CSE 231 Spring 2009

Programming Project 11

This assignment is worth 80 points and must be must **completed and turned in before** 11:59pm, Monday 4/27.

This project will give you experience on design and use of your own classes (i.e. no skeleton program is provided for this project).

Assignment Overview

The **Game of Life**, also known as **Life Game,** or simply **Life**, is a cellular automaton (a system that has rules applied to cells and their neighbors in a grid) designed by John Conway, a professor of Finite Mathematics at Princeton University in 1970. **Game of Life** is an example of "emergent complexity" or "self-organizing systems", which studies how elaborate patterns and behaviors can emerge from very simple rules.
Refer to http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life or http://www.math.com/students/wonders/life/life.html for more information about this game.

How to play the game?
The **Game of Life** is actually a zero-player game, which means the game is determined by its own rules and the initial pattern.

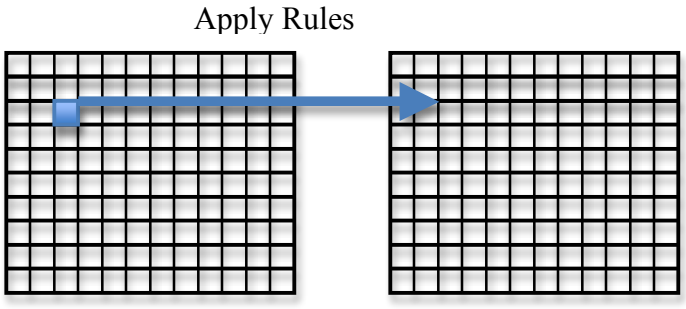
In this project, we'll start the game on a grid with "m * n" cells. Each cell of the grid has one of the two statuses: live or dead. The status of the m * n cells in the grid forms the initial pattern of the grid.
Each of the cells has 8 neighbors (unless on a boundary) as shown below.

1	2	3
4		5
6	7	8

Applying the following rules to each cell in the grid creates the next generation of the pattern. The rules are:

1. A dead cell with three live neighbors becomes a live cell.
2. A live cell with two or three live neighbors stays alive.
3. A live cell with less than 2 or greater than 3 live neighbors dies.

The rules are applied to the present grid, generating a new grid cells. That is, the rules are applied to the existing grid, but the results show up in the next step of the grid as shown below



Cells on the border have a reduced number of neighbors. That is, the grid does not "wrap" around to the other border

Task

Your task is to implement the game in Python using classes. Before implementing the game, try the example program or the online version of the game at http://www.math.com/students/wonders/life/life.html to make sure you understand it.

Program Specifications:

1) The size of the grid is specified by the user at the beginning of the game.
2) On the grid, the dead cells are represented by the dash sign ('-'), and living cells are represented by **the asterisk sign** ('*'). Here is an example of the output:

```
Column   0   1   2   3   4
Row   0  -   *   -   -   -
Row   1  *   *   *   *   -
Row   2  -   -   -   *   -
Row   3  *   -   *   -   *
Row   4  -   -   -   *   -
```

   Note that rows are counted from top to bottom; columns are counted from left to right.
3) The program has 2 parts:
   - The first part to generate an initial pattern randomly on the grid, and then applies rules of the game to calculate and display following generations step by step.
   - The second part to find all the "*Still Life*" patterns on the grid. A "*Still Life*" pattern/object is a pattern that remains still from generation to generation: all live cells remain alive and all dead cells remain dead. Refer to http://www.math.com/students/wonders/life/life.html for examples of still lives.
4) The program should have error checks like,
   - If a user gives non-integer inputs where integers are expected, notify the user that the input is incorrect and prompt again.

- If a user gives commands that are not expected when playing the game, notify the user and prompt again.

5) Part of the grade on this project will be the appropriateness of your class, methods, and any functions you use. The quality of the code will now matter as well as the performance. No skeleton file is provided; you need to come up with this yourself. Check previous project skeleton files as examples.

