PyWordle

Alison Gale

Background

This project is inspired by the popular word-guessing game, Wordle. In the game, users have six guesses to identify a hidden five-letter word. With each guess the user gets feedback on which letters aren't in the word, which are in the correct location, and which are in the word but in a different location. There are some interesting edge cases when a word with multiple of the same letter is guessed which results in different coloring depending on how many times that letter is in the hidden word. This game inspired a series of spin-off games with different themes (for example Taylordle is Taylor Swift themed) and a swath of solvers to find the optimal set of guesses to identify the hidden word.

Users

For this package, there were two sets of users I considered. The first category of users were creative folks wanting to create themed versions of the game. These users would want to just provide a solution or list of solutions and not have to worry about the logic inside the game. The second category of users were folks wanting to play unlimited games of Wordle. The official game only lets you play one game per day. I also had considered a third category of users: ones building solvers or other algorithms that would need to interact with the game state. These users might be more technically profficientand would care about things like performance. For reasons discussed further in the design section, I chose to focus on the first two categories of users.

For both of the use cases I'm focusing on, the use cases are pretty similar with the main difference being the set of possible solutions being provided to the game:

1. User passes in a list of custom words (or the official set of words from Wordle)
2. User creates a game
3. User guesses words
4. Program responds with feedback on which letters are correct

Given that I wanted to build a game engine that would allow for creating themed versions of Wordle, the main requirement was implementing a mechanism to enforce the rules of the game. This includes enforcing the number of guesses, whether guesses are valid, and enforcing "hard mode" constraints where information from previous guesses must be used in subsequent guesses. Outside of that, I wanted to provide the ability to provide a custom set of solutions to choose from and the ability to pick a solution for a given game. I also wanted to provide good validation of inputs and data to ensure that the game is played correctly.

Design

This package utilizes two modules, Wordle and Game. The Wordle module represents a themed version of the original game, while the Game module represents a specific instance of the game where a user is making guesses towards the hidden solution. The diagram below highlights the methods in each module:

Wordle	**Game**
__init__(solutions) start_game(solution, hard_mode) __repr__()	__init__(solution, hard_mode) guess(word) is_valid(word) get_status() plot_progress() __str__() __repr__()

Within the Game module, I utilized common libraries like enum and defaultdict to represent the state of previous guesses in order to enforce hard mode constraints. When hard mode isn't enabled, the process of validating guesses is very straightforward in checking whether a word is a valid word in the English language. But when hard mode is enabled or when printing out the string representation of a game board, there is a complex set of constraints that must be checked.

The most interesting part of this logic is how to handle guesses with duplicates of a letter. A letter can only be colored yellow or green at most the number of times the letter appears in the hidden solution. Priority is given to letters colored green and then yellow letters are colored from left to right until the correct number is reached. Any remaining ones are colored white. This can provide interesting information to the user about the number of times a letter appears in the final solution.

The method to get whether a guess is valid is very simple, but will help users that would prefer to not rely on catching exceptions. I personally prefer to explicitly check for error conditions like this because I find it to be more readable than catching an error and checking information about the error type or message.

The method to plot progress towards the solution was a first step towards providing interesting game analytics. In the official game, you just get a print-out of the color coded guesses that led to the solution. In order to improve your game, it would be interesting to see which guesses really narrowed the set of remaining solutions. This visualization provides a rough estimate of that progress.

Within the design, I relied heavily on dunder methods like __str__ and __repr__ to allow for a more Pythonic style. So the pretty-print version of the game board is implemented in a __str__

method. I considered having the game return an object representing the state for the user to handle on their own, but I found that there is a lot of complex logic in how to print out the board based on the previous game state that would be useful to incorporate into the game engine. This allowed the module to be deeper as well because it doesn't change the public API, but does make for a much richer functionality that is provided by the public API.

Key Decisions

When designing this package, I initially planned to make a general purpose module that could serve the use cases of building a game engine and building a solver. As I worked through the method signatures and the state needed to represent this logic I found that the methods used by the solver had almost no overlap in functionality with the game engine methods. For example, when building a solver you are interested in which words are eliminated based on information from previous guesses. Unfortunately, the hard mode that the game enforces is a more loose constraint than what would be needed by a solver. For example, in hard mode you can use letters you know aren't in the word but in a solver you wouldn't want to use those letters. Because these use cases were very disjoint, I decided that it didn't make sense to include this logic in the Game module which would increase the complexity of the module and the size of the public API, while not benefitting the two main user types I was focusing on.

Another big decision was whether to implement this package as a single module or two different modules. In some other libraries I looked at, there was a single module that would allow you to play a game. This would let you pass in the set of words and game options at once before starting a game. There are two ways this could be structured: you could either pass this all into a constructor to play a single game or you could have a separate method to start a game. In the first case the downside is that you must revalidate and process the list of solutions each time you play a game. In the second case, it would be challenging to keep track of the state of multiple games if they were being played at once. For those reasons I decided to split the logic of processing and validating the list of solutions in a small outer module that facilitates the creation of an instance of the Game module.

Extensibility

When designing this package, I wanted to make sure it would be easy to expand it to support a broader set of use cases. The most obvious way to expand the game would be to support modified game constraints like allowing different length words or different numbers of guesses. At this time, these options aren't configurable by the end user because I didn't have dictionaries for words of different lengths. But, when designing the package, I made sure to abstract out these constants so they would be easy to change after the fact because they aren't hardcoded.

For other extension points, it would mostly make sense to utilize the Wordle or Game modules within another module. One example would be if someone wanted to write a solver that used the Game module to enforce the rules of the game. There are some non-text based spin offs of Wordle, but due to the substantial differences in how guesses are evaluated and results are rendered, it wouldn't make sense to extend this package to support those use cases. There is

little overlap in the actual logic so the module would become unnecessarily complex trying to handle the different use cases.

Usage

This library allows you to easily create a themed instance of the Wordle module by providing a set of possible solutions. From this instance you can create individual games. By default it has hard mode disabled and selects a random solution. From there the user can make guesses towards the solution and print out the current state of the game. Here is an example of a basic interaction with the package:

```
from pywordle import Wordle

wordle = Wordle(WORD_LIST)
game = wordle.create_game()
game.guess("SPILL")
print(str(game))
```

Comparison

I will compare my library to the wordle-python library (https://pypi.org/project/wordle-python/) that is available on PyPI. The main difference between our libraries is that the wordle-python library handles all the I/O with the user while my library allows whoever uses the library to control receiving user input and passing it to the game. Here is an example of how that library is used:

```
import wordle

wordle.Wordle(word = wordle.random_answer(), real_words = True).run()
```

This difference allows the wordle-python library to have a single module as part of the public API because you just call a run method to start the game. I could have consolidated all of my logic into a single module that would take in a word list and settings, but the downside of this would be that for each game you create you would have to revalidate the list of solutions. My design allows that validation to be done once which improves performance for repeated games.

The wordle-python library is a highly specialized module that only lets you play with a specific set of words. It is deep because the public API is tiny, but it is not very extensible or flexible. This means that it is very useful for someone who wants to play that specific game on the command line, but if someone wanted to make a Flask app that allowed someone to play the game on a website they wouldn't be able to use the library.