Project 2: Word Search

Purpose

To practice string operations (not list) and decomposing a problem into functional units.

Description

Download a zip file containing test inputs and example outputs from Canvas.

In this project, you will implement a program that locates words in word search puzzles where all puzzles are 100 characters in size. A sample run of the program is shown below.

```
$ python3 wordsearch.py < test1.in

      WAQHGTTWEE
      CBMIVQQELS
      AZXWKWIIIL
      LDWLFXPIPV
      PONDTMVAMN
      OEDSOYQGOB
      LGQCKGMMCT
      YCSLOACUZM
      XVDMGSXCYZ
      UUIUNIXFNU

      UNIX: (FORWARD) row: 9 column: 3
      CALPOLY: (DOWN) row: 1 column: 0
      GCC: word not found
      SLO: (FORWARD) row: 7 column: 2
      COMPILE: (UP) row: 6 column: 8
      VIM: (BACKWARD) row: 1 column: 4
      TEST: word not found
```

Words can appear in the puzzle forward, backward, upward, and downward, but only once per puzzle. You will not need to check diagonals. You may use the `Python built-in str find` function on strings, which returns the index of the beginning of a given word located in a given string (or `-1` if the word is not present). For example, `"UUIUNIXFNU".find("UNIX")` returns `3` because the first character of `UNIX` starts at index `3` of the string. However, `"UUIUNIXFNU".find("SLO")` returns `-1` since `SLO` is not contained in the string.

Dimensionality Conversion

While the given puzzle is a 100-character string (a one-dimensional sequence), the process of finding the rows and columns of words requires operating in two dimensions. To do so, you must convert a given arbitrary index of the string into two values - one for the row and one for the column - which can be done using simple arithmetic operations.

0	1	2	3	4	5	6	7	8
A	B	C	G	I	T	X	Y	Z

One-Dimensional Character Sequence

	0	1	2
0	A	B	C
1	G	I	T
2	X	Y	Z

Two-Dimensional
Character Sequence

In the diagrams above, the word being searched (`GIT`) is highlighted in yellow, with the first letter of the word highlighted in green. Given this 9-character string, calling `"ABCGITXYZ".find("GIT")` evaluates to `3`, the index of the `G`. However, in order to report the row and column of this character, this 3 must be converted to two values: `1` for the row and `0` for the column.

The resulting output of this example would be:

```
    GIT: (FORWARD) row: 1 column: 0
```

Implementation

You may not use lists, list slicing operations, the `split` function, nor any language features and functions not yet discussed in the lecture for this assignment.

Input

Your program must get the following two inputs from the user:

- The first input is a line of text containing 100 characters. This line might have a trailing newline character. The characters are the puzzle.
- The second input is a line of text containing space-separated words. This line might have a trailing newline character. These are the words to be searched for in the puzzle.

Output

Your program must print the following text:

- The given puzzle as a 10x10 grid of characters. Hence, a row is 10-character long, i.e. row_len=10, and a column is also 10-character long.
- The result of searching for each word, specifying its direction, row, and column if found or a message indicating that it was not found

Minimum Required Program Structure

`reverse_string(string:str)->str`

- Returns the reverse of the input string

`transpose_string(string:str, row_len:int)->str`

- Returns a transposition of the input string, assuming `row_len` characters per row
  - Transposing a two-dimensional grid means converting its rows to columns and its columns to rows
  - Since strings are one-dimensional, the result will be a string with its characters shifted around
  - Hint: use two for loops, one nested in the other loop. Let the outer loop iterate over columns and the inner loop iterate over rows. Covert the two coordinates, row & column, to a linear position in the puzzle string.
- For example, `"ABCGITXYZ"` transposes to `"AGXBIYCTZ"`

`find_word(puzzle:str, word:str, row_len:int)->str`

- Searches the puzzle for the given word (in any direction)
  - This function may call other functions that search in a specific direction
    - to search backward, reverse the puzzle string before using find(). You have to convert the returned position in the reversed puzzle string back to the position in the original puzzle string.

        - to search down, transpose the puzzle string before using find(). You have to convert the returned position in the transposed puzzle string back to the position in the original puzzle string.
        - to search up, you can do so by combining the two techniques described above. Do not forget to convert the position back to the position in the original puzzle string.
    - You may use find() built-in string method to find the first occurrence of the word in the puzzle string. e.g. puzzle.find(word)
        - The find() method finds the first occurrence of the specified value and returns its position in the string. The position of the first character in a string is 0.
        - The find() method returns -1 if the value is not found.
- Returns a string containing the search result to be printed in `the main()`
    - In the example above, this function would return:
      `"GIT: (FORWARD) row: 1 column: 0"`

`main()`

- Use the `input` function to read in the puzzle and words to find (without `split`)
    - Recall that each call to `input()` reads one line from the user input.
    - Use `strip()` to remove trailing newline characters. e.g. puzzle.strip()
- Display the puzzle, one row per line (this step may be done in another function)
- Iterate through the words string, searching for each word by calling a function/functions
    - Use the fact that each word to search for is delimited by a space
    - **YOU MAY NOT USE split() NOR YOU MAY PUT WORDS IN A LIST.**
    - Ensure the order of the words printed matches the test files

Testing

Each puzzle can be found in a separate file:

```
test1.in, test2.in, test3.in
```

Your program should be run using (replace # with a number between 1 and 3):

```
python3 wordsearch.py < test#.in > my_test#_out.txt
```

You should compare your output with the corresponding output files using `diff -wB`:

```
e.g. diff -wB test1.out my_test1_out.txt
```

If there are no discrepancies between two files, you will see nothing on the screen.

You are required to write at least 3 tests for each function that you create (except `main`). Create wordsearch_tests.py and put your test code in the file. As done previously, tests are written using `assert` statements. Since we are emphasizing test-driven development, you should write tests for each function first. In doing so, you will have a better understanding as to what the functions take as input and produce as output, which makes writing the function definitions easier.

YOU MAY SUBMIT YOUR wordsearch.py TO Gradzilla TO TEST THE REQUIRED FUNCTIONS EXCEPT FOR main().

Submission

Demo your program to the instructor or one of TAs.
Submit `wordsearch.py, wordsearch_tests.py, and the outputs from the program (my_test1_out.txt, my_test2_out.txt, and my_test3_out.txt)` to Canvas.