The University of Texas
Computer Science

313E Programming Assignment 06
Evil Wordle

Setup

This assignment may be completed individually or with a pair programming partner. Be sure to include your and your partner's name and EID in the Python file header. If you are working alone, you may delete one of them.

Complete these steps to prepare for the Assignment.

Check	Description
▢	Download evil_wordle.py, test_evil_wordle.py, valid_guesses.txt and test_guesses.txt. You will be working on the evil_wordle.py file.
▢	Place all files in the same folder/directory.
▢	You may not change the file names. Otherwise, the grading script will not work.

A format called ANSI is used in this assignment to color the letters. ANSI escape codes are characters recognized by the terminal and interpreted as ways to color any characters that are surrounded by them.

The colors in the VSCode terminal may be hard to see because of the font size. You can increase terminal font size by going to the Settings editor. Navigate to File > Preferences > Settings (On Mac, instead go to Code > Settings). Then, search for **terminal.integrated.font** in the search bar. Set the font size as needed. You may also want to switch between light and dark themes on VSCode, or use the color blind mode for readability.

If you would like to turn on autosave in VSCode, click on **File -> Autosave.**

Problem Description

For this assignment, you will be programming Evil Wordle. In this version of the game, the program cheats, but in a very subtle and clever way. See the Wordle assignment description for a review of the rules of Wordle.

Evil Wordle is "evil" because the program delays picking the secret word as long as possible. At the beginning of the game, the program starts with a list of all valid words, and considers all such words "active," meaning any of these words could be chosen as the secret word. As the user guesses words, the program tries to reduce its list of active words by as little as possible in order to prolong the game and outlast the user's guesses.

All words in the active list must match the same pattern as the colored letters (aka "feedback") displayed to the user based on the current guesses. This is because Evil Wordle must cheat in a way that the user cannot detect; that is, all active words must comply with *all prior hints* that Evil Wordle has given to its user. For example, if the program told the user that their first guess of "fable" has a correctly placed letter "a," then the program cannot tell the user that their second guess of "table" has no correct letters.

In short, Evil Wordle cheats because it does not pick a secret word until one of the following scenarios occurs:

A. The user runs out of guesses, in which case the program chooses a word randomly from its active list and displays it as the secret word, or
B. The user has made smart guesses which reduce the active word list to only a single possible word. (Evidently this is not easy, hence the evilness of Evil Wordle.)

To help with cheating, Evil Wordle uses two mechanisms that are explained below: **patterns** and **word families**.

Patterns

Here is an example of how Evil Wordle works. A user guesses the word "seven" and gets the below result:

```
Enter your guess: seven
                  seven
```

Or with high-contrast mode turned on:

```
Enter your guess: seven
                  seven
```

The result of this guess shows that the current pattern of the active word list is "⬛🟩⬛🟩⬛"/"⬛🟥⬛🟥⬛". A **pattern** is a list of five colors. These colors describe the relationship between the current guess and one or more active words. In patterns, NOT_IN_WORD_COLOR tiles "⬛" represent any character that has not been guessed yet, CORRECT_COLOR tiles "🟩"/"🟥" represent characters which the user guessed in the correct positions, and WRONG_SPOT_COLOR tiles "🟨"/"🟦" represent characters that the user guessed correctly, but not in the correct position. So, given the above feedback of "**seven**"/"**seven**", the words "beget", "beret", "ceded", "defer", "deter", "rebel", etc. would be in the list of active words, since those words "match" the pattern "⬛🟩⬛🟩⬛"/"⬛🟥⬛🟥⬛" when compared to the word "seven." Patterns are useful for categorizing potential secret words; this allows the program to select difficult secret words and prolong the game further.

Another way to think of patterns is this: pick any random word from the active list (e.g. ["beget", "beret", "ceded", "defer", "deter", "rebel"]). Imagine that this word is the secret word in a game of standard Wordle. If the user guesses the word "seven," then no matter whether the secret word is "beget" or "beret" or "ceded" or "defer" or "deter" or "rebel", the resulting feedback will always be "**seven**"/"**seven**". The pattern "⬛🟩⬛🟩⬛"/"⬛🟥⬛🟥⬛" is a generalized way to describe the relationship between the user's guess and each of the active words: if we compare "seven" to each word in this active list, we see the only matching letters are the two e's (hence the two green/red tiles); no other letters match (hence the three other gray tiles).

