CS 120 (Spring 22): Introduction to Computer Programming II

Short Project #2 - List Strings in a Grid

due at 5pm, Thu 27 Jan 2022

NOTE: The `itertools` and `copy` libraries in Python are **banned** in this class. Could you use them for this project? I don't know, maybe. But let's not try.

1 Overview

Write a program named `grid_scan.py` .

In this project, you will practice with reading data from a file, storing it into a 2D grid, and then searching through it. In the Long Project, you will be implementing a Word Search, and this project should help you learn some tricks which will make the Word Search easier.

For this program, you will read an input file which is a simple grid of letters; it is the same format as the Word Search problem, except that after the blank line, this project will **not** have a list of words to search for.

You get the name of the file to read from the user; after you read the file, you will then read a single command from the user, which tells you what you must print out. There are several possible commands:

-
  ```
  E x y count
  N x y count
  SW x y count
  ```

  These commands tell you to search the grid, in a certain direction. To simplify, I only require that you support these three directions; in the Long problem, you will have to support all 8.

  The `x,y,count` parameters are all integers (you can assume that there will be exactly the right number of parameters, and they are all integers). (x, y) tells you the location to start your search; $(0, 0)$ is the lower-left corner of the grid. `count` tells you how many steps to go in that direction.

  If you can't read `count` many characters (because it would fall off of the grid), then print an error message (see the testcases for the format). However, if you can successfully read the whole string, then print out the string you find, followed by the grid again. But, when you print the grid, mark the string you found by changing all of the characters in that string to periods.

- (other commands removed, at the suggestion of the TAs)

2 How to Search

There are a number of ways that you might search the grid. For some of the directions, it seems like it would be easiest to simply slice the various arrays. However, that strategy won't work when going diagonally.

To extract a string from the grid, the trick is to "walk the grid," in the direction that you want to go. Let's assume that we have the variables `start_x,start_y` representing the start position of our search. We can now build a loop, like this:

```
for i in range(how_many_characters_do_we_want_to_read):
    x = start_x + i
    y = start_y
    ... do something or other with (x,y) ...
```

The example loop above will move us in the **positive x** direction: each iteration of the loop we are one step further to the right, and the y value stays the same the whole time. However, we have to consider that there might be a problem; what if we fall off of the grid? Let's add some error checking:

```
for i in range(how_many_characters_do_we_want_to_read):
    x = start_x + i
    y = start_y

    # does (x,y) fall off the grid?
    if x is too small or x is too large or y is too small or y is too large:
        ... do some error handling here.  No string can be read! ...

    ... do something or other with (x,y) ...
```

Of course, if we are moving in the positive x direction, x can never get too small, and y will never change. But let's plan ahead - if you have all 4 checks, then this code would hardly have to change at all if we were moving in another direction!

In fact, while it's not required, I **highly recommend** that you do all of your searching with a **single function.** Ask yourself: what is the difference between searching East, and searching West? Between West, and North? Between North and Northeast? Could you combine all of these searches into a single function, with a single loop inside it?

If you simplify this far enough, you'll find that adding support for all 8 directions is easy - you just pass different parameters to the function that you've already written.

3 Data Structure Hints

While I don't have any requirement about how you store you data internally, I have some advice for you. Think ahead about what your program is going to have to do, and you will find that it will be much easier to write the program.

Your first idea will probably be to store the grid as an array of strings: one string per line in the input. That's easy to create, and it's not a terrible idea. It actually has the nice feature that you can read a single character very quickly, like this: `grid[row][column]`. It's a 2D grid, and you can access it with two array-index operations: pretty good!

But if you plan ahead, you'll notice that you will need to edit the grid later on in the program. Strings in Python are immuatable, and so editing them isn't easy - you have to muck about with slicing and concatenation. No fun! But arrays, of course, are mutable - so what if you had an array of arrays, with exactly one character per cell? That would still allow you to do array indexing (good), and also allow you to modify the grid (even better).

The next thing you'll notice is that you are used to using (x, y) to represent coordinates in a plane, but the indices are reversed in your data. That is, the first index is `[row]` and the second one is `[column]`. Wouldn't it be even easier if we could access our data as `grid[x][y]` ?

Moreover, you will also notice, when you read the data, that the first row (and thus, element `[0]` in the outer array) is the **top** of the grid. But Russ has required that $(0, 0)$ be at the **bottom** left. Wouldn't it be nice to swap the y coordinate, so that the bottom row of the grid was $y = 0$?

It turns out its not too hard to convert your data. I would recommend (not require) that you read the data from the file as a bunch of strings, and then pass it to a function, which re-arranges the data in a more handy format:

```
def this_function_will_build_grid_in_the_nicer_format(original_grid):
    retval = []

    for x in range(????):
        this_column = []

        for y in range(????):
            # we are looking for (x,y) in the *better* grid format.  Can
            # we look it up in the old format?  Add it to the column here.
            this_column.append(????)

        # we've built an entire column, for some value of x.  Add it to
        # the entire grid
        retval.append(this_column)

    return retval
```

4 Input File Format

See the long problem. The only difference in the input file is that, for this problem, the grid will not be followed by a list of words to search; there will simply be a blank line, to mark the end of the grid.

4.1 Input File Error Handling

In the Long project, you will need to check to see if the input file exists. However, for this project, you can simply assume that it does. Don't worry about handing the missing-file case.

Also, to simplify your program, you don't need to print a prompt: simply read the filename from the user, and then read the command.

5 Turning in Your Solution

You must turn in your code using GradeScope.