# Hangman Game
# Project Documentation

A program that plays the guesser role in hangman game

Hangman is a well-known game; I am sure you have played it previously in school or with friends. But did you think for once, what is the best way to play it? Or as we programmers love to call it, ‘**the OPTIMAL way**’ ?

Our project idea is to create a program that plays as the guesser, ‘thinking’, and making reasonable guesses, winning at a high rate.

So when we were thinking about this project, we discovered a greedy algorithm[1] that utilizes a dataset of words, to make the best guess at each round.

If we think about the way we play the game, we try to find similar words, with same length, guess a letter, if wrong, we think of similar words that do not contain this letter… Essentially, the program mimics our way, but in a more organized defined way. And with its ability to scan, filter & digest thousands of words very quickly, it will outperform us for sure.

[1] Greedy algorithm is an algorithm that focuses on the ‘current moment’ maximum gain possible, not caring or thinking about the future or the long-term better choices. Though it may seem trivial, but its applications are numerous, and its simple idea can make many hard complex problems easy to solve, maybe it will not find the best solution, but it will definitely find one. read more

So now we have the algorithm, but how can we implement it?

Before diving to the code directly, it is better for our projects to be modular, so we divided our project to several modules:

- Main
- Thinker
- Word
- Data

We'll dive into the details of each module soon, but for now, it is enough to state that each module represents a role, so as if they are different people collaborating to play hangman together as one player.

Now the project has 4 steps:

- **Data Preparation**
- **Planning**
- **Coding**
- **Testing**

## Data Preparation

Data preparation is a crucial step. Data quality is the key for our program performance. Bad data will make a loser. Also, the dataset needs to contain many words, as the size of it will determine the program ability to cover and know a lot of words, and thus survive even for hard or strange ones.

So, we searched for a dataset with such properties. We found one, but there exists nothing perfect. As any dataset out there, it needs cleaning.

Our dataset contains 333,333 word, but a lot of them are actually invalid English words, they are just random combinations of few letters. Using pandas library, we managed to filter words based on occurrences count, and removed most of invalid words. Having around 50,000 words remaining, we split the words into groups, based on word length, to save each group into an independent text file.

Our data is ready!

## Planning

What is an outline?

Since we are working as a team, we needed a way to organize all our ideas, work in parallel, and clarify anything unclear till now.

An outline is a module structure, which includes all classes & methods declaration (without implementation), and specifies the flow of the program.

We sat all together, discussed the project ideas and the flow of the program, assigned roles, and agreed to a timeframe for the coding part.

Then each one wrote the outline of his module, making sure all modules outline align together.

Thanks to this step, coding became much easier, because every idea is now clearer, the collaboration between different modules is defined, and all methods and classes are just waiting to be filled some clean code.

Here is a brief documentation of the code:

# Code Outline

## Main Module

The main module in the Hangman game project serves as the controller, responsible for facilitating user interaction and monitoring the progress of the game. It acts as the intermediary between the user and other modules, primarily the "Thinker" module. The main module does not perform significant computational tasks by itself but orchestrates the game flow and coordinates actions with the "Thinker" module.

**Role of Main Module :**

1. **User Interaction:**
   - The main module initiates the conversation with the user, guiding them through the gameplay process.
   - It prompts the user to input the desired word length and interacts with them to collect responses during the game.
2. **Game Progress Monitoring:**
   - Throughout the game, the main module monitors the progress, ensuring that the game proceeds smoothly and according to the rules.
   - It tracks the number of attempts remaining and handles the end of the game gracefully.
3. **Interactions with other Modules:**
   - The main module interacts primarily with the "Thinker" module to make guesses, process user responses, and update the game state.
   - It imports and utilizes functions from the "Thinker" module to perform these actions.

**Functionality Overview:**

1. **play_game():**

   This function manages the entire gameplay process, from initializing the game environment to handling the end of the game.

**Parameters:**

- word_length (int): represents the length of the word the player needs to guess.
- Tries (int, optional): represents the number of attempts allowed to guess the word. Default is 10.

2. **is_word_guessed():**

   This utility function checks whether the word has been completely guessed by examining the current state of the word.

   **Parameters:**

   State (str): Represents the current state of the word with guessed letters filled in.

   **Returns:** (bool) returns True if all letters have been revealed, indicating that the word has been guessed, and False otherwise.

3. **right_guess():**

   This function prompts the user to confirm whether the guessed letter is present in the word.

   **Parameters:** guessed_letter (str): Represents the letter guessed by the program.

   **Returns:** (bool) returns True if the user confirms the presence of the guessed letter, otherwise False

4. **is_numeric():**

   This utility function checks whether a given string consists only of numeric characters.

   **Parameters:** s (str): Represents the string to be checked.

   **Returns:** (bool) Returns True if all characters in the string are numeric and False otherwise.

5. **get_positions():**

This function prompts the user to input the positions of the guessed letter within the word.

**Parameters:** guessed_letter (str): Represents the guessed letter for which positions need to be entered.

**Returns:** (list[int]) Returns a list of integers representing the positions entered by the user.

6. **handle_game_end():**

This function handles the end of the game by prompting the user to enter the word if it was not guessed correctly.