Word Families

With many active words, it's likely that a user's guess word results in many different patterns, with each pattern corresponding to one or more active words. So, how does Evil Wordle decide which pattern to use to give feedback (i.e. the colored letters)?

The program follows these steps: every time the user guesses a new word, the program compares the current guess to each active word, obtains the pattern for that active word based on the current guess, and groups these active words together. This mapping of pattern to active words is called a **word family**. The program selects a word family and gives feedback to the user based on the selected word family's pattern.

Each **word family** consists of the following: i) a pattern, ii) a list of words that follow that pattern, and iii) a difficulty score. The colors in a pattern correspond to where the current guess has characters that are correct and in the correct spot, correct but in the wrong spot, or incorrect, in relation to each of this word family's words. A word family's **difficulty score** rates how similar the current guess is to any word in this word family. This difficulty score is calculated using the word family's pattern: NOT_IN_WORD_COLOR tiles "⬛" are worth two points, WRONG_SPOT_COLOR tiles "🟨"/"🟦" are worth one point, and CORRECT_COLOR tiles "🟩"/"🟥" are worth zero points. For example, a word family whose pattern shows four green tiles (i.e. four correct letters in the correct spots) and one gray tile means that the current guess is very similar to the words in this family. Thus, a family with this pattern has a low difficulty score of two.

To illustrate the process of creating and selecting over word families, let's continue our example with the pattern "⬛🟩⬛🟩⬛"/"⬛🟥⬛🟥⬛" and the current list of active words ["beget", "beret", "ceded", "defer", "deter", "rebel"]. If the user next guesses the word "repel", the program classifies the active words using the following patterns:

1. The pattern "🟨🟩⬛🟩⬛" (if this pattern is chosen, feedback for "repel" is displayed as: **r**🟩**e**🟩**l**)
    a. In high-contrast mode: the pattern "🟦🟥⬛🟥⬛" with feedback is displayed as: **r**🟩**e**🟥**l**
2. The pattern "⬛🟩⬛🟩⬛" (if this pattern is chosen, feedback for "repel" is displayed as: **r**🟩**e**🟩**l**)
    a. In high-contrast mode: the pattern "⬛🟥⬛🟥⬛" with feedback is displayed as: **r**🟩**e**🟥**l**
3. The pattern "🟩🟩⬛🟩🟩" (if this pattern is chosen, feedback for "repel" is displayed as: **r**🟩**e**🟩**l**)
    a. In high-contrast mode: the pattern "🟥🟥⬛🟥🟥" with feedback is displayed as: **r**🟩**e**🟥**l**

The program groups the active words into separate families based on these three possible patterns:

4. A word family with the pattern 🟨🟩⬛🟩⬛ has words ["beret", "defer", "deter"] and difficulty score 5.
    a. In high-contrast mode, this pattern is 🟦🟥⬛🟥⬛
5. A word family with the pattern ⬛🟩⬛🟩⬛ has words ["beget", "ceded"] and difficulty score 6.
    a. In high-contrast mode, this pattern is ⬛🟥⬛🟥⬛
6. A word family with the pattern 🟩🟩⬛🟩🟩 has the word ["rebel"] and difficulty score 2.
    a. In high-contrast mode, this pattern is 🟥🟥⬛🟥🟥

After creating these word family groupings, the program selects the "hardest" word family, and it will use that word family's pattern to give feedback to the user based on their guess. When a word family is selected and feedback is given to the user, all the words in that word family become the new active word list.

The "hardest" word family is defined as the family with the largest number of words. If there is a tie for the family with the most words (i.e. if two of the word families are of equal size), pick the family with the highest difficulty score. If there is still a tie on difficulty scores, pick the family that has the "smallest" pattern based on the lexicographical ordering of the patterns. Note that this final tiebreaker (pattern comparison using lexicographical ordering) is the default ordering when using the comparison operators between strings.

In summary, follow these steps in order to find the "hardest" word family:

1. The hardest word family is the family with the largest number of words.
2. If there are two or more families with the largest number of words, break the tie by picking the family with the highest difficulty score.
3. If there are two or more families with the maximum number of words and the same difficulty score, break the tie by picking the word family whose pattern is "smallest" based on the ordering of strings (the ordering of the strings is based on their Unicode values, but remember that the comparison is already implemented for you using ==, <, etc.).

The hardest word family should be found by sorting the word families in ascending order using your sort method, where the 0th index should be the index of the hardest word family.

