MTH 3300
Spring 2019
E. Fink
April 14, 2019

Homework 9

Directions: Download the template files I have provided on Blackboard. Then open Spyder, load these template files, and write the following programs. Submit your source code to me via Blackboard, in `.py` format; do NOT send any other files. READ THE INSTRUCTIONS on how to submit your work in the Course Documents section of Blackboard.

Be sure to read the SPECIFICATIONS carefully for all problems! And write comments!

1) beverages.py

Remember the Coke and Pepsi problem from Lecture 17? Reimplement this using object-oriented programming.

Recall that the goal of that problem was to create a program whose `main()` function contains code that allows the user to input an x- and y-coordinate. The program should then output the number of Coke drinkers within a distance of 1 from that location (using the usual distance formula, $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$), and the number of Pepsi drinkers within a distance of 1. The data should come from the `surveydata.txt`. Recall that each line of that file corresponds to a single resident, given the x-coordinate, y-coordinate and drink preference of that resident.

Your solution should involve making a class called `Resident`. Each `Resident` object should contain three attributes: an x-coordinate, a y-coordinate, and a drink preference. The class should support at least one method aside from `__init__`, called `.distance()`. This method, when applied to a `Resident`, should accept exactly two (outside) arguments – the x and y coordinates of some point – and return the distance from the `Resident` to that point. For example, if `r` is a `Resident` who is located at $(0, 0)$, then `r.distance(3,4)` should return the value `5.0`. You are free to include other methods, at your discretion.

When you load the `surveydata.txt`, instead of creating a 2D list out of the contents, use the information from each line of the text file to initialize a `Resident` object, and then append this object to a (1D) list of `Resident`s.

Other design decisions are left to you.

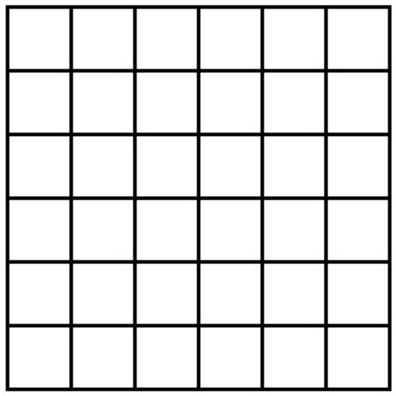
Specifications: your program must

- allow the user to input an x- and y-coordinate, and then print out the number of Coke drinking residents who lie with a distance of 1 of that point, and the number of Pepsi drinking residents who lie with a distance of 1 of that point. Your counts should be out of the people represented in the file `surveydata.txt` – each line of that file represents the x-coordinate, y-coordinate, and drink preference for one of the residents.
- write and use the definition of a class named `Resident`, whose objects should have (at least) three attributes (for x-coordinate, y-coordinate and drink preference).
- write the definition of at least one method aside from `__init__`, named `.distance()`, which should behave as described above.
- write your code in such a way that when you read the file `surveydata.txt`, the data is stored as a 1D list of `Resident` objects, NOT a 2D list (whose entries would consist of floats and strings).

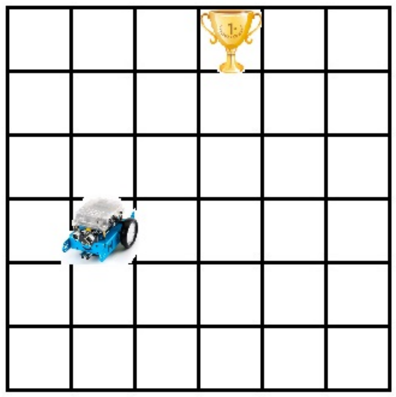
Challenge*: create the `matplotlib` plot showing the "red and blue dot" scatter plot using the list of `Resident` objects. The plot should also include a legend; the location that the user entered, marked with an "X"; and a circle showing all the points within a distance of 1 from the location that the user entered.*

2) robot.py

Consider the following 6×6 grid:



Suppose I drop a robot and a prize on two random squares:



I then play a game: I let the robot move randomly. It makes one move at time, each time moving one square up, down, to the left, or to the right. I let the robot move until it either reaches the prize, or moves to a square that is off the grid.

