# SCRABBLE GAME

## Game Requirements:

- This needs to be able to be run as a function as shown below (not an input statement!)

- Name the python file: `scrabble.py` with the main function inside `scrabble.py` named `run\_scrabble`

- **Make a separate module named `wordscore.py`** which contains, at a minimum, a function called `score\_word`. This `score\_word` function will take each word and return the score (scoring dictionary is described below). Import this function into your main `scrabble.py` program.

**- \*\*NOTE\*\* Only have the function defs and import statements in your `.py` files as the autograder will fail if there are other statements outside the functions.**

- Allow anywhere from 2-7 character tiles (letters A-Z, upper or lower case) to be inputted.

- Do not restrict the number of same tiles (e.g., a user is allowed to input ZZZZZQQ).

**- Return two items from the `run\_scrabble` function:**

- 1) The \*\*total\*\* list of valid Scrabble words that can be constructed from the rack as (score, word) tuples, sorted by the score and **then by the word alphabetically as shown in the first example below. All outputted words need to be in upper case.**

- 2) The Total number of valid words as an integer

- See examples below for the required output. The autograder is looking for this output so please make sure your solution is in the same format shown.

- You need to handle input errors from the user and suggest what that error might be caused by and how to fix it (i.e., a helpful error message). \*\*Return\*\* this error message as a string from the run\_scrabble function (do not raise an exception).

- Implement wildcards as either `\*` or `?`. There can be a total of \*\*only\*\* two wild cards in any user input (that is, one of each character: one `\*` and one `?`). Only use the `\*` and `?` as wildcard characters. A wildcard character can take any value A-Z. Replace the wildcard symbol with the letter in your answer (see the second example below).

- Wildcard characters are scored as 0 points, just like in the real Scrabble game. A word that just consists of two wildcards can be made, should be outputted and scored as 0 points.

- In a wildcard case where the same word can be made with or without the wildcard, display the highest score. For example: given the input 'I?F', the word 'if' can be made with the wildcard '?F' as well as the letters 'IF'. Since using the letters 'IF' scores higher, display that score.

- For partial credit, your program should take less than one minute to run with 2 wildcards in the input. For full credit, the program needs to run with 2 wildcards in less than 30 seconds.

- Write docstrings for the functions and puts comments in your code.

- You may only use the Python standard library in this assignment. However, any function in the standard library is allowed.

## Pseudocode

Instead of permutating the input to check all possible values in the file, check each value in the file and see if it can be manipulated into the input.

Walk-through:

- create a wildcard dictionary to track what characters were used for wildcards

- for each word in the file (sowpods,txt) create a temp variable that is the input itself, check if the length is bigger than input, if so skip to next word

- if the lengths are equal (or less), loop through the characters in the current word (in the file)

- if we reach a character that doesn't exist in our input, check if wildcards exist, if not skip word

- if the character (or a wildcard for a non-existent character) exists in our input, remove it from the temp input value

- update the wildcard dictionary if a wildcard was used for that character

- continue until we have looped through all the characters in the current word in the file

- if the loop finished (for a word in the file) make sure the length is equal or less than the input

Here is some pseudo code:for word in all\_words:

            # for each word re-initialize a temp value that is set equal to the input -> will be removing from this

            temp = [c for c in input\_word]

            count = 0

            for char in word:

                if char in temp:

                    count += 1

                    temp.remove(char)

                elif "\*" in temp:

                    count += 1

                    temp.remove("\*")

                    letters\_not\_scored[word] += char

                elif "?" in temp:

                    count += 1

                    temp.remove("?")

                    letters\_not\_scored[word] += char

                else:

                    break

                if len(word) == count:

                    valid\_words.append(word.lower())# I want nowto run the scoring function on each with the passed in letters\_not\_scored value for that valid word

## Score\_Word function

The score\_word function calculates the Scrabble score for a given word, considering wildcard characters.

* **Arguments:**
  + word: The word for which the score needs to be calculated.
  + letters\_not\_scored: A dictionary containing letters replaced by wildcard characters in each word.
* **Returns:**
  + The Scrabble score for the word.

Previous function: returned one argument

return sum(scoring\_dict.get(letter, 0) for letter in word.upper())

## from itertools import permutations

The **permutations** function from the **itertools** module is used to generate all possible permutations of a sequence (or iterable) of elements. Each permutation is a rearrangement of the elements in the input sequence.

In the context of the Scrabble game, **permutations** is used to generate all possible combinations of letters from the rack. For example, if the rack contains the letters "CAT", **permutations** will generate permutations like "CAT", "CTA", "ATC", etc. This is useful for finding all possible words that can be formed using the letters in the rack.

Other alternatives to **permutations** include **combinations** and **combinations\_with\_replacement** from the **itertools** module. However, these functions generate different types of combinations:

* **combinations**: Generates all possible combinations of a specified length from the input sequence, without repeating elements.
* **combinations\_with\_replacement**: Generates all possible combinations of a specified length from the input sequence, allowing repeated elements.

In the context of Scrabble, **permutations** is the most suitable choice because it allows for rearranging the letters in the rack to form different words, which is essential for the game's rules.

## Score\_Word function

The **score\_word** function you provided calculates the score of a word in a Scrabble game, considering both regular letters and letters replaced by wildcard characters (**\*** or **?**). Here's what it does:

1. It initializes a dictionary called **scoring\_dict** that maps each letter to its Scrabble score.
2. It initializes the score to 0.
3. It iterates through each character in the word:
   * If the character is a regular letter (not a wildcard), it adds its score to the total score.
   * If the character is a wildcard, it skips it and does not add any score.
4. After iterating through all characters, it ensures that the score is not negative.
5. It then subtracts the score of any letters replaced by wildcard characters from the total score. This is done by iterating through the characters in **letters\_not\_scored**, which contains the letters replaced by wildcard characters for the given word.
6. It ensures that the final score is not negative.
7. Finally, it returns the calculated score.

In your example **score\_word("word", {})**, since there are no wildcard characters (i.e., the **letters\_not\_scored** dictionary is empty), the function will calculate the score for the word "word" without any modifications and return the appropriate score.

## Scrabble Function

In the **run\_scrabble** function:

* The function takes a Scrabble rack as input and returns all valid Scrabble English words that can be constructed from that rack, along with their Scrabble scores, sorted by score.
* The rack must consist of 2 to 7 characters (A-Z, upper or lower case), and optionally include wildcard characters '\*' or '?'.
* The function returns a tuple containing a list of tuples **(score, word)** representing valid Scrabble words constructed from the rack, sorted by score and then alphabetically, and an integer representing the total number of valid words.
* The **letters\_not\_scored** dictionary keeps track of letters replaced by wildcard characters ('\*' or '?') in each word.
* For each word in the dictionary of valid words, the function iterates through the characters to determine if the word can be formed from the rack. If a character in the word matches a character in the rack or a wildcard character, it removes that character from the temporary rack and updates the **letters\_not\_scored** dictionary accordingly.
* After checking all words, the function calculates the score for each valid word using the **score\_word** function and returns the scored words along with the total number of valid words.

## Before Publishing on git:

Stored in: Ubuntu > PythonSolutions > **PROJECTS\_mygit** > **Scrabble\_Game** folder

Backups:

Save score\_function as a .py file as well scrabble function. Clean out the comments and beautify the code.

Take a look at HW6 draft.ipynb in my github> midsdatascience… folder to see older versions specifically the permutation one.