For a demonstration of a full game of Evil Wordle, check out the Extended Example below.

Code Overview

Evil Wordle will be played through the command line. Note that similar to the prior Wordle assignment, this assignment uses ANSI escape codes to provide color in the command line. evil_wordle.py contains four constant colors: `CORRECT_COLOR`, `WRONG_SPOT_COLOR`, `NOT_IN_WORD_COLOR`, and `NO_COLOR`. `NO_COLOR` represents the default color, which depends on your display settings. Additionally, to ensure colors are not mixed between letters, `NO_COLOR` is used after each letter. Review the visible test cases and prior Wordle assignment for more information and examples on how to set up and use ANSI escape codes.

We provide the gameplay loop of handling user input and managing wins/losses in the main method of **evil_wordle.py**. The **main** function is already implemented for you, and it calls helper functions to assist with narrowing down the pool of potential secret words. The helper functions you *do not* have to implement are:

print_explanation(attempts**)**
Prints the official rules and instructions for Wordle, as well as how many attempts the user is allowed.

color_word(colors, word**)**
Formats a character (or list of characters) with a single ANSI escape code color (or list of ANSI escape code colors), and returns the colored character(s) as a string. Note: this function can be used to format either one single character (if the **colors** parameter is a string) or multiple characters (if the **colors** parameter is a list). This function handles both cases by checking if the **colors** parameter is a string, and if so, it wraps **colors** in a list.

If **colors** is a list, then len(**colors**) == len(**word**). If **colors** is a string, then len(**word**) == 1.

pre:

colors: A single ANSI escape code color or list of ANSI escape code colors
word: A string containing the character(s) to be formatted

post: Returns a string where each character in word is wrapped in the corresponding color from colors, followed by `NO_COLOR`.

get_attempt_label(attempt_number)
Generates the label for the given attempt number.

pre:

attempt_number: 1 < attempt_number < 100 and attempt_number is an integer.

post: returns a string label for a given attempt

prepare_game()

Prepares the game by setting the number of attempts and loading the initial pool of valid words through the file, "valid_guesses.txt." The function also handles optional command-line arguments such as a custom number of attempts and a debug flag.

pre: None

post: Returns a tuple (attempts, valid_words) or raises a ValueError on invalid user
attempts: The number of tries the user gets before the game automatically ends.
valid_words: A list of valid guess words and is the initial pool of secret words..

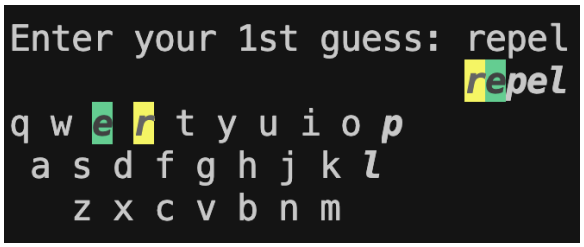
More information on these already-implemented helper functions are given in the function headers. The other helper functions (which you *will* be implementing) are listed below in Requirements.

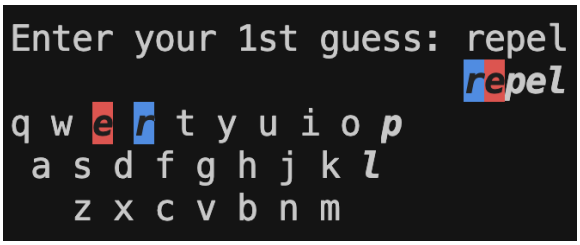
You will also implement methods for two classes: the Keyboard class and WordFamily class.

Keyboard Class

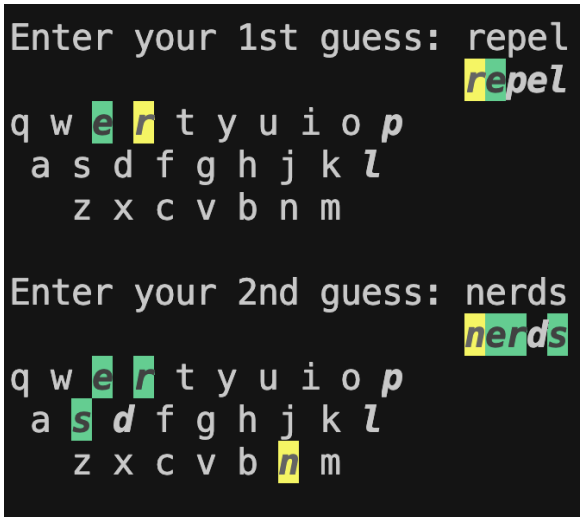
The **Keyboard** class displays and updates a text-based keyboard that prints after every guess. The colors of the keyboard letters are updated per guess to reflect which guessed letters are correctly placed, incorrectly placed, or not present in the secret word. Take a look at the Custom Wordle website to test expected behavior.

