CS1110 Summer 2021 — Assignment 5*

Due Tuesday 7/27 at 11:59pm

Read this before you begin

In the last assignment, we simulated the growth of fictional cells a long a single line. In this assignment, we are going to take it one step further, and simulate cells in a two-dimensional grid. The patterns that you will see at the end of this assignment will not just make pretty tapestries — they will be animated! It is hard to give you an example of what this should look like in a PDF file, but you can take a look at this Wikipedia page to get an idea of the patterns that may pop up. We will also demonstrate the end product in a lecture soon.

Learning Goals We will practice working with nested lists and nested for-loops (Unit 16), initializers (Unit 17), methods (Unit 18) and a little bit of type design (Unit 19). As announced in the previous assignment, you will be writing your own test cases again (Unit 7). We assume you are now comfortable using `if`-statements (Units 9 and 10) and object methods (Unit 8). If you run into any trouble along the way, please be sure to use your debugging skills (Unit 11), particularly reading error messages, to help solve issues you run into.

Good style We want you to keep adhering to a clean coding style. That is why you can again earn **10 points** for adhering to the style guide as posted on Canvas. It may be worth it to consult this guide again, because we have discussed some new topics since the last time you looked at it.

Restrictions You may use only the concepts we discussed in class so far. In particular, this means that you may *not* use `while`-loops or recursion to complete this assignment. Moreover, we encourage you to refrain from using any methods in the Python standard library beyond the ones that you have seen in class or in the labs.

Submission For this assignment, you will submit three files: `a5.py`, `a5test.py` and `a5app.py`

Submission within 24 hours after the deadline will be accepted with a 10% late penalty. Even if only one file (of several) is submitted late, the assignment as a whole will be marked late. Assignment submission on Canvas closes 24 hours after the deadline and no further submission will be allowed.

Do not put your name in the files that you will submit, but **do** include your NetID. The reason is to help promote fairness in grading. Thank you for heeding our request, even if it sounds a little strange.

Group work and academic integrity You must work either on your own or with one partner.

If you work with a partner, you must first register as a group in Canvas before you submit your work.

For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is *certainly* not OK to copy code from another person or from other sources.

If you feel that you cannot complete the assignment on you own, please seek help from the course staff.

*Adapted from an earlier assignment by A. Kellison and S. Marschner, as well as a very similar assignment that has been a staple of W. Koster's course at Leiden University for quite some years now.

Background: Conway's Game of Life

The board In Conway's Game of Life, the world consists of a two-dimensional grid of elements, called *cells*. Each cell can be either alive or dead in the current *state* of the game. Here is an example world that is five by five cells in size, where we use `x` to represent a cell that is alive, and `.` (a period) to denote a cell that is dead.

```
.....
..x..
...x.
.xxx.
.....
```
(1)

Let's adopt the convention that the origin — i.e., position $(0, 0)$ — is at the top left of the world, and that the x-axis runs from left to right, while the y-direction runs from top to bottom; this will be convenient later on. Under this convention, the cell at $(0, 0)$ is dead, while the cell two steps to the right and one down at $(2, 1)$ is alive.

Neighborhoods To calculate the *next* state of the game, we proceed as follows. Just like the last assignment, we look at the *neighbors* of each cell. However, because we are working in a two-dimensional grid, each cell has *eight* instead of three neighbors, given by the cells that are horizontally, vertically, or diagonally adjacent to the cell. For example, the eight neighbours of the live cell at $(2, 1)$ in (1) are as follows (the cell itself is drawn as `o`):

```
...
.o.
..x
```
(2)

Similarly, the eight neighbors of the live cell at $(3, 2)$ in (1) are as follows:

```
x..
.o.
xx.
```
(3)

Just like in the last game, we need to figure out what to do about cells at the edges of the grid. For this assignment, we adopt the convention that all cells outside of the grid are dead. So, the neighborhood of the dead cell at $(2, 4)$, the bottom-center position of the grid, can be drawn as follows (with the off-grid cells represented as dead cells):

```
xxx
.o.
...
```
(4)

The next generation The neighbors determine the state of the cell in the next generation, as follows:

- *Loneliness*: if a living cell has fewer than 2 neighbors, it dies.
- *Happiness:* if a living cell has 2 or 3 live neighbors, it lives on.
- *Overcrowding:* if a living cell has more than 3 live neighbors, it dies.
- *Reproduction:* a dead cell with exactly 3 live neighbors becomes alive.
- *Stasis:* a dead cell with any other number of live neighbors stays dead.

Let's go through some examples:

- The living cell at $(2, 1)$ has a neighborhood with just one live cell, as can be seen in (2). According to the *loneliness* rule, it will be dead in the next generation.
- The living cell at $(3, 2)$ has a neighborhood with exactly three live cells, as can be seen in (3). According to the *happiness* rule, it will (still) be alive in the next generation.
- The dead cell at $(2, 4)$ has a neighborhood with exactly three live cells, as can be seen in (4). According to the *reproduction* rule, it will be alive in the next generation.

