6.0002 Problem Set 3: Robot Simulation

Handed out: November 5, 2018
Due: 9:00 PM, November 14, 2018
Checkoffs: Monday November 19th - Monday November 29, 2018

Introduction

In this problem set, you will design a simulation and implement a program that uses classes to simulate robot movement. We recommend testing your code incrementally to see if your code is not working as expected. To test your code, run `ps3_tests_f18.py`.

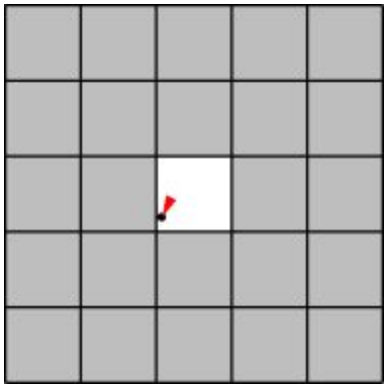
As always, please do not change any given function signatures. Remember to consult the the 6.00/6.0002 Style Guide (available on Stellar) as **we will be checking for Style Guide violations as part of your checkoff.**

A) Simulation Overview

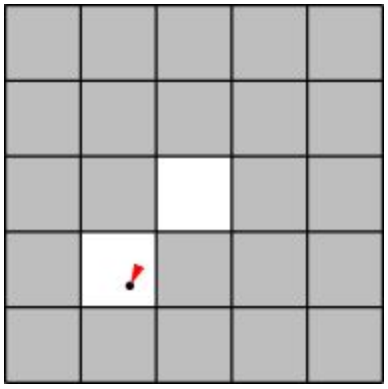
iRobot is a company (started by MIT alumni and faculty) that sells the Roomba vacuuming robot (watch one of the product videos to see these robots in action). Roomba robots move around the floor, cleaning the area they pass over.

You will code a simulation to compare how much time a group of Roomba-like robots will take to clean the floor of a room. The following simplified model of a single robot moving in a square 5x5 room should give you some intuition about the system we are simulating. A description and sample illustrations are below.

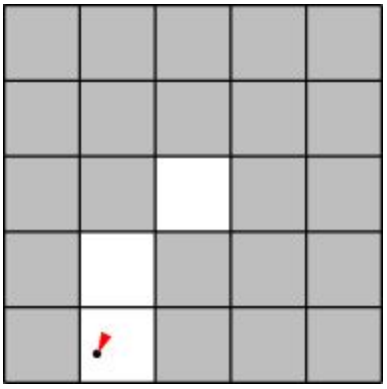
The robot starts out at some random position in the room. Its direction is specified by the angle of motion measured in degrees clockwise from “north.” Its position is specified from the lower left corner of the room, which is considered the origin (0.0, 0.0). The illustrations below show the robot's position (indicated by a black dot) as well as its direction (indicated by the direction of the red arrowhead).



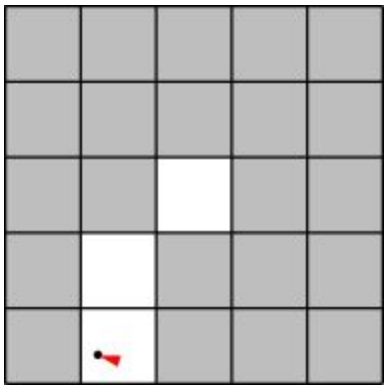
Time *t* = 0
The robot starts at the position (2.1, 2.2) with an angle of 205 degrees (measured clockwise from "north"). The tile that it is on is now clean.



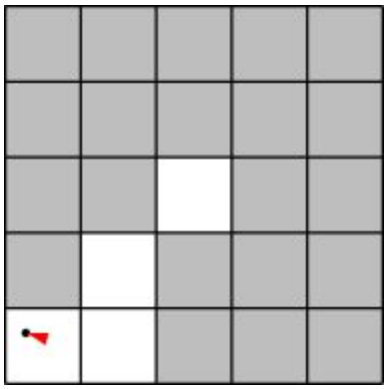
***t* = 1**
The robot has moved 1 unit in the direction it was facing, to the position (1.7, 1.3), cleaning another tile.