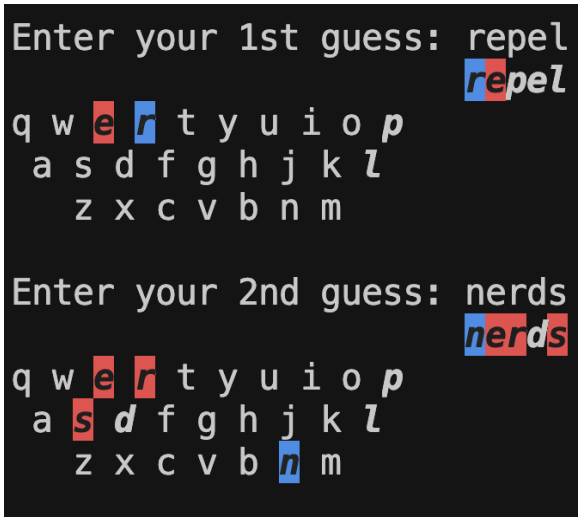
See below for examples of what your Keyboard UI should look like (high-contrast colors on the right; default colors on the left). Note this example game uses a fixed secret word of "ferns" for demonstrative purposes. Guessed letters (regardless of color) are always italicized and bolded on the keyboard (this is already taken care of for you, assuming you use the provided ANSI escape code colors).





Note that the colors of guessed letters on the keyboard may change between guesses in certain scenarios. This can happen when a letter that was previously in the incorrect spot is then placed in the correct spot in a subsequent guess. See below, where the keyboard letter "r" turns from yellow to green (or blue to red, in high-contrast mode) between the first and second guesses.





The Keyboard constructor **__init__** is already implemented for you.

WordFamily Class

The **WordFamily** class is a way to group active words by patterns after each guess. A class representing a group or 'family' of words that match a specific pattern of feedback_colors. Each word family has a difficulty level determined by the total `feedback_color` difficulty and the number of words in the family. Each WordFamily object serves the purpose of a "word family" as mentioned in the first portion of the Problem Description.

The WordFamily methods **__str__** and **__repr__** are already implemented for you.

Requirements

You will implement some functions and helper functions, methods in the Keyboard class, and methods in the WordFamily class as described in the chart below.

Name	Description
Keyboard update	Updates the color of each Keyboard letter based on feedback from a guessed word. If a letter's feedback color is `CORRECT_COLOR`, the letter on the keyboard is updated. If the color is `WRONG_SPOT_COLOR`, the letter on the keyboard updates only if the keyboard's current color for that letter is not `CORRECT_COLOR`. Letters marked with `NO_COLOR` retain that color unless any feedback changes it. pre: **feedback_colors**: A tuple of color ANSI escape codes indicating feedback for each letter. **guessed_word**: A list of string characters; the word guessed by the user len(feedback_colors) == len(guessed_word). Each item in **feedback_colors** must be a valid color constant. post: None

Keyboard String (__str__)	Returns a string representation of the keyboard, showing each letter in its corresponding color. Each row of the keyboard is formatted for readability, with spacing adjusted for alignment. Color each individual letter using color_word() based on the colors in the dictionary. See the method header for an example of the print format. pre: None post: Returns a formatted string with each letter colored according to feedback and arranged to match a qwerty keyboard layout.
WordFamily Constructor (__init__)	Create a new WordFamily from the provided pattern and list of words. The attributes in this constructor are already initialized for you; you will be implementing the difficulty score calculation for this WordFamily. pre: **feedback_colors**: The color pattern (list of ANSI escape codes) for this WordFamily **words**: A list of words belonging to this WordFamily; words != null, words.size() > 0
WordFamily Comparison (__lt__)	Compares two WordFamily objects for sorting purposes. Enforce that only WordFamily objects can be compared to each other. Using this method (and potentially some other comparator methods), you should be able to obtain the "hardest" WordFamily out of a list of WordFamily objects. Raises: A NotImplementedError if other is not a WordFamily object with the message **`< operator only valid for WordFamily comparisons.`** pre: `other` is a WordFamily object. post: True if this instance is 'less than' the other, False otherwise
fast_sort	Accepts a list and returns the sorted list. This function does **not** sort in place; it **returns a new list**. You must implement **merge sort** or **quick sort**. This function is already called for you in **main**, where it is used to sort a list of WordFamily objects. Your sorting function must be able to sort other types such as integers, floats, and strings. Thus, use comparison operators such as **<** to compare items within your implementation. Recall that using these operators to compare WordFamily objects will inherently call your **WordFamily** class's **__lt__** method implementation. pre: **Lst:** A list of words post: Returns a new list that is sorted based on the items in **lst**.
get_feedback	Processes the guess and generates the colored feedback based on the hardest word family. Use get_feedback_colors to group the words based on their feedback, and then create word families based on these groups. The hardest word family is then chosen by sorting the families, where the 0th index is now the hardest word family. pre: **remaining_secret_words:** is a list of words **guessed_word:** must be a string of exactly 5 lowercase alphabetic characters

