# I Built a Wordle Cheater

---

## Overview

- Instruction

To run my code, you need to download all the files in the `src` folder. Once downloaded, execute the following command:

```
python game.py
```

The game offers two play modes:

1. **debug mode**: If you want to see the assistant in action directly, you can simply enter `1`. In this mode, the solution is entirely automated by the cheat program.
2. **manual mode**: You can freely play Wordle, and if you encounter difficulties, simply type `yes` to get some outside help anytime. 😊

- File Description

### 1.full_data.csv

A word bank containing over 12,000 words is stored here. The Wordle program will randomly select a word from this list for gameplay. It also serves as a critical dataset for the cheat program. This dataset was extracted from the original Wordle source code.

The extracted dataset originates from:

https://github.com/uilicious/wordle-solver-and-tester/blob/main/src/word-list/original-wordle-list.js

(Additionally, this author also provides an effective Wordle-solving approach! Their method involves calculating the frequency of individual letters at each position and designing a penalty mechanism based on the current guess state. A score is assigned to each letter to determine the most promising next guess.)

### 2.game.py

In this script, I built a simplified Wordle game logic solely to facilitate the development of the cheat program.

### 3.cheater.py

This script contains all the cheat logic based on the dataset, which will be explained in detail in the later sections of this document.

## Step One: Filter Words

First, in `cheater.py` , I updated words list based on previous guesses, ensuring it only contains words that could potentially be the answer.

The `filter_words` method aims to reduce the list of possible words (remaining_words) by applying the feedback (green, yellow, and gray) from previous guesses. The goal is to ensure that only words conforming to all feedback rules remain in the list.

After processing all words for a given guess-feedback pair, the `remaining_words` list is updated to contain only the filtered words.

## Step two: Entropy-Based Word Selection

Information entropy is a measure of uncertainty in a system. In the context of Wordle, entropy quantifies the "information gain" of a guess, which means a good guess should have enough information to distinguish itself from remaining word selections.

**Formula:**

$$H(X) = -\sum_{i=1}^{n} P(x_i) log_2 P(x_i)$$

- $X$: A random variable representing feedback patterns.
- $x_i$: A specific feedback pattern (e.g., `["green", "gray", "yellow","gray","gray"]`).
- $P(x_i)$: The probability of a feedback pattern $x_i$ occurring.
- $H(X)$: The average information (in bits) conveyed by $X$.

Higher entropy means more information, when a system has higher entropy, it implies that observing the outcome of an event provides more new information.

In the Wordle case, the higher the information entropy of a word in the possible word list, the more information it can provide about the correct answer.

First, the program will iterate through the `remaining_words` list filtered in the previous step, treating the current element as an **answer candidate**. It then determines the **feedback pattern** for the remaining elements excluding the candidate and calculates the frequency $P(x_i)$ of each pattern $x_i$:

$$P(x_i) = \frac{Number\ of\ words\ matching\ pattern\ x_i}{Total\ remaining\ words}$$

Based on $P(x_i)$, we can calculate the entropy $H(X)$ of current candidate. This process is repeated to compute the entropy for all remaining words when treated as the **answer**. Finally, the word with the highest entropy is selected as the next guess.

## here is an example:

- **remaining words:**

  remaining_words = ["apple", "ample", "ample", "angle"]

- **Candidate Guess:**

  candidate="ample"

- **Feedback Generation:**

`"ample"` **vs** `"apple"` → `["green", "green", "gray", "green", "green"]`

`"ample"` **vs** `"ample"` → `["green", "green", "green", "green", "green"]`

`"ample"` **vs** `"amber"` → `["green", "green", "yellow", "gray", "gray"]`

`"ample"` vs `"angle"` → `["green", "gray", "yellow", "gray", "green"]`

| FEEDBACK PATTERN | WORDS MATCHING PATTERN | FREQUENCY |
|---|---|---|
| `["green", "green", "gray", "green", "green"]` | `"apple"` | 1 |
| `["green", "green", "green", "green", "green"]` | `"ample"` | 1 |
| `["green", "green", "yellow", "gray", "gray"]` | `"amber"` | 1 |
| `["green", "gray", "yellow", "gray", "green"]` | `"angle"` | 1 |

### code for this section:

```
def calculate_entropy(guess, remaining_words):
    pattern_counts = defaultdict(int)

    for possible_answer in remaining_words:
        pattern = get_pattern(guess, possible_answer)
        pattern_counts[pattern] += 1

    total_words = len(remaining_words)
    entropy = 0
    for count in pattern_counts.values():
        p = count / total_words
        entropy -= p * math.log2(p)

    return entropy
```