***t* = 2**
The robot has moved 1 unit in the same direction (205 degrees from north), to the position (1.2, 0.4), cleaning another tile.



***t* = 3**
The robot could not have moved another unit in the same direction without hitting the wall, so instead it turns to face in a new, random direction, 287 degrees.



***t* = 4**
The robot moves along its new direction to the position (0.3, 0.7), cleaning another tile.

B) Simulation Components:

Here are the components of the simulation model.

1. **Room:** Rooms are rectangles, divided into square tiles. At the start of the simulation, each tile is covered in some amount of dirt, which is the same across all the tiles. You will first implement the class **RectangularRoom** in Problem 1.
2. **Robot:** Multiple robots can exist in the room. iRobot has invested in technology that allows the robots to exist in the same position as another robot without causing a collision. You will implement the abstract class **Robot** in Problem 1. You will then implement the subclasses **SimpleRobot**, **RobotPlusCat** and **BoostedRobot** in Problems 2, 3, and 4.

More details about the properties of these components will be described later in the problem set.

C) Helper Code

We have provided an additional file: `ps3_visualize.py`. This Python file contains helper code for visualizing your robot simulation. You do not need to worry about the contents of this file. To test your code, run `ps3_tests_f18.py`.

Problem 1: Implementing the RectangularRoom and Robot classes

Read `ps3.py` carefully before starting, so that you understand the provided code and its capabilities. **Remember to carefully read the docstrings for each function to understand what it should do and what it needs to return.**

The first task is to implement the class `RectangularRoom` and the abstract class `Robot`.

In the skeleton code provided, the abstract class contains some methods which should only be implemented in the subclasses. **If the comment for the method says "do not change," please do NOT change it.** You can test your code as you go along by running the provided tests in `ps3_tests_f18.py`.

In `ps3.py`, we've provided skeletons for these classes, which you will fill in for Problem 1. We've also provided for you a complete implementation of the class `Position`. Do not change the `Position` class.

Class Descriptions:

- **RectangularRoom** - Represents the space to be cleaned and keeps track of which tiles have been cleaned.
- **Robot** - Stores the position, direction, and cleaning capacity of a robot.
- **Position** - Represents a location in *x*- and *y*-coordinates. `x` and `y` are floats satisfying $0 \leq x < w$ and $0 \leq y < h$, where `w` and `h` are the room's width and height.

RectangularRoom Implementation Details:

- Representation:
    - You will need to keep track of which parts of the floor have been cleaned by the robot(s). When a robot's location is anywhere inside a particular tile, we will consider the dirt on that entire tile to be reduced by some amount determined by the robot. We consider the tile to be "clean" when the amount of dirt on the tile is 0. We will refer to the tiles using ordered pairs of **integers**: (0, 0), (0, 1), …,

(0, *h*-1), (1, 0), (1, 1), …, (*w*-1, *h*-1).
    - Tiles can **never** have a negative amount of dirt.
- Starting Conditions:
    - Initially, the entire floor is uniformly dirty. Each tile should start with an integer amount of dirt, specified by `dirt_amount`.

Robot Implementation Details:

- Representation
    - Each robot has a **position** inside the room. We'll represent the position using an instance of the `Position` class. Remember the `Position` coordinates are floats.
    - A robot has a **direction of motion**. We'll represent the direction using a float `direction` satisfying `0 ≤ direction < 360`, which gives an angle in degrees from north.
    - A robot has a **cleaning capacity**, `capacity`, which describes how much dirt is cleaned on each tile at each time.
- Starting Conditions
    - Each robot should start at a random position in the room (hint: the `Robot`'s `room` attribute has a method you can use)
- Movement Strategy
    - A robot moves according to its movement strategy, which you will implement in `update_position_and_clean`.

If you find any places above where the specification of the simulation dynamics seems ambiguous, it is up to you to make a reasonable decision about how your program/model will behave, and document that decision in your code.

Complete the `RectangularRoom` class and `Robot` abstract class by implementing their methods according to the specifications in `ps3.py`. Remember that the `Robot` class will never be instantiated; we will only instantiate its subclasses.

