CSc 110, Spring 2017
Programming Assignment #9: Segregation (20 points)
Due Tuesday, April 4th, 7:00 PM

Thanks to Frank McCown from Harding University and Marty Stepp from the University of Washington.

This assignment focuses on lists of lists. Turn in a file named `segregation_simulation.py`.

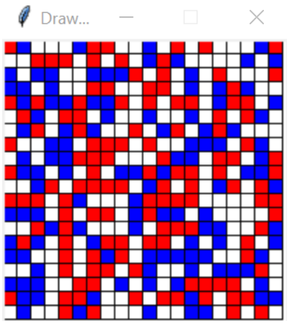
Background Information:

Racial segregation has always been a pernicious social problem in the United States. Although much effort has been extended to desegregate our schools, communities, and neighborhoods, the US continues to remain segregated by race and on economic lines. Why is segregation such a difficult problem to eradicate?

In 1971, the American economist Thomas Schelling created an agent-based model that might help explain why segregation is so difficult to combat. His model of segregation showed that even when individuals (or "agents") didn't mind being surrounded or living by agents of a different race, they would still choose to segregate themselves from other agents over time! Although the model is quite simple, it gives a fascinating look at how individuals might self-segregate, even when they have no explicit desire to do so.

In this assignment, you will create a simulation of Schelling's model.

How the program works:



The model consists of two types of agents: 1 and 2. The two types of agents might represent different races, ethnicities, economic statuses, etc. You will create an animation of these two types of agents moving over time.

When the simulation starts the simulation grid is populated randomly with 40% agent 1 and 35% agent 2. The rest of the squares are left empty.

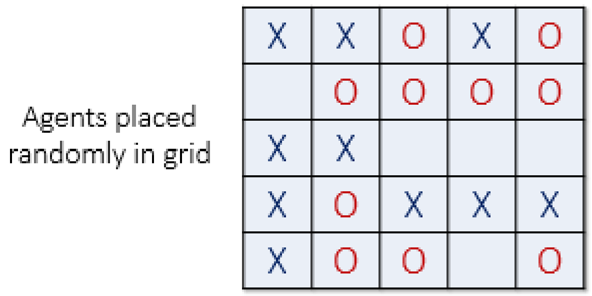
Then, throughout the simulation, at every tick of the clock all dissatisfied agents are moved to new locations. These new locations can be any empty square. They do not have to be squares that the agent will be happy in.

The simulation continues until all agents are satisfied.

Program Behavior and Development Strategy:

Step 1: populate the simulation grid

The two populations (the two agent types) are initially placed into random locations in a neighborhood represented by a grid (list of lists). After placing all the agents in the grid, each cell is either occupied by an agent or is empty as shown below.


Agents placed randomly in grid

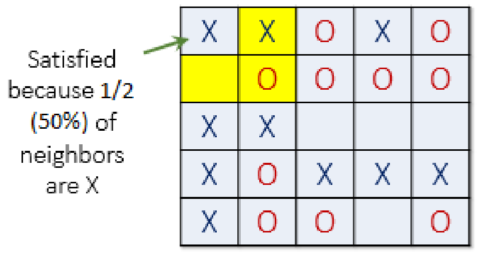
X	X	O	X	O
	O	O	O	O
X	X			
X	O	X	X	X
X	O	O		O

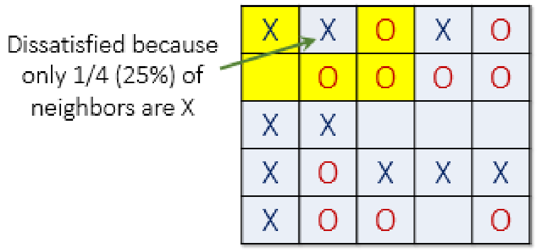
There should be a 40% probability a square contains agent 1, a 35% probability a square contains agent 2 and a 25% probability a square is empty. To fill squares with this probability, generate a number between 1 and 100 and place agent 1 in a square if it is 40 or below and agent 2 in a square if it is above 40 and 70 or below. You can represent the agent types using whatever symbols you prefer. The examples in the spec will all use `X` and `O` characters as in the diagram at left.

Step 2: Finding unsatisfied agents

The next step is to determine if each agent is satisfied with its current location. A satisfied agent is one that is surrounded by at least 30% percent of agents that are like itself. This 30% threshold is one that will apply to all agents in the model, even though in reality everyone might have a different threshold they are satisfied with. Note that the higher the threshold, the higher the likelihood the agents will not be satisfied with their current location.

The picture below (left) shows a satisfied agent because 50% of X's neighbors are also X. The X in the picture below to the right is not satisfied because only 25% of its neighbors are X. Notice that in this example empty cells are not counted when calculating satisfaction.





Step 3: Moving agents

You will need to move each unsatisfied agent to a new location. This new location does NOT have to be one in which they will be satisfied. For each dissatisfied agent you find you will need to find a random empty location to move it to. To do this, start at a random location on the board and traverse your grid until you find an empty location. Move the dissatisfied agent here.

Step 4: Visualization

In order to watch the simulation you will need to create a visualization on a drawing panel. Draw a grid on the drawing panel with the dimensions of your list of lists. Make each square 10 by 10 pixels. Color the squares differently depending on which type of agent occupies them. The picture in this document uses red and blue for agents and white for empty squares but you may use whatever colors you would like. Make your drawing panel the same size as your grid.

Step 5: Simulating

Animate the simulation by checking for dissatisfied agents, moving them and redrawing every 100 milliseconds. Your simulation should continue until all agents are satisfied. Note that this may never occur in some cases.

Style Guidelines:

For this assignment you are required to have the following **five constants**:

- one for the **percentage of agent 1** (40%)
- a second for the **percentage of group 2** (35%)
- a third for the **size of the grid** (you may assume it is square) (20)
- a fourth for the **speed of the animation** (100 milliseconds)
- a fifth is the **happiness percentage** (30%)

For full credit it should be possible to change the constant values and cause your program to change its behavior. Refer to these constants in your code and do not refer to the bare numbers such as 40 or 30 directly. You may use additional constants if they make your code clearer.

We will grade your function structure strictly on this assignment. Use at least **five nontrivial functions** besides `main`. These functions should use parameters and returns, including lists of lists, as appropriate. The functions should be well-structured and avoid redundancy. No one function should do too large a share of the overall task. You may not nest these functions inside each other or in main.

Your `main` function should be a concise summary of the overall program. It is okay for `main` to contain some code but `main` should not perform too large a share of the overall work itself, such as traversing over the list of lists. Also avoid "chaining," when many functions call each other without ever returning to `main`.

We will check strictly for redundancy on this assignment. If you have a very similar piece of code that is repeated several times in your program, eliminate the redundancy such as by creating a function, by using `for` loops over the elements of lists, and/or by factoring `if/else` code.

Since lists of lists are a key component of this assignment, part of your grade comes from using lists of lists properly. Carefully consider how they should be passed as parameters and/or returned from functions as you are decomposing your program. Recall that lists use *reference semantics* when passed as parameters, meaning that a list passed to a function can be modified by that function and the changes will be seen by the caller.

You are limited to features in lectures 1 - 25. Follow past style guidelines such as indentation, names, variables, line lengths, and comments (at the beginning of your program, on each function, and on complex sections of code). You may not have any global variables or code outside of functions.