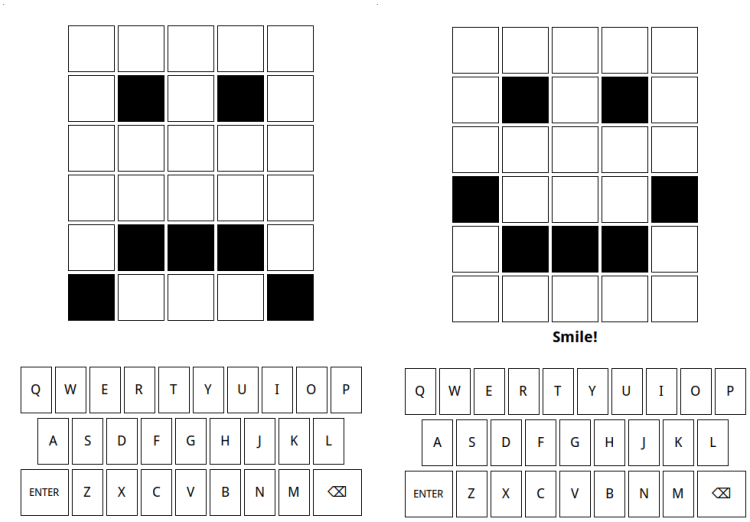
1. **Happy Wordle Graphics**

   All of the graphics of the Wordle project have been handled for you, and are made available to you through the `WordleGWindow` object type. On object of this type has already been created for you in the Wordle project (and in `Prob3.py` here) and given the name `gw`. Whenever you want to change the state of the graphics, or to get information back from the graphics, you will need to interact with this object through one of the many different methods documented in the Wordle Project Guide. Since many of these method interact with particular squares on the Wordle grid, you will need to frequently indicate which square you are interacting with by providing a row index and column index, each of which starts at 0.

   To get you some experience working with these methods, your first task in this problem is to color the squares of the Wordle grid such that they create a sad face, as shown in Figure 1a below. You can just use the string `"black"` for the color of the squares, or you can choose your own color.



(a) A frowning face in Wordle (b) A happy face in Wordle!

Figure 1: Wordle can be an emotional game at times...

   Once you have this looking nice, add code to the `enter_action` function to change the frowing face to a smiley face! The `enter_action` function is run whenever the `Enter` or `Return` key is pressed, and so you should see your frowing Wordle face change to a smiling face when you press enter! Note that you don't need to change *all* of the squares to turn that frown upside down, so just change the necessary squares so that the new image looks like Figure 1b.

2. **Coloring Wordle squares** The most difficult part of the Wordle project is Milestone 3, in which you must decide what color to apply to each of the squares in the guessed word. This turns out to be non-trivial, owing largely to the possibility of words having multiple of the same letter. Here, your task is just to work out an algorithm to correctly assign the color, not to write any actual code. Writing the algorithm in a pseudocode or diagram form would be sufficient, whichever works best for your brain to understand what is happening.

   As you design the algorithm, you should keep the following ideas in mind:

   - You need to determine all of the squares that should be colored green before assigning any of the yellow squares. Why is this necessary? What might happen for certain guess and hidden word combinations if you didn't do this, and instead tried to assign each letter a color, one by one?
   - You will need to keep separate track of what letters have already been matched in a guess. Again, why is this necessary. What would happen for certain pairings if you always compared to the original hidden word?

   When you go to implement this second condition (which you don't need to do as part of just designing the algorithm), there are a few approaches that may prove useful:

   1. Create a variable named something like `unmatched` that contains all the letters of the hidden word that have yet to be matched. This should be set equal to the hidden word at the start of each guess.
   2. When you assign a color to a letter, you need to remove that letter from the `unmatched string`. Remember that subtraction is not defined for strings! Here you have some options:
      (a) Write a small function that loops through the letters of `unmatched` and concatenates them to a new string, except for the letter that was just colored. You could entirely omit this character or instead concatenate something like an underscore or space instead to show the missing letter. Then return that new string and assign it back to `unmatched`
      (b) Use the string method `replace` to replace the colored letter with another symbol (like a space, underscore, or empty string). By default `replace` will replace *all* the matched letters, so you should provide a third argument which is a 1, and indicates that you want to replace at most 1 instance of the pattern: `unmatched.replace(letter, "_", 1)`

3. **Generating Passwords** We have all been faced with the situation of needing to come up with a new password, only to have the program inform us that our password fails one (or more) of a myriad of checks. Here your task will be to write a password generator which, given a string of possible characters and a desired length, will randomly create a password using the given characters.

   Write a function called `generate_password(|length|, |chars|)`, where the length is the number of characters you want the password to have, and `chars` is the string of possible characters that can be used to form the password. Your function should return the password once it has been created. You may find it useful to review the functions that the `random` library makes available to you. As with most programming problems, there are several approaches that you could take.

   When working with different types of strings, it can sometimes be useful to import the `string` library. While this library doesn't define any new data types (the string type is defined by default in Python), it does export several constant strings that might be of use in a variety of cases:

   - `ascii_letters`: All the lowercase and uppercase English letters
   - `digits`: All the numeric digits, 0-9
   - `punctuation`: All the punctuation symbols

   Using these, combinations of these, or subsets of these can be an easy way to generate passwords that have different properties.

   Done early? Write your function to also ensure that the random password meets certain other criteria. Maybe possessing at least 1 lowercase, 1 uppercase, 1 number, and 1 symbol.