Problem 1: The Game of Life

The Game of Life is a *cellular automaton* invented by John Conway, a mathematician from Cambridge. The game of life is not so much a "game" in the traditional sense, but rather a process that transitions over time according to a few simple rules. The process is set up as a grid of cells, each of which is "alive" or "dead" at a given point in time. At each time step, the cells live or die according to the following rules:

1. A cell that has fewer than two live neighbors dies (because of isolation)
2. A cell that has more than 3 live neighbors dies (because of overcrowding)
3. A cell that is dead and has exactly 3 live neighbors comes to life
4. All other cells maintain their state

Although these rules seem simple, they give rise to complex and interesting patterns. For more information and a number of interesting patterns see the Wikipedia article on the Game of Life.

In this lab, you will implement a Python program to run the Game of Life.

Thinking about Life

As always, it is important to break the problem down into pieces and develop the program in stages so that others can understand the code and so that you can ensure that each piece is correct before building on top of it. We will break this problem down into the following steps:

- Creating a 2d array of cells
- Displaying the board and updating it with new data
- Allowing the user to change the state of the cells
- Implementing the update rules for the "Game of Life"

Before you start, you need to develop a scheme for keeping track of your data. Basically, the data you need to maintain are the states of all of the cells in the board. To do this, you should keep track of this data in a 2D array of integer values, where `0` represents an empty (off) cell and `1` represents a live (on) cell.

First, you'll implement the basic functionality for creating 2d arrays of data, changing them, and having them evolve according to the rules of Life.

Creating an Empty 2d Board of Cells

The function in the trinket below offers a starting point for creating *one-dimensional* lists—but the same idea applies for building nested list structures arbitrarily deep.

Get started by making a copy of this trinket.

`createBoard(width, height)`

Building on this example, write a function named `createBoard(width, height)` that creates and returns a new 2D list of `height` rows and `width` columns in which all of the data elements are `0` (no graphics quite yet, just a Python list!).

Avoid re-implementing the `createOneRow` function!

Rather, use `createOneRow` inside your `createBoard` in the same way that `0` is used to accumulate individual elements in `createOneRow`. Here is a template—copy and paste this and then fill in the parts you'll need to complete it:

```
def createBoard(width, height):
    """ returns a 2d array with "height" rows and "width" cols """
    A = []
    for row in range(height):
        A += SOMETHING    # What do you need to add a whole row
here?
    return A
```

That's all you'll need! Again, the idea is to follow the example of `createOneRow`—but instead of adding a `0` each time, the function would add a whole row of `0`s, namely the result from `createOneRow`!

Test out your `createBoard` function! For example,

```
>>> A = createBoard(5, 3)
>>> A
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0,
0]]
```

Printing your 2D Board of Cells

You no doubt noticed that when Python prints a 2D list, it blithely ignores its 2D structure and flattens it out into one line (perhaps wrapping, if needed). To print your board in 2D using ASCII, copy this function into your file:

```
def printBoard(A):
    for row in A:
        line = ''
        for col in row:
            line +=
str(col)
        print line
```

This `printBoard` function bypasses Python's behavior of placing a space between items it prints by creating strings without spaces for Python to print. This is why it needs to convert the integers in the array A to strings before writing them.

Make sure your `printBoard` is working as follows:

```
>>> A = createBoard(5,
3)
>>> printBoard(A)
00000
00000
00000
```

Adding Patterns to 2D Arrays

`diagonalize(width, height)`

To get used to looping over 2d arrays of data, copy this function named `diagonalize(A)` into your file:

```
def diagonalize(width, height):
    """ creates an empty board and then modifies it
        so that it has a diagonal strip of "on"
cells.
    """
    A = createBoard(width, height)

    for row in range(height):
        for col in range(width):
            if row == col:
                A[row][col] = 1
            else:
                A[row][col] = 0

    return A
```

This function, `diagonalize`, takes in a desired width and height. It then creates an array `A` and sets `A`'s data so that it is an array whose cells are empty *except for the diagonal* where `row == col`.

Try displaying the result with

```
>>> A = diagonalize(7, 6)
>>> A
[[1, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0,
0],
[0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0,
0],
[0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 1,
0]]
>>> printBoard(A)
1000000
0100000
0010000
0001000
0000100
0000010
```

Take a moment to note the direction the diagonal is running—that indicates which way the *rows* of the board are being displayed: top-to-bottom, in this case.

Also, this example shows that the height and width do not have to be the same—though it's certainly ok if they are.