If you calculate the status of each of the cells in the next generation, you will end up with the following grid:

```
.....
.....
.x.x.
..xx.
..x..
```
(5)

In fact, if you keep calculating the next generation of this pattern (on a sufficiently large board), you will find that it repeats itself, circling back to the initial pattern in (1), but "gliding" down and to the right. That is why this pattern is called a glider. You can look at that page on Wikipedia for an animation of its behavior.

That's it These really are all of the rules that govern Conway's Game of Life. Hopefully, you will see later on that they can lead to some fascinating patterns and interesting animations.

Part A: Creating a new board [10 points]

We will start by creating a data structure to hold information about the game board. You have been provided the skeleton file `a5.py`, which holds a stubbed out version of the class `Life`. Your task for this part is as follows.

- Implement the initializer `__init__` for the class `Life` according to the docstring.
- Implement the test function `test_init` in `a5test.py`, which tests this initializer on at least two inputs. Argue in a comment (a line preceded by `#`) that those two inputs test different things.

  You *must* use the functions from the module `testcase` that we used in Assignment 2, *for all test functions that you implement in this assignment.*

Part B: Randomizing the board [15 points]

The next thing we need to do is initialize the board randomly. To this end, you should first import the module `random`. Then, implement the method `randomize` in the class `Life` that does the following:

- Provide Python's random number generator with a seed-value, such that the random bits generated can be generated again later on.[1] This can be done using the function `seed` that lives in the `random` module.
- Next, using the function `randint` from `random`, populate the attribute `board` with cells that are randomly dead or alive. Try writing this yourself first; if you get stuck, you can have a look at the function `random_generation` that was part of the skeleton for Assignment 4.
- Implement the test function `test_randomize` in `a5test.py`. Note that testing this is actually a bit tricky. You *could* test it on different seeds to see if the board is given the same random cells for the same seed each time, but such a test might break when the internals of Python's random number generator change.

  Instead, try to think of two things that should *not* change when you call the `randomize` method, and test whether those two things stay the same.

Part C: The lines, they are a-changin' [40 points]

Counting neighbors Now we get to the part where the magic happens, and you get to implement the rules we discussed in the background section. The first step is to implement the method `count_neighbors`, which counts the number of live neighbors for the given coordinates, according to its docstring. You also need to implement the function `test_count_neighbors` in `a5test.py`. Try and think of at least two different test cases to try.

[1] This is the same principle that allows you to generate the same random Minecraft map as your friends, if you have the same seed!

The next generation Implement the method `next` according to its docstring. We suggest the following strategy:

- Create a new, blank, two-dimensional list of dead cells of the same width and height as in the attributes of current board, and store that two-dimensional list in a local variable `next_board`.
- For each cell on the *current* board, calculate how many neighbors are alive or dead using `count_neighbors`. Then, determine whether that cell should be alive or dead in the next generation based on its current state and the five rules that you were given; store the result at the appropriate position in `next_board`.
- Finally, overwrite the attribute `board` with `next_board`.

Just like the other methods, you should implement the relevant test function in `a5test.py`.

Part D: Printing out the current board [10 points]

We need a way to look at the current board. To that end, you should implement the method `print` in `Life` according to its docstring. This should be relatively straightforward, if you keep in mind the convention for the coordinates of the board: the x-axis runs from left to right, while the y-axis runs from top to bottom.

You do not need to implement any tests for the method `print`.

Part E: Putting it all together [15 points]

Finally, you should write a little user interface around your class in a new file called `a5app.py`. This file acts as a script. This script should do the following things, in the following order:

- Ask the user for the height and width of the board, and convert those to integers.

  You do not need to write code to handle the case where the user inputs something other than an integer.
- Ask the user for the random seed to use, and convert that to an integer.

  Again, no error-handling code for input that is not an integer is necessary.
- Create a new instance of the `Life` class with the given width and height.
- Randomize the board using the method you implemented in Part B and the seed provided by the user.
- Print the first 100 generations using the methods from Part C and Part D.

Don't forget to add a docstring to the top of your new file.

Top tip To make the animation look nice, you can put a bit of a delay between two generations. The built-in module `time` has a function `sleep` that takes a float f and causes your script to wait for f seconds.

So, for instance `time.sleep(1.5)` will make your script wait for a second and a half, before continuing We find that sleeping for 0.5 seconds provides for a nice animation.

Play around You can now experiment with the file. Keep in mind that if you supply a width greater than your terminal, the lines will not fit in your window. A good starting point would be a board of 70 characters in width and 15 characters in height — you can always supply higher numbers if there is more room.