Question: what is the probability that the robot reaches the prize before it moves off the grid? Answer this question by simulating 10,000 games, and writing a class as follows.

Create a class called `Robogame`, whose objects will each represent a game. The attributes for each `Robogame` object will represent the locations of the robot and the prize; numerous methods will be written that allow the game to play out.

Specifically, you should implement your class to the following specifications. I suggest you deal with the methods in the order I write them. Remember that the attributes, which both begin with an underscore (`._robot`, `._prize`) should *only be used in the class definition – not in your client code* (aside from the two lines of test code that I provide)! You are also free to add additional attributes and methods, if you wish.

Each object in the class should have the following **attribute variables**:

- `._robot`, which is a list containing two `int`s, representing the square which currently contains the robot.
- `._prize`, which is a list containing two `int`s between 0 and 5, representing the square which contains the prize.

The class should also contain the following **methods**:

1. `.__init__()`. This constructor, used to create a game, should receive no arguments, other than `self`. The constructor should initialize both `._robot` and `._prize` randomly, each as a list containing two `int`s between 0 and 5, inclusive. (They could be initialized to the same square, in which case the game will be over immediately.)
2. `.set_robot()`, which should receive two `int`s `i` and `j`, and return nothing. This should set the location of `self._robot` to be `[i, j]`. (You may not use this method very much, but it's helpful for testing.)
3. `.set_prize()`, which should receive two `int`s `i` and `j`, and return nothing. This should set the location of `self._prize` to be `[i, j]`. (You *almost certainly* will not use this method very much, but again, it's helpful for testing.)
4. `.display()`, which should receive no outside arguments and return nothing. This should print out a grid representation of the game in its current state: it should print 6 lines, each with 6 characters – a `P` for the square with a prize, a `R` for the square with the robot, or a `.` for any of the other squares. For example, `.display()` could produce

```
...P..
......
......
.R....
......
......
```

if `self._robot` was `[3,1]` and `self._prize` was `[0,3]`. If the prize and robot happen to coincide, that square should contain a `C`; if the robot's position is off the grid, it should not be depicted.

5. `.win()`, which should receive no outside arguments, and return `True` if the current value of `self._robot` is equal to the value of `self._prize`; the function should return `False` otherwise.
6. `.off_grid()`, which should receive no outside arguments, and return `True` if the current value of `self._robot` has either entry not between `0` and `5`, inclusive; the function should return `False` otherwise.
7. `.step()`, which should receive no outside arguments, and return nothing. This function should choose a random direction out of {up, down, left, right}, and change the robot's position by one square in the chosen direction. (Each direction should be equally likely to be chosen.)

I've put some test code in my template file – please leave it in your final submission. **I strongly suggest that you build this class one component at a time, in the order I suggested, and then run the relevant portions of my test code, commenting out the parts that correspond to what you haven't implemented yet.**

Finally, answer the question we asked initially. Do this by simulating 10,000 different `Robogame`s (create a new object for each simulation!) in client code. Have the game's robot take random steps until either the robot lands on the square with the prize, or lands on a square outside the grid. The program should count how many games out of the 10,000 end with the robot finding the prize. You should probably be using the `.win()`, `.off_grid()` and `.step()` methods.

Specifications: your program must

- contain a definition of the class `Robogame` that includes the 7 methods listed above, which allows my test code to run properly.
- contain code which uses this class to simulate 10,000 games, and reports the percent of the games where the robot reaches the prize before reaching a square off of the grid.
- not contain any references to `._robot` or `._prize` in client code (i.e., outside of the class definition).

Challenge*: answer the same question for a slightly different game. In this version, the robot still makes random moves of length 1, but there are two prizes, and the robot is not allowed to visit any square twice. The game ends with a win when the robot gets both prizes, or ends with a loss when the robot wanders off the grid or cannot make a move.*