Hints:

- Make sure to think carefully about what kind of data type you want to use to store information about the floor tiles in the `RectangularRoom` class.
- A majority of the methods should require only one line of code.
- In the final implementation of the `Robot` abstract class, not all methods will be implemented. Not to worry — their subclass(es) will implement them (e.g., `Robot`'s subclasses will implement the method `update_position_and_clean`).
- Remember that tiles are represented using ordered pairs of **integers** (0, 0), (0, 1), …, (0, *h*-1), (1, 0), (1, 1), …, (*w*-1, *h*-1). But a robot's `Position` is specified as **floats** (x, y).

Be careful converting between the two! We recommend using math.floor(x) to always round down when converting to ensure that Positions are always in the room.

- Remember to give the robot an initial **random** position and direction. The robot's position should be of the `Position` class and should be a valid position in the room. Note that the class `RectangularRoom` has a `get_random_position` method that may be useful for this.
- **Your implementation may occasionally fail a few simulation tests. This is ok, and will not be counted against you as long as all of the tests pass most of the time.**

Problem 2: SimpleRobot and Simulating a Timestep

Each robot must also have some code that tells it how to move about a room, which will go in a method called `update_position_and_clean`.

We have already refactored the robot code for you into two classes: the abstract `Robot` class you completed above (which contains general robot code), and a `SimpleRobot` class inheriting from it (which contains its own movement strategy).

The movement strategy for `SimpleRobot` is as follows: in each time-step:

- Calculate what the new position for the robot would be if it moved straight in its current direction at its given speed.
- If that is a valid position, move there and then clean the tile corresponding to that position by the robot's capacity. The position is valid if it is in the room. Do not worry about the robot's path in between the old position and the new position.
- Otherwise, rotate the robot to be pointing in a random new direction. **Don't clean the current tile or move to a different tile**.

We have provided the `get_new_position` method of the `Position` class, which you may find helpful in implementing this. It computes and returns the new `Position` for the current `Position` object after a single clock-tick has passed with the given angle and speed parameters. Read the docstring for this method for more information.

Complete the `update_position_and_clean` method of `SimpleRobot` to simulate the motion of the robot during a single time-step (as described above in the time-step dynamics).

Testing Your Code:
Before moving on to Problem 3, check that your implementation of `SimpleRobot` works by uncommenting the following line under your implementation of `SimpleRobot`:

```
test_robot_movement(SimpleRobot, RectangularRoom)
```

The test file will display a 5 by 5 room as implemented in `RectangularRoom` and a robot as implemented in `SimpleRobot`. Initially, all dirty tiles are marked as black. As the robot visits each tile and cleans the tile according to its given capacity, the color of the tile changes from black to gray to white, with white meaning the tile is completely clean.

Make sure that as your robot moves around the room, the tiles get lighter (from black to gray to white as shown below on page 10) each time when your robot traverses. The simulation terminates when the robot finishes cleaning the entire room. Make sure your robot doesn't violate any of the simulation specifications (e.g., your robot should never move to a position outside of the room, it should never clean the tile if it also had to choose a new direction, etc.)

Do not worry if it appears your robot is "cutting corners" as it cleans, as long as its final position in each time step is never outside of the room. When you've checked that your robot moves correctly, **make sure to comment out the** `test_robot_movement` **line**.

Problem 3: Implementing RobotPlusCat

iRobot's roombas have become quite the hit sensation. In fact, they are really popular with cats. Your cat likes to take a ride on your roomba, and it will sometimes paw at the controls while aboard the robot. If your cat interferes with the robot's operation, the robot will forget to clean a tile and will change direction. You wonder how badly this affects the time it takes a robot to clean a room and decide to design a simulation.

Note: Cat interference is determined for each timestep. If your cat plays with the robot's controls at one timestep, it may or may not mess with the controls again at the next timestep.

Write a new class `RobotPlusCat` that inherits from `Robot` (just as `SimpleRobot` inherits) but factors in your cat's behavior. `RobotPlusCat` should have its own implementation of `update_position_and_clean`.

The behavior for a `RobotPlusCat` is outlined in the docstring.