	post: Returns a tuple (feedback_colors, new_remaining_secret_words) where: ● **feedback_colors**: a tuple of feedback colors (CORRECT_COLOR, WRONG_SPOT_COLOR, or NOT_IN_WORD_COLOR) that correspond to the remaining secret words ● **new_remaining_secret_words**: the remaining secret words, picked by choosing the hardest word family, where the hardest word family is decided by these tiebreakers: 1. Largest word family (length of the word list) 2. Difficulty of the feedback 3. Lexicographical ordering of the feedback (ASCII value comparisons)
get_feedback_colors (helper function for get_feedback())	Processes the guess and generates the colored feedback based on the potential secret word. This function should not call color_word and instead returns the list of colors used for the corresponding letters. This should be extremely similar to what you have from assignment 3: Wordle. pre: **secret_word:** must be a string of exactly 5 lowercase alphabetic characters. **guessed_word:** must be a string of exactly 5 lowercase alphabetic characters. post: The return value is a string where: ● Correctly guessed letters are wrapped with CORRECT_COLOR. ● Correct letters in the wrong position are wrapped with WRONG_SPOT_COLOR. ● Letters not in secret_word are wrapped with NOT_IN_WORD_COLOR. There will only be 5 lowercase letters with the ANSI coloring in the returned value.

This assignment includes the following technical requirements:

1. You may **not** make any changes to any of the methods/functions that are already implemented for you in evil_wordle.py (**main**, **print_explanation**, **color_word**, **get_attempt_label**, **Keyboard.__init__**, **WordFamily.__str__**, **WordFamily.__repr__**) in **evil_wordle.py**.
2. **You may not use any built-in sorting functions or methods.** You **must** implement **merge sort** or **a stable version of quick sort** for the fast_sort() method. You may **not** use selection sort, insertion sort, or any other sorting method such as the built-in sort() and sorted().
3. Before turning in your assignment, make sure your implementation works for the full valid_guesses.txt**.**
4. **You may not change the names or parameters of the functions listed. They must have the functionality as given in the specifications. You can always add more functions than those listed.**
5. **You may not import any additional external libraries in your solution besides random and sys.**

Input

You can run the program by typing the following command into your terminal:

```
python3 evil_wordle.py
```

We have provided you the file **valid_guesses.txt** which has all valid words for a game of Evil Wordle. Your program should consider all of these words as active words at the beginning of the game. If you'd like to test

with a smaller set of words for debugging purposes, run the command again ending with debug:

```
python3 evil_wordle.py debug
```

This will use the smaller set of words test_guesses.txt.

You may also override the default number of attempts (6) by passing in an integer between 2 and 99 (inclusive) to the command line arguments:

```
python3 evil_wordle.py 10
```

You can also combine this with the debug option:

```
python3 evil_wordle.py 10 debug
```

You can assume that all input files are structured correctly.

Output

Code to produce the output already exists in the file **evil_wordle.py**. The existing code will call the functions/methods that you wrote. For all example outputs, we will be using the file valid_guesses.txt as the list of words.

The following command is an example of what will be used in the test cases. The first output will be the default coloring, and the second output will be the high-contrast coloring. Colors are inverted for readability.

