CS 120 (Spring 20): Introduction to Computer Programming II

Short Project #3

due at 5pm, Tue 4 Feb 2020

1 Overview

How do we represent a rectangular grid in our programs? It turns out that there are a bunch of different ways to do it. Most of them use nested lists, but not all of them - and even with nested lists, there are several interesting questions you have to answer.

In this Short Problem, you are going to implement the same 3 functions several times over. Each time, the functions will do the same basic tasks, but different versions of the same function will need to be different, because the data structure we're using will be different.

2 What You Must Represent

In all cases, we'll be implementing data structures which represent the exact same thing: a rectangular grid of characters. The grid will always be a rectangle (not necessarily a square). The smallest grid you'll have to support will be 3x3. However, there's no limit to how large it can be, in either dimension.

The grid will always hold exactly one character in each space; any printable character is legal (including space), **except** for a newline.

3 Required Functions

For each required data structure, you will implement the following three functions (details below):

- `build_rect(wid,hei)` - builds a rectagular grid of the required size. Fills up the grid with characters, as described below. Returns the newly-created grid.
- `print_rect(data)` - prints out the grid to the screen.
- `update(data, x,y, c)` - updates one of the slots in the data structure, replacing the character there. For nearly all of the data structures, this will update the grid **in place** - meaning that it changes the object it's been given, and returns nothing. But for one, it will be required to build a new data structure from scratch, and return that.

4 Data Structures, and Files

You will create several different Python files. Each file will contain the functions for a single data structure. Each file will implement the same three functions.

You will be required to implement the following data structures (details below):

- A list of lists, where the inner lists are the columns (a "list of columns"), where $y = 0$ is the top of the grid.

  Place the code for this data structure in the file `list_of_cols_0_on_top.py`

- A list of lists, where the inner lists are the columns (a "list of columns"), where $y = 0$ is the **bottom** of the grid.

  Place the code for this data structure in the file `list_of_cols_0_on_bottom.py`

- A list of lists, where the inner lists are the **rows** (a "list of rows"), where $y = 0$ is the top of the grid.

  Place the code for this data structure in the file `list_of_rows_0_on_top.py`

- A list of lists, where the inner lists are the rows (a "list of rows"), where $y = 0$ is the **bottom** of the grid.

  Place the code for this data structure in the file `list_of_rows_0_on_bottompy`

- A list of strings, where each string represents a single row of the grid, and $y = 0$ is the top of the grid

  Place the code for this data structure in the file `list_of_strings.py`

- A single string, with newlines mixed in, so that, when printed, produces a grid.

  Place the code for this data structure in the file `one_big_string.py`

- A dictionary, where the keys are (x, y) tuples, and the values are the characters. In this, $y = 0$ is the top of the grid.

  Place the code for this data structure in the file `dictionary_grid.py`

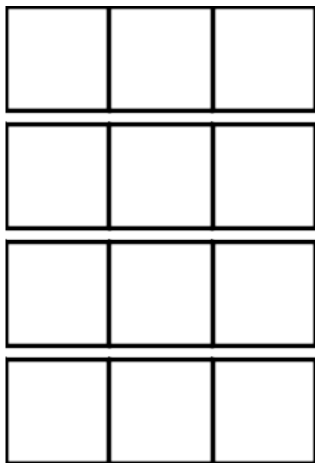
5 Lists of Lists

Several of the required data structures are lists of lists. We already know that we can build a list of lists. For instance, the following code creates a list of length 4, where each element is a list of length 3:

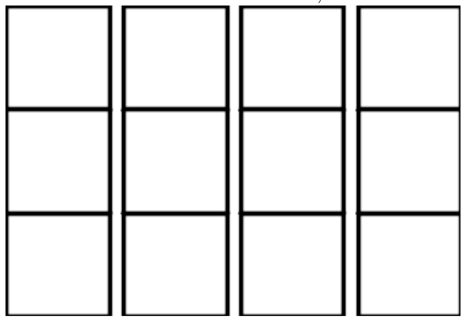
```
grid = [ [1,2,3], [4,5,6], [7,8,9], [0,1,2] ]
```

(Of course, these lists hold integers, not characters, but you've got the idea.)

But here's the question: is the list-of-lists above a 3x4 grid, or a 4x3 grid? It turns out that both are totally reasonable options. If you intepret the code above as a "list of rows," then the data structure looks like this:


and we would say that the size of the grid is 3x4 (width 3, height 4).

On the other hand, another totally reasonable interpretation would be to view this as a “list of columns, which would look like this:



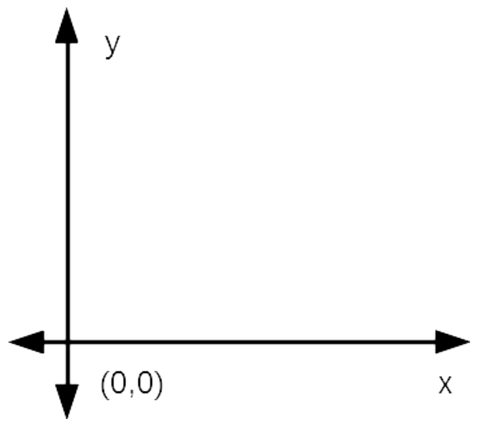
So what does this mean for you? It means that, when you build (or access) these data structures, you will need to be careful about how you use your x and y variables:

- In a list of rows, the first index will tell us what row we are on (that is, the y coordinate), and the second index will be the x coordinate.
- In a list of columns, the two are reversed: the x coordinate will be the first index, and the y coordinate will be the second.

