COMSC-205PY FALL 2020 ASSIGNMENT 1

Due October 29th at 9:00am Eastern Time

Submission checklist

Your final submission should have the following file, uploaded to moodle:

- **Assignment1.py**

At the top of each file, clearly state:
- Your name and the name of the assignment
- Attribution for any sources used, including people (other than instructors) who you asked for help
- Description of the file

Read all the way through this assignment twice before you start working on it. The second time through, take notes, and work on the design.

Background

In this assignment, you will design and write a Python program for a one-dimensional version of the Game of Life. The basic idea of the Game of Life is that cells in a grid are used to simulate biological cells. Each cell is either alive or dead. At each step of the simulation, each cell's current status and its number of living neighbors are used to determine the status of the cell for the next step of the simulation.

Typically, the Game of Life is set up as a two dimensional grid. In your case you will build a one-dimensional version that has N cells numbered 0 through N-1. Your grid will be stored in a Python list. Each cell *i* is adjacent to cells *i - 1* and *i + 1*. Exceptions: cell 0 is adjacent only to cell 1, and cell N-1 is adjacent only to cell N-2. A live cell has value 1, a dead cell has value 0. At each step of the simulation, the cell values at that step determine what changes will occur at the next timestep as follows:

- If a live cell *i* has two live neighbors, then cell *i* will die.
- If a live cell *i* does not have two live neighbors, then any dead neighbors of cell *i* will become alive.

Note that for all interior cells, two neighbors need to be examined. For the first and last cell (edge cases), there is only one neighbor to examine, and the rules above still apply (this has the effect that once alive, edge cells can never die).

Consider the state with 10 cells: **0111010101**
- Cell 0 is dead; it does nothing.
- Cell 1 is alive; it has a dead neighbor in cell 0, so cell 0 becomes alive in the next timestep. It does nothing to cell 2.

- Cell 2 is alive; it has two alive neighbors, so cell 2 dies in the next timestep.
- Cell 3 is alive; it has a dead neighbor in cell 4, so cell 4 becomes alive. (It treats cell 2 as still "alive", since that was the cell's state when this timestep started)
- Cell 4 is dead; it does nothing (again, based on the current timestep!)
- Cell 5 is alive; it has two dead neighbors, so both cell 4 and cell 6 become alive.
- Cell 6 is dead; it does nothing.
- Cell 7 is alive; it has two dead neighbors, so both cell 6 and cell 8 become alive.
- Cell 8 is dead; it does nothing.
- Cell 9 is alive; it has a dead neighbor in cell 8, so cell 8 becomes alive.

So at the next step of the simulation, the state would be: **1101111111**

And after another simulation step, then the state would become: **1111000001**

Work through this example by hand to make sure you understand the rules for cell modification

A Word About Python

You will need to do list copies as part of your solution to this problem. In Python the easiest way to do that is using a built-in method. You will need to import the Python **copy** library. Once you have that, you can write code like this

```
list2 = copy.copy(list1)
```

which makes an exact copy of list1 (otherwise you would have to write a loop yourself in order to make the copy). **DO NOT** try to copy a list by saying list2 = list1! This doesn't make a real copy, and will cause any changes to list2 to also change list1! If you want to see this in action, take this code, execute it in PythonTutor (http://pythontutor.com/) and you can see what happens in memory.

```
list1 = [1, 2, 3]
list2 = list1
list2[0] = 7
```

The Assignment

You will write four Python functions, as follows:

shouldDie(list, index)

- The *shouldDie* function takes a list and an index *i* as parameters. It returns True if cell *i* is alive and has two live neighbors; otherwise it returns False.

isZero(list, index)

- The *isZero* function takes a list and an index *i* as parameters. It returns True if the element at position *i* is zero, otherwise returns False.

advanceTime(list)

- The *advanceTime* function takes a list, the cells at timestep t, carries out one simulation of the Game of Life, and returns a list representing the cells at timestep t+1. The rough steps are:
    - copy the list
    - traverse the list and make changes in the copy as required by the rules
    - return the new version of the list

main()

- Create two constant variables:
    - NUM_TIME_STEPS: this is how many “generations” the simulation will run for. Set to a low value e.g., 1 or 2 as you write and test your code, and increase it to 10 after you confirm expected outputs for the lower values. It should be set to 10 in your submission.
    - NUM_ELEMENTS: this is the size of the Game of Life list. It should be set to 10 in your submission.
    - Create a variable called *cells*, a list of length NUM_ELEMENTS, each value initialized to 0 or 1. There are two ways to do this:
        - Create a method that creates and returns a random list of length NUM_ELEMENTS
        - Hard-code a new list of length NUM_ELEMENTS
        - Either way, be sure to test many variations of initial lists!
    - Print the initial value of *cells*
    - Run NUM_TIME_STEPS simulation steps, printing the updated *cells* after each simulation step.