Default Coloring

```
Welcome to Command Line Evil Wordle!

How To Play
Guess the secret word in 6 tries.
Each guess must be a valid 5-letter word.
The color of the letters will change to show
how close your guess was.

Examples:
weary
w is in the word and in the correct spot.
pills
i is in the word but in the wrong spot.
vague
u is not in the word in any spot.

Enter your 1st guess: salet
                      salet
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 2nd guess: round
                      round
```

```
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 3rd guess: chimp
                      chimp
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 4th guess: womby
                      womby
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 5th guess: forge
                      forge
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 6th guess: kooky
                      kooky
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Congratulations! You guessed the word 'kooky' correctly.
```

High-Contrast Coloring

```
Welcome to Command Line Evil Wordle!

How To Play
Guess the secret word in 6 tries.
Each guess must be a valid 5-letter word.
The color of the letters will change to show
how close your guess was.

Examples:
weary
w is in the word and in the correct spot.
pills
i is in the word but in the wrong spot.
vague
```

```
u is not in the word in any spot.

Enter your 1st guess: salet
                      salet
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 2nd guess: round
                      round
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 3rd guess: chimp
                      chimp
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 4th guess: womby
                      womby
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 5th guess: forge
                      forge
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Enter your 6th guess: kooky
                      kooky
q w e r t y u i o p
 a s d f g h j k l
   z x c v b n m

Congratulations! You guessed the word 'kooky' correctly.
```

Extended Example

Here is a full game of Evil Wordle. Note that the program doesn't pick a secret word until forced to do so. Instead, at the start of the game, the program considers all words in valid_guesses.txt as active words. As the user makes guesses, the program tries to reduce the active word list as little as possible to prolong the game.

Suppose that the valid_guesses.txt contains only the following ten words:

[being, conks, dryly, flowy, grasp, hinge, inset, linen, olive, sidle]

At the start of the game, all words from valid_guesses.txt are considered active words. Let's say the user guesses the word "belay". The program now groups all active words based on the pattern between the current guess and each active word. Each word in the active word list falls into one of the below "word families":

- "⬛🟨🟨⬛⬛"/"⬛🟦🟦⬛⬛": the pattern for the word family [linen, olive, sidle], with difficulty score 8
- "🟩🟩⬛⬛⬛"/"🟥🟥⬛⬛⬛": the pattern for the word family [being], with difficulty score 6
- "⬛⬛⬛⬛⬛"/"⬛⬛⬛⬛⬛": the pattern for the word family [conks, grasp], with difficulty score 10
- "⬛⬛🟨⬛🟩"/"⬛⬛🟦⬛🟥": the pattern for the word family [flowy, dryly], with difficulty score 7
- "⬛🟨⬛⬛⬛"/"⬛🟦⬛⬛⬛": the pattern for the word family [hinge, inset], with difficulty score 9

Per the criteria for selecting the "hardest" word family, the program will select the family with the pattern "⬛🟨🟨⬛⬛"/"⬛🟦🟦⬛⬛" since it has the largest number of words. This reduces the active list of words to:

[linen, olive, sidle]

Then, the program reveals the selected pattern to the user in the form of colored feedback:

Default Output:

```
Enter your 1st guess: belay
                      belay
```

High-Contrast:

```
Enter your 1st guess: belay
                      belay
```

Let's continue with the user's next guess. Given this three-word active list, if the user guesses the word "close", then the program would break down the active word list into the following word families:

- A word family with pattern "⬛🟨⬛⬛🟩"/"⬛🟦⬛⬛🟥" containing [linen], with difficulty score 7
- A word family with pattern "⬛🟨🟨⬛🟩"/"⬛🟦🟦⬛🟥" containing [olive], with difficulty score 6
- A word family with pattern "⬛🟨⬛🟨🟩"/"⬛🟦⬛🟦🟥" containing [sidle], with difficulty score 6

Although all these word families are the same size, the first word family has the highest difficulty score and is thus selected by the program. The active word list then becomes:

[linen]

And the following feedback is updated for the user:

Default Output:

```
Enter your 1st guess: belay
                      belay
Enter your 2nd guess: close
                      close
```

High-Contrast Output:

```
Enter your 1st guess: belay
                      belay
Enter your 2nd guess: close
                      close
```

At this point, since there is only one word remaining in the active word list, the user will win if they correctly guess "linen" within their remaining guesses. From this point onward, the program can no longer cheat; gameplay proceeds as if this was a standard Wordle game with the secret word "linen".