High Level Algorithm:

1) Create a class for the game. After the game starts, prompt the player for the size (number of rows and columns) of the grid. Within the class, create a list of lists to represent the grid.
2) Define a member function *printGrid* to print out current grid on the screen.
3) Define a member function *getAdj* to count the number of living cells that are immediately adjacent to a given cell horizontally, vertically or diagonally. The given cell's column and row number are passed as arguments to this function.
4) Your class should have a member function called *nextStep* to apply rules of the game to current pattern and get the next generation of the pattern.
5) Ask the user if he wants to play the game with a randomly generated pattern or find all the "*Still Life"* patterns for the grid.
6) If the user chooses to play the game with a random pattern, ask him for the initial number of living cells in the grid.
    a. After that, randomly place the living cells on the grid. For placing living cells, you could have a member function called *placeLivingCellsRandomly*. This function takes the initial number of living cells as a parameter.
    b. Keep calling the *nextStep* and *printGrid* functions as the user chooses to view the next generation of the pattern. Create a member function *isEmptyGrid* to check if the current grid is empty (no live cells inside) or not. If grid is empty, program should not give user the option to view the next generation of the pattern.
7) If the user chooses to find all the "*Still Life"* patterns for the grid, first ask for the number of cells you want to assign as alive in the grid.
    a. Then enumerate all the possible patterns of that size in the grid. Note that the combinatorics can get very large. For example, for a 5x5 grid, 5 live cells, you are looking at 25 choose 5 combinations, or about 53,000 patterns. For a 10x10 grid with all possible 10 live cell patterns, 100 choose 10 is about 17 trillion! Test your idea on small grids!
    b. For each pattern call *getGrid* to test the initial pattern and then call *nextStep* to generate the next pattern. If the initial pattern is the same as the new pattern, it's a "*Still Life"* pattern. Print out every "*Still Life"* pattern found during this process.
    c. More details on how to enumerate all the possible patterns in a grid are given later.
8) Repeat steps 5) ~7) until the user chooses to quit. Steps 5) ~7) could be coded in the main() function of the program.

Deliverables

You must use **handin** to turn in the file **proj11.py** – this is your source code solution; be sure to include your section, the date, the project number and comments describing your code. Please use the specified file name, and save a copy of your proj11.py file to your H drive as a backup.

Note on scoring. The life game itself will be worth 60 of the 80 points. The "still life" will be worth 20 points. Choose what to implement accordingly.

Tips/Hints:

1) Play with the provided demo program and get a feel for the game.
2) As a starting point, identify the variables and methods (also called as member functions) that you could have in the class.
3) When coding class methods remember the parenthesis—no error is generated for missing parenthesis, but results will not be what you expect.
4) Assuming perfect input first and then adds error checking later.
5) Test each individual function once it is implemented. This is easier than implementing the whole program and then debugging all functions.
6) A function called *combinations* is provided for you to enumerate all the possible patterns in a grid. It takes 2 parameters: a list of items **L** and the number of items **N** to be taken from **L**. This is the binomial coefficient or the choose function (see http://en.wikipedia.org/wiki/Binomial_coefficient).
    a. For example, given the list L = [0, 1, 2] and K = 2, function *combinations* will find all the combinations of size 2 that can be made from the three elements of L: [0,1], [0,2] and [1,2]. An example program *combinationExample.py* is provided to demonstrate the use of function *combinations*.
    b. To use *combinations* for the game, assign each cell in the grid an index number. For example, for a 5*5 grid, create a list [0, 1, 2, …, 24], such that each number in the list represents a cell in the grid. Then use *combinations* to find combinations of the cells of a particular size. For example *combinations(range(25),5)* yields all the combinations of 5 elements from the 25 cells of the 5x5 grid.
    c. Note that *combinations* is a generator and will not print the list. Rather, you must call it as many times as there are results in a loop (see the *combinationsExample.py* code)