**Parameters:** word_length (int): Represents the length of the word the player needs to guess.

7. **get_word_length():**

This function prompts the user to input the desired length of the word to be guessed.

Returns: int: Returns the length of the word specified by the user.

# Thinker Module

This module, as its name suggests, performs the task of thinking and making guesses for the program. The 'Thinker' class is responsible for providing the best possible guess each time, relying on the frequencies of letters within the set of words. The optimal guess is determined by the most frequently occurring letter. This module interacts directly with the **main module**, the **word module**, and the **data module**. Its objective is to facilitate the connection between the main module and the word module. This is achieved by delivering the user's response from the main module to the word module, and conveying the updated word state from the word class back to the main class.

## Class Overview:

**Attributes:**

The initializer accepts the number of letters 'n' for the user's word from the 'main' module.

- **dataset** (list): This attribute holds the dataset of words with 'n' letters, utilizing the ‘read_dataset_for_length’ method in the data module.
- **guessed_letters** (list): It stores the guessed letters to prevent re-guessing a letter.
- **word** (Word_manager): Initialized by the number of letters, the ‘thinker’ serves as the intermediary between the ‘word’ module and the ‘main’ module.

## Functionality Overview:

1. **buildFrequency():**
   This function constructs a dictionary and accesses the words within the dataset. It then adds the encountered letters in the words list along with the number of times each letter occurs.

**Returns:** (dict): frequency dictionary, where the keys are the letters encountered within the dataset, and the values are their respective frequencies.

2. **mostFrequent():**
   This function is designed to optimize the guess by selecting the letter with the highest frequency of occurrences.

   **Parameters:** frequency (dict): the pairs (letter-frequency)

   **Returns:** (str) most_freq.

3. **guess():**

   This function uses the two previous methods to make the guess, then appends the guess to 'guessed_letters' list.

   **Returns :** (str) guessed letter

4. **think():**

   This function interacts with the main and word modules.

   - Delivers given user response to word.update_state()
   - Updates the dataset attribute using filter_words() method from the data module based on user response

   **Parameters:**

   - letter (str): guessed letter
   - positions (list[int]): list of positions of the letter in the word. If empty, letter is not in word

   **Returns:** (str) the updated word state as a string by calling word.getState()

# Data Module

The Data module serves as the manager of word datasets, who facilitates interactions among various program modules. Its primary functions include handling data operations such as loading and updating word datasets efficiently.

<u>**Functionality Overview:**</u>

**Methods:**

**1. read_dataset_for_length():**

**Parameters:** num_letters (int): length of words needed

**Returns:** (list[str]) list of words read from file with corresponding num_letters

**2. learn_word():**

This function checks if the given word is a new word. If it is new, it appends it to the file and renames the file to indicate an increase in the size of words.

**Parameters:** word (str): the word to be learned

**3. filter_words():**

This function uses a Word Manager class to generate a pattern based on the letter and positions, and then filters the dataset to include only words matching that pattern**.**

**Parameters:**

- dataset (list[str]): list of words of same size
- letter (str): letter to be filtered on
- positions (list[int]): list of correct positions of letter in word

**Returns:**

(list[str]) list of filtered words

# Word Module

The Word_Manager class serves as a tool for managing and manipulating words, particularly for tasks like word guessing games or word puzzles. It provides functionality to track the state of a word (with hidden letters), generate regex patterns for word matching, and update the word state based on user input.

## Class Overview:

### Attributes:

- wordState (list): represents the current state of the word, where each letter is either revealed or hidden. Underscores (_) in the list indicate hidden letters, while actual letters represent revealed positions.

### Functions Overview:

**1. def getState** ():

**Returns:**

(str): The current state of the word, with hidden letters represented by underscores.

**2. def updateState()**:

Updates the word state by revealing new letters at specified positions.

**Parameters:**

- letter (str): The letter to reveal in the word state.
- positions (list[int]): A list of integers representing positions to reveal the letter.

**Raises:**

- (ValueError) if any position is invalid (negative, out of range, or already occupied).

**3. def getPattern()**:

Generates a regex pattern based on a given letter, its positions in the word, and the word length.

**Parameters:**

- letter (str): The letter to generate the pattern for.
- positions (list[int]): A list of integers representing positions where the letter appears.
- word_length (int): The total length of the word.

**Raises:**

- (ValueError) if the positions are invalid or out of range.

**4. def isPattern()**:

**Parameters:**

- word (str): The word to check against the compiled pattern.
- compiled_pattern (re.Pattern): The compiled regex pattern for matching.

**Returns:**

(bool): True if the word matches the pattern, False otherwise.

## Testing

For any project that consists of different packages or files, life becomes easier if we separate the 'validity' of each module, and also of each function. Instead of testing the whole project as a user, wasting time to insert inputs again and again and fix the errors on the run for different modules and methods, unit testing allows us to test each method independently from everything else.

So we wrote a test for each module, using unittest library.

We then fixed any issues left and handled different edge cases.

Try to beat our program ☺