Unit Testing Framework

We have provided you with unit test cases in test_evil_wordle.py. This file contains all visible unit test cases for this assignment, and it uses the Python unittest test framework. This framework uses the following format:

```
import unittest
import sys
from evil_wordle import (
    Keyboard,
    WordFamily,
    CORRECT_COLOR,
    WRONG_SPOT_COLOR,
    NOT_IN_WORD_COLOR,
    NO_COLOR,
    color_word,
    fast_sort,
    get_feedback_colors,
    get_feedback,
)

class TestKeyboardUpdate(unittest.TestCase):
    """Keyboard Update Tests"""

    def check_keyboard_colors(self, keyboard, expected_colors):
        """Helper method to check the entire keyboard color state"""
        for letter, color in expected_colors.items():
            self.assertEqual(
                keyboard.colors[letter],
                color,
                f"Expected '{color_word(keyboard.colors[letter], letter)}' to be
{color_word(color, letter)}",
            )

    def test_1(self):
```

```
        """update(): Single correct letter 'a' in word 'apple'"""
        keyboard = Keyboard()
        keyboard.update(
            [
                CORRECT_COLOR,
                NOT_IN_WORD_COLOR,
                NOT_IN_WORD_COLOR,
                NOT_IN_WORD_COLOR,
                NOT_IN_WORD_COLOR,
            ],
            "apple",
        )
        expected_colors = {letter: NO_COLOR for letter in "qwertyuiopasdfghjklzxcvbnm"}
        expected_colors.update(
            {
                "a": CORRECT_COLOR,
                "p": NOT_IN_WORD_COLOR,
                "l": NOT_IN_WORD_COLOR,
                "e": NOT_IN_WORD_COLOR,
            }
        )
        self.check_keyboard_colors(keyboard, expected_colors)
```

Each test is contained in its own method, and the **name of each method must begin with "`test`"**. Each function has its own test *class*. You'll learn more about Python classes this semester, but for now just know that the class serves as a way to group together all our tests.

The tests each check if the expected output matches the actual output (i.e. the output of your traversal.py functions when run with the given test input). If these values match for all tests, then you will see something like the output below (note that you'll see a lot more than just one test!).

```
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

If the expected and actual values do not match on one or more tests, then error(s) will be thrown:

```
======================================================================
FAIL: tes_1 (__main__.TestKeyboardUpdate)
update(): Single correct letter 'a' in word 'apple'
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_evil_wordle.py", line 53, in test_1
    self.check_keyboard_colors(keyboard, expected_colors)
  File "test_evil_wordle.py", line 25, in check_keyboard_colors
    self.assertEqual(
AssertionError: '\x1b[0m' != '\x1b[3;1m'
-
+
 : Expected 'e' to be e

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Throughout your implementation of all your programming assignments, we highly encourage you to take advantage of the existing test framework by writing your own tests. For whichever of the function(s) you'd like to test, simply create a new `test` method within the corresponding class and create any necessary testing parameters (i.e. for initializing an object/calling a method). Within this test method, provide the expected (correct) output of the test, then check to see if your actual output matches the expected output.

1. Running All Tests:

If you run the script without any arguments, all the test cases defined in the file will be executed:

python3 test_evil_wordle.py

However, we suggest you run it using the following syntax so you can easily change to run more specific tests:

Windows Command:

python -m unittest test_evil_wordle

Mac Command:

python3 -m unittest test_evil_wordle

2. Running All Tests for a Function:

If you run the script with the test name, it will run only the test cases for that function:

Windows Command:

python -m unittest test_evil_wordle.[test_class]

Mac Command:

python3 -m unittest test_evil_wordle.[test_class]

Windows Example Command:

python -m unittest test_evil_wordle.TestKeyboardUpdate

Mac Example Command:

python3 -m unittest test_evil_wordle.TestKeyboardUpdate

3. Running A Specific Test:

If you'd like to run a specific test, you can provide the name of the class followed by the test number for that test. The format for this command is:

Windows Command:

python -m unittest test_evil_wordle.[test_class].[test_name]

Mac Command:

python3 -m unittest test_evil_wordle.[test_class].[test_name]

Windows Example Command:

python -m unittest test_evil_wordle.TestKeyboardUpdate.test_1

Mac Example Command:
`python3 -m unittest test_evil_wordle.TestKeyboardUpdate.test_1`

This command will run the first test for the TestLongestSubstringN3 class.

