (HEIGHT - SPACE - MAX_GUESSES * OUTLINE_WIDTH * 2)//MAX_GUESSES)
```

Figure 4: Box Size Calculation

This calculation ensures that for any number of guesses and word length, the boxes are centralised and do not occupy too little or too much space on the screen and remains consistent for all boxes. There will be examples of different value combinations in the appendices below.

The aforementioned video also gave us the idea to treat letters as separate objects, that could draw themselves on to the screen. This was extremely helpful as we could update the screen every time the player typed a letter or removed a letter swiftly and with ease, giving the illusion of typing. It also allowed us to give each letter their appropriate clue colour when the user submitted the word, so that they could change their own colour independently of the other letters. A combination of these two factors results in the main gameplay process consisting of waiting for the user to enter a word, drawing the letter in the correct box as they type it, then calling the correct guess function after the row is filled and they confirm their guess with the enter key. Once the function is called, each letter is updating with their result, and they change their own background accordingly. This continues until all guesses are made or the user quits.

However, our assertions defined in the main code caused a lot of problems, as the game would crash if they entered a word wrong or did not follow the hard mode rules when selected. We decided the best way to notify the user of a misinput was to catch the error and display a popup informing them of a valid input. This is similar to how regular wordle handles this, although it does not automatically close the popup. This is something that we would improve upon in the future. We decided the simplest way to implement popups was with Tkinter windows, which turned out to be extremely easy to do. The only downside we encountered was that once the user closes the

popup, they have to manually refocus on to the PyGame window in order to continue typing. We found a Stack Overflow thread about this exact problem but had trouble implementing it. This is also something that we would like to work on in the future.

Once we finished making the GUI, we tested it by just playing the game over and over again, testing different scenarios, such as "what would happen if we selected hard mode and give an invalid guess" to make sure everything was working, and fixing anything that was slightly off. In addition, we tested the main methods and logic in the testing section of the notebook, with a bunch of print statements and expected outputs.

Overall, we are incredibly pleased with the final artefact. It is a wordle replica that logically functions very well, with a familiar, simple GUI to accompany it. All the required features are implemented, even a few of the extensions given. We have gained a very strong understanding of how wordle works and we are positive New York Times will also be able to learn a lot from this artefact.

However, as mentioned before, there were a couple of issues we had with popups and refocusing. In the future, we would improve our implementation of them to reduce these problems, with some ideas including making them messages that appear on the main UI, or by flashing a popup with a set timeframe. Furthermore, we would love to implement some other variations of wordle that exist on the Internet now, such as Quordle or Octordle. Another problem we found was that our calculations for box sizes and placement on the screen are very convoluted. Although they place the boxes in appropriate positions for reasonable values of "WORD_LENGTH" and "MAX_GUESSES", they break for values such as words of length ten and three maximum guesses. If we had more time in the future, this is definitely something we would look at correcting first – the fact that more reasonable values work gives the impression that the calculations are not that far off. Finally, we would implement another rule for hard mode that disallows the player from entering a yellow letter in the exact same space that it was in the previous guess, to make it slightly more difficult.

Learning PyGame was a very fun experience, though it was difficult to get used to initially. If it weren't for previous experiences with Tkinter, it would have proven to be an arduous task. We will definitely be using it again in the future though, now that we have some experience with making a game in it. It would also be a great experience to learn how to implement animations with the library next time; current wordle animates the tiles flipping into their colours whereas our implementation does not.

References

- Skwarecki B. 2022 "How to Get a Wordle Hint Without Spoiling the Whole Thing", Available at: https://lifehacker.com/how-to-get-a-wordle-hint-without-spoiling-the-whole-thi-1849610779 (Accessed: 14 March 2024).

- Pierce K. 2024 “Task Description for Science of Programming”, Available at: https://ncl.instructure.com/courses/49959/pages/task-description-for-science-of-programming (Accessed: March 2024).
- Harsit 2022 “How to Make Wordle with Python and PyGame in 30 mins!”, Available at: https://www.youtube.com/watch?v=mJ2hPj3kURg (Accessed: 12 March 2024).
- Stack Overflow, “How to Change Focus to PyGame Window”, Available at: https://stackoverflow.com/questions/63395415/how-to-change-focus-to-pygame-window (Accessed: 13 March 2024).