We have written a method `gets_cat_interference` inside `RobotPlusCat` for you that you should use in order to determine if your cat messes with your robot's controls. Initially the cat plays with the controls with probability `p = 0.15`. As with `SimpleRobot`, you may find the provided `get_new_position` method of `Position` helpful.

Testing Your Code

Test out your new class. Perform a single trial with the new `RobotPlusCat` implementation and watch the visualization to make sure it behaves as expected.

```
test_robot_movement(RobotPlusCat, RectangularRoom)
```

Problem 4: Implementing BoostedRobot

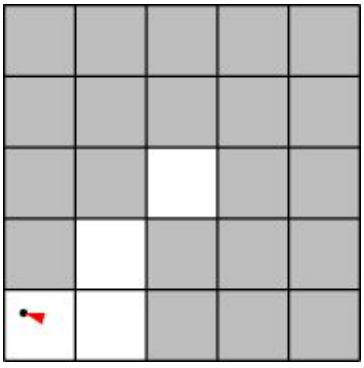
Determined to keep cats off their robots to increase cleaning speed, iRobot churned out a batch of Boosted Robots. These robots are able to clean two tiles in one timestep by using super boosters. However, if they hit a wall, they might knock dust off the wall and dirty a tile.

Write a new class `BoostedRobot` that inherits from `Robot` (just as `SimpleRobot` inherits) but implements a new movement strategy. `BoostedRobot` should have its own implementation of `update_position_and_clean`.

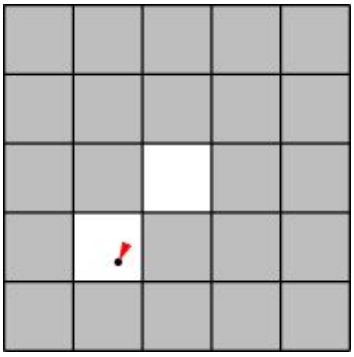
If the `BoostedRobot` hits a wall when it attempts to move in its current direction, it may dirty the tile adjacent to the wall because it moves very fast and can knock dust off of the wall. This will increase the dirtiness of the tile by 1 unit.

There are three possible cases:

1. The robot tries to move. If it immediately hits the wall, it does not move forward, turns to face a random direction, and stops for this timestep. It does not dirty the tile with any probability, because it was not traveling fast when it hit the wall. An example of this case is shown below.



2. If the robot can move, it moves and cleans the tile it moves to. Then, it tries to move a second time. If it hits the wall, it dirties the tile it is on by one unit with probability *p.* After hitting the wall, regardless of whether it dirties the tile, the robot turns to a random direction and then stops. An example of this case is shown below.



3. If the robot can move, it moves and cleans the tile it moves to. Then, it tries to move a second time. If it does not hit the wall, it moves and cleans the tile it moves to.

We have written a method `dirties_tile` inside `BoostedRobot` for you that you should use in order to determine if the robot coughs up dust. Initially the robot dirties the tiles adjacent to the wall with probability `p = 0.1337`. As with `SimpleRobot`, you may find the provided `get_new_position` method of `Position` helpful.

Testing Your Code

Test out your new class. Perform a single trial with the new `BoostedRobot` implementation and watch the visualization to make sure it behaves as expected.

```
test_robot_movement(BoostedRobot, RectangularRoom)
```

Problem 5: Creating the Simulator

In this problem you will write code that:

1. Simulates the robot(s) cleaning the room up to a specified fraction of the room; and
2. Outputs how many time-steps are needed on average to clean the room.

Once you have written this code, you'll comment on the results of your simulation in Problem 6.

Implement `run_simulation(num_robots, speed, capacity, width, height, dirt_amount, min_coverage, num_trials, robot_type)` according to its specification.

Simulation Starting Conditions:

1. Each robot should start at a random position in the room.
2. Each room should start with a uniform amount of dirt on each tile, given by *dirt_amount*. A given trial of the simulation terminates when a specified fraction of the room tiles have been fully cleaned (i.e., the amount of dirt on those tiles is 0).

Simulation Animation:
If you want to see a visualization of your simulation, similar to the visualization that pops up when you call `test_robot_movement`, check the end of this pset for instructions!