4. Run Tests with Failfast Option

To stop execution on the first failure, include -f:

Windows Example Command:
`python -m unittest -f test_evil_wordle.TestKeyboardUpdate`

Mac Example Command:
`python3 -m unittest -f test_evil_wordle.TestKeyboardUpdate`

5. Command-Line Arguments

- **`test_class`**: The name of the specific test you want to run. The valid options are:
  - `TestKeyboardUpdate`: Tests the `update()` function.
  - `TestKeyboardStr`: Tests the `__str__()` function.
  - `TestWordFamilyDifficulty`: Tests the `WordFamily()` class.
  - `TestWordFamilyComparison`: Tests the `__lt__()` function.
  - `TestFastSort`: Tests the `fast_sort()` function.
  - `TestGetFeedbackColors`: Tests the `get_feedback_colors()` function.
  - `TestGetFeedback`: Tests the `get_feedback()` function.
- **`test_name`**: test_1 - test_4 for TestKeyboardStr, test_1 - test_6 for TestWordFamilyDifficulty, for all other test classes test_ 1 - test_10. Functional test cases are named after their final secret word, such as test_sages

Grading

Visible Test Cases - 60/100 points

The program output exactly matches the sample solution's output. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Hidden Test Cases - 30/100 points

The program output exactly matches the sample solution's output. To receive full credit, the program must produce the correct output based on all requirements in this document and pass all the test cases.

Style Grading - 10/100 points

The evil_wordle.py file must comply with the PEP 8 Style Guide.

If you are confused about how to comply with the style guide, paste the error or the error ID (e.g. C0116 for missing function or method docstring) in the search bar here in the Pylint Documentation, and you should find

The TAs will complete a manual code review for each assignment to confirm that you have followed the requirements and directions on this document. Deductions will occur on each test case that fails to follow requirements.

Pair Programming Guidelines

Pair programming means working on an assignment together, looking at the same code. One person drives (types at the keyboard) while the other person observes, comments, and makes suggestions to the driver. This form of programming can speed up the programming process by having a helper immediately available. It also reduces the likelihood of syntax and design errors, as there are two pairs of eyes watching for mistakes. If the driver tries out a flawed solution, the partner is there to provide constructive criticism and helpful alternatives. **Pair programming does not mean taking an assignment and partitioning into two pieces and then having each person complete a piece.**

Here are the requirements for pair and group programming in this course:

1. Every team member present must act as a driver, and the driver must change every 30 minutes.
2. At least 70% of your time must be spent working together collaboratively, with all members of the group present, in communication with each other, and looking at the same code.

Submission

Follow these steps for submission.

Check	Description
☐	Ensure that you have completed the honor statement in your coding file. ***NOTE***: You will not receive credit for this assignment if your honor statement is not filled out.
☐	Verify that you have no debugging statements left in your code. This may cause your test cases to fail.
☐	Run the visible test cases on your implementation. This is recommended prior to submission, as you have a maximum of 15 tries to test your code on the hidden test cases via Gradescope.
☐	Ensure that you are following PEP 8 style guidelines via Pylint. ***NOTE***: To receive full credit for style points on your programming assignment, you must have addressed ALL problems detected by Pylint.
☐	Submit evil_wordle.py to the assignment in Gradescope. You **must** submit via Github, regardless of if you are in a group. To submit as a group, follow the instructions here. This will run the grading scripts (hidden + visible tests).
☐	When the grading scripts are complete, check the results. If there are errors, evaluate the script feedback, fix your code, test your code, and then re-submit the file to Gradescope. You can submit on Gradescope up to 15 times until the due date. **You will not be able to submit after your 15th submission**. So, if you're failing hidden test cases, try to narrow down and fix your bug prior to

Check	Description
	submitting! More techniques on this are described in the Gradescope guide here and in the VS Code Debugger guide here. ***NOTE***: By default, only the most recent submission will be considered for grading. If you want to use a previous submission for your final grade, you must activate it from your submission history on Gradescope before the due date.

Academic Integrity

Please review the Academic Integrity section of the syllabus. We will be using plagiarism checkers as well as checking for AI-generated code. Remember, the goal in this class is not to write the perfect solution; we already have many of those! The goal is to learn how to problem-solve, so:

- don't hesitate to ask for guidance from the instructional staff, and
- be sure you stay within the Academic Integrity discussion guidelines outlined in the syllabus.

Attribution

Thanks to Dr. Carol Ramsey for the instruction template and to Mike Scott for inspiration on this assignment.