6.189 Project 2

Readings

Review OOP notes and readings; http://en.wikipedia.org/wiki/Conway_game_of_life .

What to hand in

Please print out ONLY the Board class from your code (ie from `class Board` through `def animate`). Be sure to write *your name* and *section* clearly at the top of your code!

0. Preliminaries

This project will have you implement a version of Conway's Game of Life. It will help you prepare for the final project, an implementation of Tetris, by familiarizing you with some of the basic Tetris framework. You may work with a partner, but you must each submit your own solution (no copying or cheating allowed).

To begin, please visit the course webpage and download the code template to the same directory where you have saved the file `graphics.py`. All the places where you will need to add code have a comment '**YOUR CODE HERE**'. At the end of the project, you should have code in all the places where you find this comment. Note we have coded the Block and Board classes such that you don t have to think in pixels. The Block `__init__` method takes care of sizing the block to the pixels specified in the global variable `BLOCK_SIZE` and placing it on the grid. Thus, you only need to think of the board as a grid of size `BOARD_WIDTH` by `BOARD_HEIGHT`. Remember that square (0,0) will be in the top left corner, with square (10,10) in the bottom right.

Read through the provided code, and understand the above paragraph, before progressing any further! Whenever you get template code, it is essential you read through it before coding - that way, you understand what is going on, and what classes and methods are available to you. Ask an LA if you don't understand aspects of the code.

1. A block for each square

In the `__init__` method of the `Board` class, you need to finish laying out the board. Currently, only the board is drawn. We wish to make a Block to each square on the board, and store that Block in a dictionary (`self.block_list`). Remember from Exercise 4.3 that Blocks can be any color listed in `rgb.txt`! Read the comments in the code and think about the best way to do this. Note that you just want to *create* the Block; you do *not* want to set it to live! Also, be sure to delete the `raise Exception` line.

Before continuing, test if your implementation works. Find the section `RUNNING THE SIMULATION`, and make sure the code underneath `PART 1` is uncommented. Run the code, and if everything works, the board should pop up with random squares colored in.

2. Adding specific tiles

Now we wish to be able to seed the board with specific patterns of tiles. It's your job to implement the `seed` method of the Board class, which takes in a Python list of block coordinate tuples - something like [(x1, y1), (x2, y2)] - and sets the Blocks in those positions to be live. If you get stuck, review the implementation of `random_seed` to see how we can get the blocks to be visible on the board; however, *you should not use the same for loop that* `random_seed` *uses* - hint: your loop should involve the parameter `block_coords`...

Before continuing, test if your implementation works. Comment the random_seed method and uncomment the code underneath `PART 2`. If everything works, one phase of the Toad pattern (found on the Wikipedia page) should appear.

3. Finding our neighbors!

For part 4, you ll need a way of figuring out the neighboring Blocks bfor any given Block. You'll need to implement the method `get_block_neighbors` in the Board class. This method takes in a Block object, and should return a Python list of neighboring blocks. Neighbors can be horizontally, vertically, or diagonally adjacent - so a block in the center of the board would have 8 neighbors, while one in a corner would only have 3. There are many ways of approaching this problem, so take a few minutes to devise a good algorithm, and then try to implement it.

Comment the code under `PART 2` and uncomment the two lines under `PART 3`. If everything works, squares at positions (0,0) and (1,1) should appear colored in, and your Python interpreter should print Passed neighbor test .

4. Adding the rules

Now it is time to actually code in the rules of the Game, within the `simulate` method. Note the rules are as follows:

1. Any live cell with fewer than two live neighbours dies, as if caused by under-population.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with exactly two or three live neighbours lives on to the next generation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

It is easiest to do this in two discrete steps, although you may try any approach that you would like. The first step is to go through the board's blocks and calculate the `new_status` attribute of each block - ie, the status the blocks need to change to - and save the calculated status in the `new_status` attribute. The second step is to go through the blocks again and set them to live or dead (the calculated `new_status` accordingly. Run the provided `reset_status` method on each Block to reset its status (read through that method and understand what it does before using it). We encourage you to break up this task into a few smaller functions, that you can more easily debug. Think about what parts you can separate into smaller methods.

Comment the code under `PART 3` and uncomment the two lines under `PART 4`. If everything works, the Toad pattern should appear, and two seconds later, it should shift phase. MAKE SURE THIS WORKS BEFORE CONTINUING!

5. Running the animation!

Congrats, you're seconds away from being done! Comment out the line that says `win.after(2000, board.simulate)`, and uncomment the code underneath `PART 5`. Now, run your code. Your simulation should animate as expected! Try seeding the board with some of the different blocklists in the GLOBAL VARIABLES section, try randomly seeding the board, and try making up your own blocklists! Let us know if you find anything cool.

6.189 A Gentle Introduction to Programming

January IAP 2011