Appendices

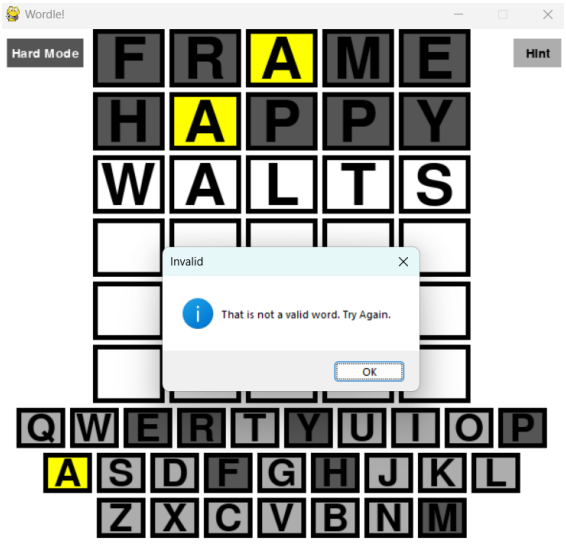

Figure 5: Invalid Word Popup

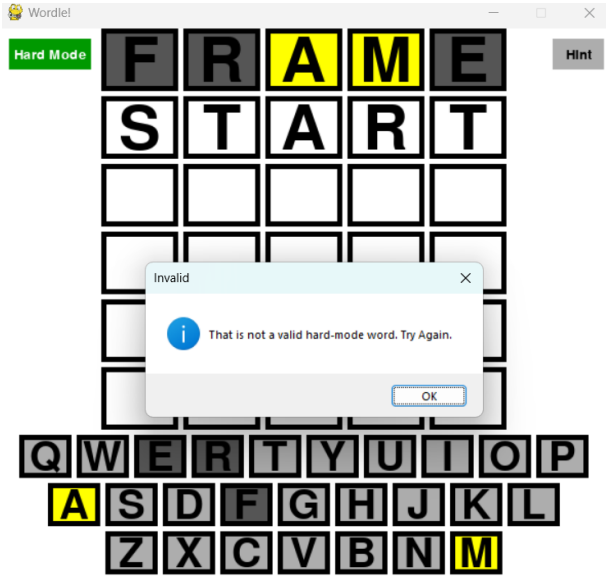

Figure 6: Invalid Hard Mode Guess Popup

Figure 7: Example of Testing

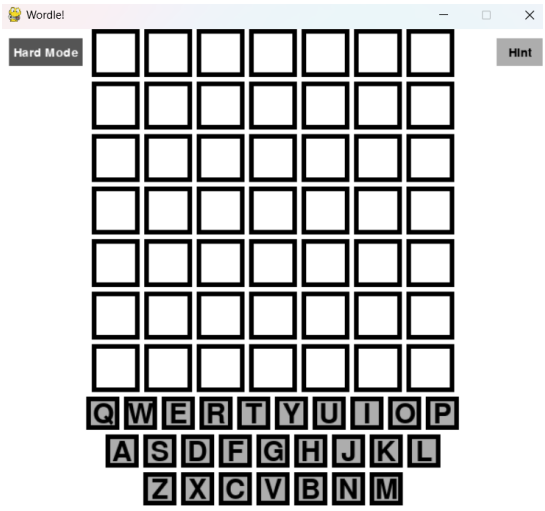

Figure 8: WORD_LENGTH = 7, MAX_GUESSES = 7

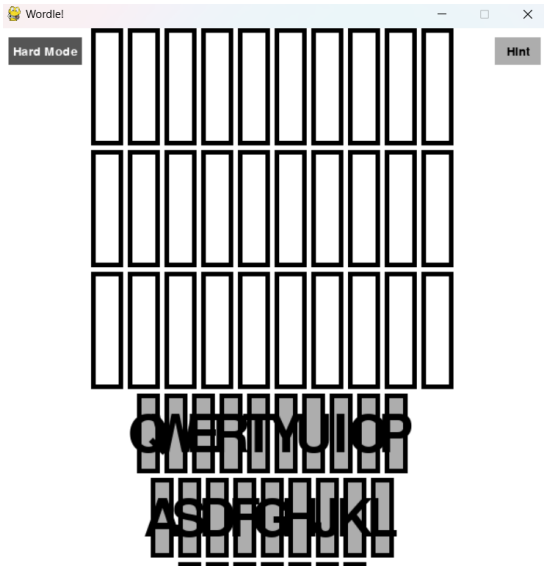

Figure 9: WORD_LENGTH = 10, MAX_GUESSES = 3

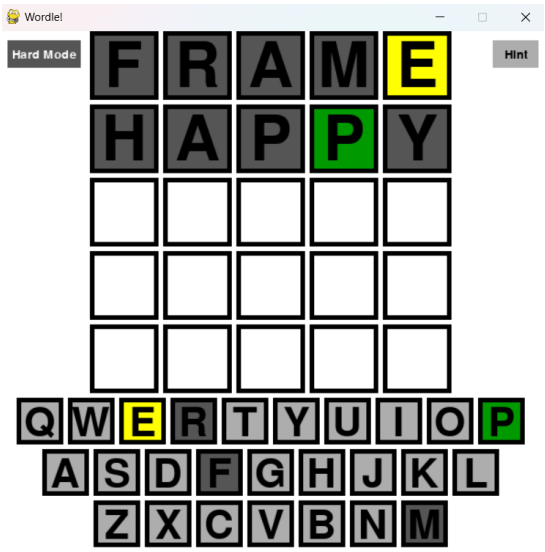

Figure 10: Duplicates Handled Correctly

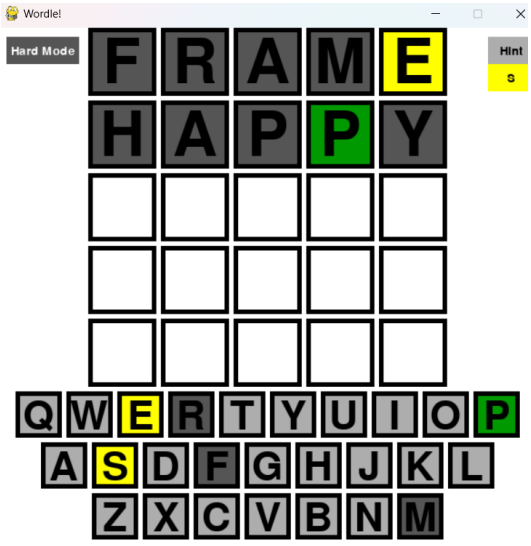

Figure 11: Hint Functionality

Figure 12: Game Win

Figure 13: Game Lose