CS 101
Fall 2013
Program 8
Algorithm due Sunday, Dec. 1.
Program due Sunday, Dec. 8.

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.

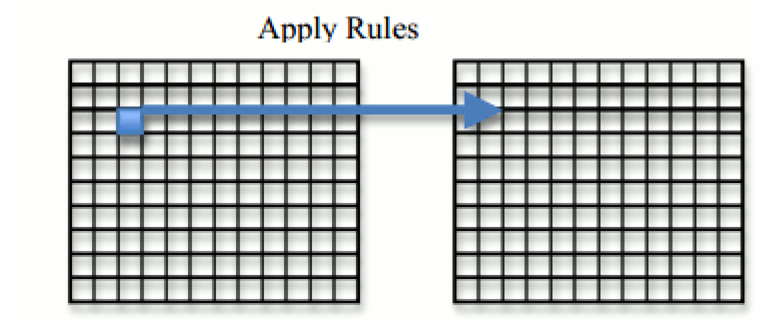
In this project, we'll start the game on a grid with "m * n" cells. Each cell of the grid has one of the two statuses: live or dead. The status of the m * n cells in the grid forms the initial pattern of the grid. Each of the cells has 8 neighbors (unless on a boundary) as shown below:

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 3 live neighbors becomes a live cell.
2. A live cell with two or three neighbors stays alive.
3. A cell with 0 or 1 live neighbors, or more than 3 live neighbors, dies.

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



Cells on the border have a reduced number of neighbors; the grid does not 'wrap around' to the other border.

Your task is to implement the game in Python using classes. Before implementing the game, try the example program or online version at http://www.math.com/students/wonders/life/life.html to be 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 cellsare 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. Bear in mind that once you start placing living cells, it's possible that a randomly-chosen cell has already been selected. In this case, you must choose another cell.
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 with 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

Turn in your python file by the deadline. The game itself will be the bulk of the points; the "still life" will be worth about 15 of the 60 points. Choose what to implement accordingly.

Tips/Hints:

1) Play with the online 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. Also remember to use 'self' as the first parameter of all member functions.
4) Assume perfect input first and then add 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) To generate all the possible ways to select a subset of grid squares, you should use the combinations() function in the itertools module. This is actually a generator, which is similar to a function with one important difference. When a function returns a value, the function ends and all of its local variables are garbage-collected. A generator *yields* a value, and remembers its current state for the next time it's called. The upshot is that if we need a large list of values (say, a list of all the ways you could select 5 cells from 25), each call to the generator returns one such item, and we never generate the entire list all at once. This is faster, and uses memory more efficiently. Here's an example:

```
>>> import itertools
>>> for combo in itertools.combinations('ABCDE', 3):
        print(combo)


('A', 'B', 'C')
('A', 'B', 'D')
('A', 'B', 'E')
```

```
('A', 'C', 'D')
('A', 'C', 'E')
('A', 'D', 'E')
('B', 'C', 'D')
('B', 'C', 'E')
('B', 'D', 'E')
('C', 'D', 'E')
```

Note that we iterate through all the possible combinations, by making a separate call to the generator each time.

The first parameter is an iterable object—a list, tuple, set, string, etc. The second is the number of items we want to select. To use this in your program, make a list of the appropriate size—for example, if the grid is 5x5, make a list of 25 numbers (0 or 1 based indexing doesn't matter as long as you're consistent — 0-based indexing will be easier to code.) You want each cell in the grid to have its own unique index number. Then call itertools.combinations(YourList, NumberToSelect) to get each combination, and test it as discussed above. Note that each call to itertools.combinations() returns ONE combination; you must call it as many times as there are results in a loop:

```
>>> import itertools
>>> j = list(range(25))
>>> for combo in itertools.combinations(j, 2):
     print(combo)
```