Homework 3

Due 4:00pm, Wednesday, February 20, 2019

Installation and Handin 1

Homework Setup. For each homework assignment, there may be support files that you will need to complete the assignment. These can be copied to your home directory by using the `cs4_install` command in a Brown CS Terminal window. For this homework type

```
cs4_install hw03
```

There should now be a `hw03` folder within your homeworks directory. Using Terminal, you can move into the `hw03` folder with the cd command:

```
cd ~/course/cs0040/homeworks/hw03
```

Homework hand-in. Be sure to turn in all the files requested and that they are named exactly as specified, including spelling and case. When you're ready to submit the files, run:

```
cs4_handin hw03
```

from a Brown CS Terminal window from your `~/course/cs0040/homeworks/hw03` directory. The entire contents of your `~/course/cs0040/homeworks/hw03` directory will be handed in. Check for a confirmation email to ensure that your assignment was correctly submitted using the `cs4_handin` command. You can resubmit this assignment using the `cs4_handin` command at any time, but be careful, as only your most recent submission with be graded.

Part I: Mutability (40)

Babies and Puppies have united because they have been united in a singular struggle: they know their own names but no one else seems to! They keep getting called by these "nicknames" by adults but have no idea who they're referring to, and they need your help to understand.

Include your answers and code to this part of the homework in `hw03_1.py`. Remember to put any written responses inside of a triple quoted string.

Problem 3.1a) Tracing References

Consider the following Python program:

```
def foo1(values, x):
    for i in range(len(values)):
        values[i] = i
    x += 1

a = [4, 5, 6]
b = a[:]
c = a
x = 5

foo1(c, x)

print('a =', a)
print('b =', b)
print('c =', c)
print('x =', x)
```

Trace this code in Python Tutor, then answer the following questions:

1. Before the function call `foo1(c, x)` is made:
    - How many distinct lists are there?
    - Which variables refer to the same list?
2. During the execution of the function:
    - What value or values does the parameter *values* take? (inside of foo1)
    - What values does the variable *i* take on?

- When the function call is about to complete, what return value does Python Tutor specify? Why does this make sense?

3. After the function call returns:
   - What output does the program produce?
   - Why have the values of some of the variables changed, while others have not?

Problem 3.1b) Functions for 2-D Lists

In this problem we'll work with 2-D lists of single-digit integers.

We have given you a function called `create_grid(height, width)` that creates and returns a 2-D list of `height` rows and `width` columns in which all of the cells–i.e., all of the elements of the sublists–have a value of `0`.

Additionally, we've also provided a function called `print_grid(grid)` that returns the grid in a well-structured format, printing each row in a new separate line.

For example:

```
>>> grid = create_grid(3,5)
>>> print_grid(grid)
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Write a function called `mod_grid(grid, n)` which takes a 2-D list of integers (`grid`) and an integer `n`, and replaces each value in the grid with the modulus(%) of that value with `n`.
For example:

```
>>> mygrid = [[5, 4, 3, 2], [3, 6, 9, 1], [4, 7, 2, 9]]
>>> print_grid(mygrid)
5 4 3 2
3 6 9 1
4 7 2 9
>>> mod_grid(mygrid, 2)
>>> print_grid(mygrid)
1 0 1 0
1 0 1 1
0 1 0 1
```

Notice that `mod_grid()` does ***not*** return a value. It doesn't need to! That's because its parameter `grid` gets a copy of the *reference* to the original 2-D list, and thus any changes that it makes to the internals of that list will still be visible after the function returns.

We will now **write a new version of this function** called `mod_grid_new(grid, n)` that leaves the original 2-D list unchanged. Instead it should create and return a new 2-D list containing the results of taking the modulus of each list value with `n`. The internals of `grid` should ***not*** be changed by this function.

Hint: Before the loop, create a new 2-D list with the same dimensions as the original one. To do this, consider calling the `create_grid` function to do the work for you! Next, modify the loop so that it stores the result of each modulus calculation in the new 2-D list, rather than in the original one.

For example:

```
>>> mygrid = [[5, 4, 3, 2], [3, 6, 9, 1], [4, 7, 2, 9]]
>>> result = mod_grid_new(mygrid, 2)
>>> print_grid(result)
1 0 1 0
1 0 1 1
0 1 0 1
>>> print_grid(mygrid)
5 4 3 2
3 6 9 1
4 7 2 9
```

Problem 3.1c) More 2-D List Functions

This problem involves writing some more functions that create and manipulate two-dimensional lists of integers. These functions or ones similar to them, will be used in the next problem to implement John Conway's *Game of Life*.

Write a function called `diagonal_grid(height, width)` that creates and returns a 2-D list of `height` rows and `width` columns. Additionally, all cells on the diagonal - i.e. the cells whose row and column indices are the same, should contain the value `1`. All other cells should have a value of `0` instead.

Hint: Since there will likely be more `0`'s than `1`'s, consider using the `create_grid(height, width)` function to create a grid of `0`'s, and then modify the appropriate cells to `1`.

For example:

```
>>> grid = diagonal_grid(6, 8)
>>> print_grid(grid)
10000000
01000000
00100000
00010000
00001000
00000100
```

Write a function called `inner_grid(height, width)` that creates and returns a 2-D list of `height` rows and `width` columns in which the "inner" cells are all `1` and the cells on the outer border are all `0`.

Hint: Consider modifying the ranges used by your loops so that they loop over only the inner cells.

For example:

```
>>> grid = inner_grid(5, 5)
>>> print_grid(grid)
00000
01110
01110
01110
00000
```

Write a function called `random_grid(height, width)` that creates and returns a 2-D list of `height` rows and `width` columns in which the inner cells are randomly assigned either `0` or `1` (with equal probability), but the cells on the outer border are all `0`.

Hint: You will need to use the call `random.choice([0, 1])`, which will return either a `0` or a `1`.

Note: You do not need to test this function.

For example (although the actual values of the inner cells will vary):

```
>>> grid = random_grid(10, 10)
>>> print_grid(grid)
0000000000
0100000110
0001111100
0101011110
0000111000
0010101010
0010111010
0011010110
0110001000
0000000000
```

As we've seen in lecture, assigning a list variable does *not* actually copy the list. To see an example of this, try the following commands from the Shell:

```
>>> grid1 = create_grid(2, 2)
>>> grid2 = grid1          # copy grid1 into grid2
>>> print_grid(grid2)
00
00
>>> grid1[0][0] = 1
>>> print_grid(grid1)
10
00
>>> print_grid(grid2)
10
00
```

Notice how changing `grid1` also changes `grid2`! That's because the assignment `grid2 = grid1` did ***not*** copy the list represented by grid1. Instead it copied the ***reference*** to the list. Thus, `grid1` and `grid2` both refer to the same list!
To avoid this problem, **write a function** called `copy(grid)` that creates and returns a ***deep copy*** of `grid`–a new, separate 2-D list that has the same dimensions and cell values as `grid`. Note that you ***cannot*** just perform a full slice on `grid` (e.g., `grid[:]`), because you would still end up with copies of the references to the rows. Instead, you should do the following:

- Use `create_grid` to create a new 2-D list with the same dimensions as `grid`, and assign it to an appropriately named variable. Remember that `len(grid)` will give you the number of rows in `grid`, and `len(grid[0])` will give you the number of columns.
- Use nested loops to copy the individual values from the cells of `grid` into the cells of your newly created grid.
- Make sure to return the newly created grid and *not* the original one!

For example:

```
>>> grid1 = diagonal_grid(3, 3)
>>> print_grid(grid1)
100
010
001
>>> grid2 = copy(grid1)   # should get a deep copy of grid1
>>> print_grid(grid2)
100
010
001
>>> grid1[0][1] = 1
>>> print_grid(grid1)     # should see an extra 1 at [0][1]
110
010
001
>>> print_grid(grid2)     # should not see an extra 1
100
010
001
```

Write a function called `invert(grid)` that takes an existing 2-D list of `0`'s and `1`'s and inverts it – changing all `0` values to `1`, and changing all `1` values to `0`.

Important notes:

- This function should ***not*** create and return a new 2-D list. Rather, it should modify the internals of the existing list.
- There should ***not*** be a return statement because its parameter `grid` gets a copy of the reference to the original 2-D list, and thus any changes that it makes to the internals of that list will still be visible after the function returns.
- You may assume all values in the grid are either 0 or 1 (no bad inputs)

- Even though this function does not return a value, you are still expected to test it like you would with functions that do return values. Think about how you could set things up before the assert statement so that your tests work.

For example:

```
>>> grid = diagonal_grid(5, 5)
>>> print_grid(grid)
10000
01000
00100
00010
00001
>>> invert(grid)
>>> print_grid(grid)
01111
10111
11011
11101
11110
```

To reinforce your understanding of references, let's walk through the following example code:

```
>>> grid1 = inner_grid(5, 5)
>>> print_grid(grid1)
00000
01110
01110
01110
00000
>>> grid2 = grid1
>>> grid3 = grid1[:]
>>> invert(grid1)
>>> print_grid(grid1)
11111
10001
10001
10001
11111
```

Problem 3.1d) Questions

Answer the following questions where specified in the file:

1. As you can see above, the value of `grid1` has been changed. What about `grid2` and `grid3`?
2. *Before entering the statements below, see if you can predict what has happened to `grid2` and `grid3`.* If we print them, will we see the original grid or an inverted one?
3. Test your understanding by first entering the following:
   `>>> print_grid(grid2)`
   What do you see? Why does this make sense?
4. Now enter the following:
   `>>> print_grid(grid3)`
   What do you see? Why does this make sense?

Part II: Conway's Game of Life (60)

Include your answers and code to this part of the homework in `hw03_2.py`.

IMPORTANT: We have included an import statement at the top of `hw03_2.py` that imports all of the functions from your `hw03_1.py` file. Therefore, you will be able to call any of the functions that you wrote previously.

Babies have decided that the only way to prove that they are better than puppies is to figure out how to get even more babies! They heard about the Game of Life, and want you to help them figure out how it works. In this part, you'll first get warmed up and then implement the Game of Life so the babies can emerge victorious.

Problem 3.2a) Warm-up

Write a function called `has_left_right_neighbor(grid)` that takes an existing generation of cells (the 2-D list `grid`), and that creates and returns a `new_grid` with the same dimensions as `grid`, but with cell values determined as follows:

- If an inner cell in `grid` has an alive left neighbor or an alive right neighbor or both, the corresponding cell in the `new_grid` should be `1`.
- If an inner cell in `grid` has neither an alive left neighbor nor an alive right neighbor, the corresponding cell in the `new_grid` should be `0`.
- The cells on the outer boundary of `grid` should be left alone.

For example:

```
>>> grid1 = diagonal_grid(6, 6)
>>> print_grid(grid1)
100000
010000
001000
000100
000010
000001
>>> grid2 = has_left_right_neighbor(grid1)
>>> print_grid(grid2)
100000
001000
010100
001010
000100
000001
```

Notice how the cells on the outer boundary are the same as the corresponding cells in the original grid, but that the inner cells have been set to `0` or `1` based on the rules mentioned above.

Write a function called `count_neighbors(cellr, cellc, grid)` that returns the number of alive neighbors of the cell at position `[cellr][cellc]` in the specified grid. You may assume that the indices `cellr` and `cellc` will always represent one of the inner cells of grid, and thus the cell will always have eight neighbors.

For example:

```
>>> grid1 = [[0,0,0,0,0],
             [0,0,1,0,0],
             [0,0,1,0,0],
             [0,0,1,0,0],
             [0,0,0,0,0]]
>>> print_grid(grid1)
00000
00100
00100
00100
00000
```

```
>>> count_neighbors(2, 1, grid1)   # grid1[2][1] has 3 alive neighbors
3
>>> count_neighbors(2, 2, grid1)   # we don't count the cell itself!
2
>>> count_neighbors(1, 2, grid1)
1
>>> grid2 = [[0,0,0,0,0,0],
             [0,0,1,1,0,0],
             [0,1,1,1,0,0],
             [0,0,1,0,1,0],
             [0,0,1,0,1,0],
             [0,0,0,0,0,0]]
>>> count_neighbors(2, 2, grid2)   # grid2[2][2] has 5 alive neighbors
5
>>> count_neighbors(2, 3, grid2)   # so does grid2[2][3]
5
>>> count_neighbors(3, 3, grid2)   # grid2[3][3] has 6 alive neighbors
6
```

Problem 3.2b) Real Life

Finally, let's implement the rules of the Game of Life! The Game of Life was invented by John Conway, who is currently a professor of mathematics at Princeton. It's not a game in the traditional sense, but rather a grid of cells that changes over time according to a few simple rules.

At a given point in time, each cell in the grid is either "alive" (represented by a value of `1`) or "dead" (represented by a value of `0`). The neighbors of a cell are the cells that immediately surround it in the grid.

Over time, the grid is repeatedly updated according to the following five rules:

1. All cells on the outer boundary of the grid remain fixed at `0`.
2. An inner cell with fewer than 2 alive neighbors must die (because of loneliness).
3. An inner cell with more than 3 alive neighbors must die (from overcrowding).
4. An inner cell that is dead and has exactly 3 alive neighbors comes to life.
5. All other cells maintain their state.

Although these rules seem simple, they give rise to complex and interesting patterns! You can find more information and a number of interesting patterns here.

Write a function called `next_gen(grid)` that takes a 2-D list called `grid` that represents the

current generation of cells, and uses the rules of the Game of Life (mentioned previously) to create and return a new 2-D list representing the next generation of cells.

Hint: Begin by creating a copy of `grid` and call it `new_grid`. Consider using previously written functions such as `count_neighbors()`. Make sure that you count the neighbors in the current generation `grid` and not the `new_grid`. When updating a cell, make sure to change the appropriate element of `new_grid` and not the element of `grid`.

Here are two patterns that you might want to take advantage of when testing your `next_gen()` function:

1. If a 3x1 line of alive cells is isolated in the center of a 5x5 grid, the line will oscillate from vertical to horizontal and back again, as shown below:

```
>>> grid1 = [[0,0,0,0,0],
                  [0,0,1,0,0],
                  [0,0,1,0,0],
                  [0,0,1,0,0],
                  [0,0,0,0,0]]
    >>> print_grid(grid1)
    00000
    00100
    00100
    00100
    00000

>>> grid2 = next_gen(grid1)
    >>> print_grid(grid2)
    00000
    00000
    01110
    00000
    00000
>>> grid3 = next_gen(grid2)
>>> print_grid(grid3)
00000
00100
00100
00100
00000
```

and so on!

2. In a 4x4 grid, if the inner cells are all alive, they should remain alive over time:

```
>>> grid1 = [[0,0,0,0],
                  [0,1,1,0],
                   [0,1,1,0],
                   [0,0,0,0]]
      >>> print_grid(grid1)
      0000
      0110
      0110
      0000
      >>> grid2 = next_gen(grid1)
      >>> print_grid(grid2)
      0000
      0110
      0110
      0000
```

Visualizing the Game

Once your `next_gen` function is working, try running these commands in terminal:

```
python3 -i gol_graphics.py
>>> grid = random_grid(15, 15)
>>> show_graphics(grid)    # run the Game of Life in a graphics window
```

The following key presses can be used to control the simulation:

- `Enter` (or `Return`): begin/resume the simulation
- `P`: pause the simulation
- `Spacebar`: clear the grid
- `Q`: quit the simulation

When the simulation is paused, you should be able to change the state of a cell by clicking on it.
Trying other patterns: The Life Lexicon is a website that includes many examples of interesting starting patterns. They use a different representation for the cells in a grid: a dot . character for a dead cell and an O character for an alive cell. For example, here is a well-known pattern known as the Gosper glider gun:

```
........................O...........
......................O.O...........
............OO......OO............OO
...........O...O....OO............OO
OO........O.....O...OO..............
OO........O...O.OO....O.O...........
..........O.....O.......O...........
...........O...O....................
............OO......................
```

You can easily try a pattern that is specified in Life Lexicon form by using the read_pattern function. It takes 2 inputs specifying the amount of padding (i.e., the number of empty rows and empty columns) that should be added around a pattern that is entered by the user. When you call this function, it waits for you to enter a pattern in the form found in the Life Lexicon, and it converts it into a 2-D list of the form that our functions use.
For example:

```
>>> grid = read_pattern(20, 20)   # use a padding of 20 on all sides
```

```
enter the pattern:
........................O...........
......................O.O...........
............OO......OO............OO
...........O...O....OO............OO
OO........O.....O...OO..............
OO........O...O.OO....O.O...........
..........O.....O.......O...........
...........O...O....................
............OO......................
```

When you are prompted to enter the pattern, copy and paste it into the Shell window and hit `Enter`. (It's not a problem if you have extra spaces at the start of some of the lines when you paste the pattern.) You can then run the pattern graphically as indicated above.

Colors
You can also change the colors used for the alive and dead cells. For example, to invert the default color scheme you can do the following from the Shell before calling `show_graphics`:

```
>>> set_color(0, 'red')
>>> set_color(1, 'white')
```

A full list of color names is available here.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this document or have any concerns about this and any other CS4 document by posting on Piazza or filling out our anonymous feedback form.