Comments and Coding Style

Make sure sure all your code is well-commented and readable. Make sure to have:

- good variable names
- high level comments that explain what the program is doing or what it is implementing
- lower level comments that explain how particular elements of the program are implemented if the implementation is not obvious

Each function needs a comment!

How to Approach the Assignment

Before you do any coding, complete this design planning exercise by carrying out the following steps:

1. Reread the gameplay instructions and consult the sample runs below until you understand **exactly** what will change from one time step to the next.
2. Determine what data is involved, based on the problem description
3. Write out a high level program outline
4. Write more refined pseudocode within the outline, sketching out each function in pseudocode
5. Write more refined pseudocode for the main function

6. Then (and only then!) start turning your pseudocode into real code. Run and test your program A LOT, usually after writing one line of code, or even a partial or temporary line of code.
7. As you test your code, unexpected things will occur! Add print statements to inspect variable values to help understand what your program is doing.

Sample runs

For testing purposes, multiple sample runs are shown below. ***Before you begin to write any code***, it is crucial that you can replicate these by hand (given the initial values, carry out the next 2, 3, or more timesteps).

Example A

Initial values	[1, 0, 0, 0, 1]
Values after 1 timestep	[1, 1, 0, 1, 1]
Values after 2 timesteps	[1, 1, 1, 1, 1]

Example B

Initial values	[0, 0, 1, 0, 0]
Values after 1 timestep	[0, 1, 1, 1, 0]
Values after 2 timesteps	[1, 1, 0, 1, 1]

Example C

Initial values	[0, 0, 1, 1, 1]
Values after 1 timestep	[0, 1, 1, 0, 1]
Values after 2 timesteps	[1, 1, 1, 1, 1]
Values after 3 timesteps	[1, 0, 0, 0, 1]

Example D

Initial values	[1, 0, 1, 0, 0, 1, 1, 1, 1, 1]
Values after 1 timestep	[1, 1, 1, 1, 1, 1, 0, 0, 0, 1]
Values after 2 timesteps	[1, 0, 0, 0, 0, 1, 1, 0, 1, 1]
Values after 3 timesteps	[1, 1, 0, 0, 1, 1, 1, 1, 1, 1]
Values after 4 timesteps	[1, 1, 1, 1, 1, 0, 0, 0, 0, 1]
Values after 5 timesteps	[1, 0, 0, 0, 1, 1, 0, 0, 1, 1]
Values after 6 timesteps	[1, 1, 0, 1, 1, 1, 1, 1, 1, 1]
Values after 7 timesteps	[1, 1, 1, 1, 0, 0, 0, 0, 0, 1]
Values after 8 timesteps	[1, 0, 0, 1, 1, 0, 0, 0, 1, 1]
Values after 9 timesteps	[1, 1, 1, 1, 1, 1, 0, 1, 1, 1]
Values after 10 timesteps	[1, 0, 0, 0, 0, 1, 1, 1, 0, 1]

Rubric

Each section will be given a letter grade. The average of those letters will generate your final score. Note: Moodle does not allow letters, so they will be translated into a percentage there.

Grade	F	D	C	B	A

shouldDie, isZero	Not present	Exist, none function correctly	Exist, one functions mostly correctly	Exist, both function correctly with a bug or two	Exist, all function correctly
advanceTime	Not present	Exists, copies the list with no change	Exists, copies the list, goes through the list	Exists, copies the list, goes through the list, makes some correct changes for the step	Exists, copies the list, goes through the list, makes all correct changes for the step
main	Not present	Present, creates a list	Present, creates a list, loops through and prints it	Present, creates a list, loops through and prints it, changes it over time	Present, creates a list, loops through and prints it, changes it correctly over time
Comments, naming, style	No comments present, no name in file, method names incorrect	Name in file, some methods named correctly, no comments	Name in file, description of file in comment, methods named correctly (typos ok), more than one error that prevents running	Name in file, description of file and most methods, methods named correctly, some variables named meaningfully, significant over- or under-use of comments or one error that prevents running	Name in file, description of file and all methods, methods named correctly, all variables named meaningfully, other comments as needed, no errors that prevent running

Did you read through twice? Not yet? Go back and read it again!

Do you have questions? Are there things you don't understand? Talk to classmates and ask questions on Piazza.