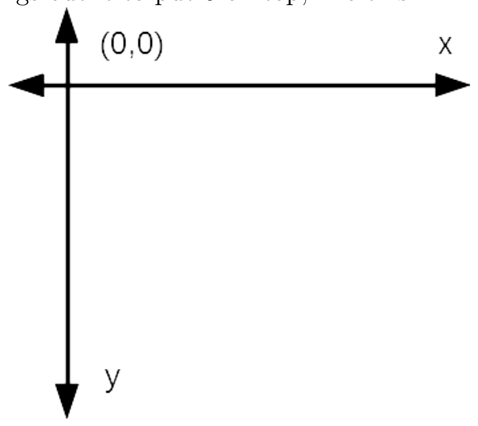
6 0 on the Top, or the Bottom?

We’ve been talking about (x, y) coordinates. But it’s important to think about how we’re going to use them. For the x coordinate, that’s pretty easy: 0 is always on the left side. But what about y?

I’m sure you remember, from your math classes, that mathematicians regularly draw their coordinates like this:



However, this can be annoying for programmers, sometimes - because, if we're printing something to the screen, we print from the top to the bottom. So, unless we're pretty tricky about our loops, the natural way for us to print things out it to put 0 on top, like this:



Which is correct?

The truth is, they both are. Either one works just fine. If you want to make your loops easy, you tend to write 0-on-top code. But that often makes it a little confusing, since "going up" in the grid means going to smaller values of y.

Personally, I prefer the 0-on-bottom style, most of the time. But we're going to practice both.

7 Data Details

7.1 Lists of Lists

The first four data structures are all lists of lists. Two of them are lists of columns, and two are lists of rows; two of them have 0 on top, while two have

0 on bottom.

In all 4 of these, you can assume (without checking) that all of the sub-lists have the same length (if that wasn't true, it wouldn't be a rectangle). You can also assume (without checking) that every element, in every sub-list, is a single character.

How does this look in real code? Consider the list-of-columns version, with 0 on bottom. Suppose that we wanted to encode the following grid:

```
a b c
d e f
g h i
```

Since this is a list of columsns, one of the sub-lists is ['g', 'd', 'a']. And since 0 is on the bottom, the 'g' must be element [0] of that sub-list. So, the overall list-of-lists is this:

```
grid = [ ['g', 'd', 'a'], ['h', 'e', 'b'], ['i', 'f', 'c'] ]
```

I know this looks strange! But it turns out to be really easy to use. For instance, what are the coordinates of the i character (bottom right)? Math would tell us that the coordinates are $(2, 0)$. And what indices do we use to access it? The same!

```
bottom_right_char = grid[2][0]
```

7.2 List of Strings

The list-of-strings is basically the same as a list-of-rows, but where we have strings, instead of sub-lists. As with the lists-of-lists, you can assume (without checking) that all of the strings are the same length.

Since we would normally only use this if we are planning to print it over and over, you will **only** implement a 0-on-top version. You will **not** implement a 0-on-bottom version.

The example grid above would be encoded as:

```
grid = ["abc", "def", "ghi"]
```

7.3 Single String

This works more or less like the list-of-strings, except that, instead of a list, it actually is one gigantic string, with newlines in the middle. You must have a newline at the end of every row (including the last one).

The grid above would be encoded as:

```
grid = "abc\ndef\nghi\n"
```

7.4 Dictionary

In the dictionary, your keys are coordinate pairs, and the values are the characters. Obviously, it has no "order."

The grid above would be encode as:

```
grid = { (0,0):'a', (1,0):'b', (2,0):'c', (0,1):'d', ...... }
```

8 Required Functions

Implement the following functions for all seven data structures - one set of functions per file. (Helper functions are allowed but not required.)

8.1 `build_rect(wid,hei)`

This builds a new grid, of the appropriate size. `wid` is the width of the grid - that is, the x dimension - while `hei` is the height (the y dimension).

You **must** assert that the width and height are both at least 3.

The grid that you build must be filled as follows:

- Put a space in each of the 4 corners.
- Along the top edge (except the corners), place the charcter `T` in each slot. Along the bottom, put `B`. Along the left edge, put `L`; put `R` on the right.
- All other spaces (the "interior") must be filled with periods.

For example, if I call `build_rect(4,6)`, you must return a data structure which represents the following grid:

```
 TT
L..R
L..R
L..R
L..R
 BB
```

8.2 `print_rect(data)`

Print out the grid to the screen. Your only parameter is a data structure which you preiously returned from `build_rect()` (or an equivalent, which the testcase builds for you).

You probably will need to know the width and height of the grid in order to print the data correctly. For lists, this is easy: you should use the `len()` function. However, for some of hte data structures, the width and height are not so obvious. In that case, you may need to write some code which figures out the width and height.

NOTE: You may find yourself wanting to print thigns out one character at a time - but Python normally prints out a newline after printing your stuff. Can we fix this? Yes! Simply use the `end` paramter to print():

```
print('a', end="")
```

8.3 `update(data, x,y, c)`

Changes one character in the grid. The (x, y) coordinates are given - along with the character you want to drop there.

For all data structures **except** "one big string," this function will always modify the object it's been given, and it returns nothing. However, since strings are immutable, we cannot modify the string in place. Instead, we slice it, reassemble it, and then return the brand-new string as the new value of the grid, after the modification.

WARNING: Pay attention to whether 0 is on the top, or on the bottom in your `update()` functions. For example, the call `update(grid, 2,0, 'x')` might modify a character on the bottom row in one data strucutre, while it modifies something on the top row in others.

9 Turning in Your Solution

You must turn in your code using GradeScope.

10 Acknowledgements

Thanks to Saumya Debray for many resources that I used and adapted for this class.