`innerCells(w, h)`

Based on the example of `diagonalize`, write a variation named `innerCells(w, h)` that returns a 2d array that has **all** live cells—with the value of `1`—***except*** for a one-cell-wide border of empty cells (with the value of `0`) around the edge of the 2d array.

For example, you might try

```
>>> A = innerCells(5,
5)
>>> printBoard(A)
00000
01110
01110
01110
00000
```

Hint: Consider the usual pair of nested loops for this one—except here, the ranges run from `1` to `height-1` and from `1` to `width-1` instead of the full extent of the board!

`randomCells(w, h)`

Next, create a function named `randomCells(w, h)` which returns an array of randomly-assigned `1`'s and `0`'s *except* that the outer edge of the array is still completely empty (all `0`'s) as in the case of `innerCells`.

Here is one of our runs:

```
>>> A = randomCells(10,
10)
>>> printBoard(A)
0000000000
0100000110
0001111100
0101011110
0000111000
0010101010
0010111010
0011010110
0110001000
0000000000
```

You might recall that `random.choice([0, 1])` will return either a `0` or a `1`. You will need to `import random` to use it!

Copying your Board of Cells

Each of the updating functions so far creates a new set of cells without regard to an old "generation" that it might depend on. Conway's game of life, on the other hand, follows a set of cells by changing one generation into the next.

To see why `copy(A)` is a crucial helper function for this process, try the following commands:

```
>>> A = createBoard(3,3)  # create a 3x3 empty board
>>> printBoard(A)         # show it
000
000
000

>>> newA = A              # create a false ("shallow")
copy
>>> printBoard(newA)      # show it
000
000
000

>>> A[1][1] = 1           # set oldA's center to 1
>>> printBoard(A)
000
010
000

>>> printBoard(newA)      # but newA changed, too—aargh!
000
010
000
```

Here we have made a "copy" of `A`, where we have named the copy `newA`. However, `newA` is simply a copy of the *reference* to the original data in `A`!

As a result, when `A`'s data changes, so does `newA`'s data, even though we never touched `newA`'s data!

The above example shows ***shallow*** copying: the copying of a *reference* to data, rather than making a full copy of all of the data.

Making a full copy of all of the data is called ***deep*** copying.

Writing `copy(A)`

For this part of the lab, write a function named `copy(A)`, which will make a ***deep*** copy of the 2d array `A`.

That is, `copy` will accept a 2d array `A` as its argument, and it will return a new 2d array of data that has the same pattern as the original array.

Remember that you can get the number of rows (the height) in `A` with `len(A)` and the number of columns (the width) in `A` with `len(A[0])`; thus, you can use the lines

```
height = len(A)
width =
len(A[0])
```

at the top of your `copy` function. Also, to make sure you return new data, use `createBoard` to get a brand new array of the same size as the argument array. The above two lines help, because now you have `width` and `height` to pass in to `createBoard`!

Hint: Use the loops from `randomCells`!

You'll need to change the assignment statement, but otherwise, those loops will be perfect.

You can make sure your `copy` function is working properly with this example:

```
>>> A = createBoard(3,3)
>>> printBoard(A)
000
000
000

>>> newA = copy(A)
>>> printBoard(newA)
000
000
000

>>> A[1][1] = 1
>>> printBoard(A)      # changes it!
000
010
000

>>> printBoard(newA)   # not
changed:
000
000
000
```

This time, `newA` has ***not*** changed just because `A` did!

`innerReverse(A)`

Copying is a very simple—and not overly interesting—way that a new "generation" of array elements might depend on a previous generation of elements.

Next you'll write a function that *changes* one generation of cells into a new generation.

To that end, write a function `innerReverse(A)` that takes an old 2d array (or "generation") and then creates a new generation of the same shape and size (either with `copy`, above, or `createBoard`).

However, the new generation should be the "opposite" of `A`'s cells everywhere except on the outer edge. In the same spirit as `innerCells`, you should make sure that the new generation's outer edge of cells are always all `0`.

However, for inner cells—those not on the edge—where `A[row][col]` is a `1`, the new array's value will be a `0`—and vice versa.

Hint: Just copy/paste and then alter your `copy(A)` function!

Try out your `innerReverse` function by displaying an example. This one uses `randomCells`:

```
>>> A = randomCells(8,
8)
>>> printBoard(A)
00000000
01011010
00110010
00000010
01111110
00101010
01111010
00000000

>>> A2 = innerReverse(A)
>>> printBoard(A2)
00000000
00100100
01001100
01111100
00000000
01010100
00000100
00000000
```

Aside: You might point out that it would be possible to simply change the old argument `A`, rather than create and return new data—this is true for simply reversing the pattern of array elements, but it is *not* true when implementing the rules of Conway's Game of Life—there, changing cells without copying would change the number of neighbors of other cells!

John Conway's Game of Life

For this step, you'll create two functions, `countNeighbors(row,col,A)` and `next_life_generation(A)`.

`countNeighbors(row, col, A)`

It will help to write a helper function, `countNeighbors(row, col, A)`, which should return *the number of live neighbors* for a cell in the board `A` at a particular `row` and `col`.

There are two basic approaches to `countNeighbors`:

1. Write two *small* for-loops to examine the nine cells **centered at** `A[row][col]`. You'll need to make sure you don't count the center as a neighbor!
2. **Or**, write eight `if` statements to check all eight possible neighbors.

Checking `countNeighbors`: Define this 5x5 array `A` and then check a few cells' neighbor counts:

```
>>> A = [ [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]]

>>> printBoard(A)
00000
00100
00100
00100
00000

>>> countNeighbors(2,1,A)
3                # correct! There are 3 live neighbors
here

>>> countNeighbors(2,2,A)
2                 # be sure not to count the cell
itself!

>>> countNeighbors(0,1,A)
1
```

`next_life_generation(A)`

Finally, you'll write `next_life_generation`, which implements the rules of life. Here is a starting signature for `next_life_generation`:

```
def next_life_generation(A):
    """ makes a copy of A and then advances one
        generation of Conway's game of life
within
        the *inner cells* of that copy.
        The outer edge always stays at 0.
    """
```

This `next_life_generation` function should accept a 2D array `A`, representing the "old" generation of cells, and it should return the *next generation* of cells, each either `0` or `1`, based on John Conway's rules for the *Game of Life*:

1. All edge cells stay zero (`0`) (but see the extra challenges for this week)
2. A cell that has fewer than two live neighbors dies (because of loneliness)
3. A cell that has more than 3 live neighbors dies (because of over-crowding)
4. A cell that is dead and has exactly 3 live neighbors comes to life
5. All other cells maintain their state

For concreteness, let's call the new generation of cells you're returning `newA` in order to contrast it with `A`.

As suggested in `innerReverse`, always keep all of the outer-edge cells empty! . This is simply a matter of limiting your loops to an appropriate range. However, it greatly simplifies the four update rules, above, because it means that you will only update the interior cells, each of which ***has a full set of eight neighbors without going out of bounds***.

Warnings/Hints: There are a few things to keep in mind:

- Only count neighbors within the old generation `A`. Change only the new generation, `newA`.
- Be sure to set *every* value of `newA` (the new data), whether or not it differs from `A`.
- A cell is **NOT** a neighbor of itself.
- A 2x2 square of cells is statically stable (if isolated)—you might try it on a small grid for testing purposes.
- A 3x1 line of cells oscillates with period 2 (if isolated)—also a good pattern to test.

Here is a set of tests to try based on the last suggestion in the list above:

```
>>> A = [ [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]]

>>> printBoard(A)
00000
00100
00100
00100
00000

>>> A2 = next_life_generation(A)
>>> printBoard(A2)
00000
00000
01110
00000
00000

>>> A3 =
next_life_generation(A2)
>>> printBoard(A3)
00000
00100
00100
00100
00000
```

and so on.

Once your Game of Life is working, look for some of the other common patterns, e.g., other statically stable forms ("rocks"), as well as oscillators ("plants") and others that will move across the screen, known as gliders ("animals/birds").

Submit Homework 9, Problem 1

20.0/20.0 points (graded)

To submit your Homework 9, Problem 1 code, you'll need to copy it from your trinket or file and paste it into the box below. After you've pasted your code below, click the "Check" button.

IMPORTANT: Make sure that there aren't spaces at the beginning of your code, and that you copied all of the characters. If there are extra spaces or you are missing spaces, our server won't be able to run your code and we won't be able to give you any of the points you deserve for your hard work.

```
import random
```

```
def createOneRow(width):
    """ returns one row of zeros of width "width"...
         You should use this in your
         createBoard(width, height) function """
    row = []
    for col in range(width):
        row += [0]
    return row
```

```
def createBoard(width, height):
    """ returns a 2d array with "height" rows and "width" cols """
    A = []
    for row in range(height):
        A += [createOneRow(width)]
    return A
```

Press ESC then TAB or click outside of the code editor to exit
correct

correct

Test results
CORRECT See full outputSee full output

You have used 1 of 3 attempts Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.