The first six parameters of `run_simulation` should be self-explanatory. If you are confused, check the docstrings. For the time being, you should pass in `SimpleRobot` for the robot_type parameter, like so:

```
avg = run_simulation(10, 1.0, 1, 15, 20, 5, 0.8, 30, SimpleRobot)
```

Then, in `run_simulation` you should use `robot_type(…)` instead of `SimpleRobot` (...) whenever you wish to instantiate a robot. (This will allow us to easily adapt the simulation to run with different robot implementations, which you'll encounter in Problem 6.) Feel free to write whatever helper functions you wish. Again, you may find the provided `get_new_position` method of `Position` helpful.

Problem 6: Running the Simulator

Now, use your simulation to answer some questions about the robots' performance. In order to do this problem, you will be using a Python package called `pylab` (aka `matplotlib`). If you want to learn more about `pylab`, please read this tutorial.

For the questions below, uncomment the function calls provided (at the very end of the problem set) and run the code to generate a plot using pylab, and then answer the corresponding questions underneath the function calls in `ps3.py`. **Save these plots and be ready to show them during your checkoff.**

1. Examine `show_plot_compare_strategies` in `ps3.py`, which takes in the parameters `title`, `x_label`, and `y_label`. It outputs a plot comparing the performance of all types of robots in a 20x20 `RectangularRoom` with 3 units of dirt on each tile and 80% minimum coverage, with a varying number of robots with speed of 1.0 and cleaning capacity of 1. Uncomment the call to `show_plot_compare_strategies`, and answer question #1. Depending on your computer, it may take a few minutes for the plot to show up.

2. Examine `show_plot_room_shape` in ps3.py, which takes in the same parameters as `show_plot_compare_strategies`. This figure compares how long it takes two of each type of robot to clean 80% of `RectangularRooms` with dimensions 10x30, 20x15, 25x12, and 50x6 (notice that the rooms have the same area.) Uncomment the call to `show_plot_room_shape`, and answer question #2. Depending on your computer, it may take a few minutes for the plot to show up.

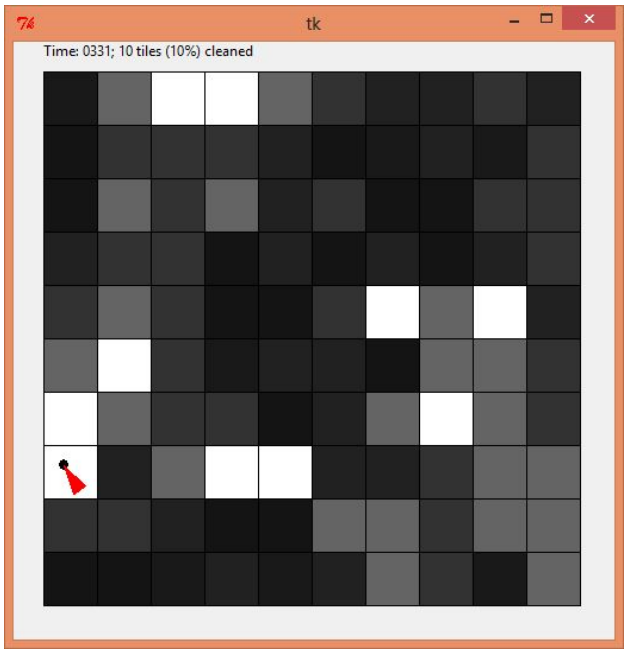
Optional: Visualizing Robot Simulation

We've provided some code to generate animations of your robots as they go about cleaning a room. These animations can also help you debug your simulation by helping you to visually determine when things are going wrong.

Running the Visualization:

1. In your simulation, at the beginning of a trial, do the following to start an animation:
   `anim = RobotVisualization(num_robots, width, height, delay)`
   If you instead want to use `RobotPlusCat`, do the following:
   `anim = CatVisualization(num_robots, width, height, delay)`
2. Pass in parameters appropriate to the trial, of course. `delay` is an optional parameter that is discussed below. This will open a new window to display the animation and draw a picture of the room.
3. Then, during each time-step, after the robot(s) move, do the following to draw a new frame of the animation:
   `anim.update(room, robots)`
   where `room` is a `RectangularRoom` object and robots is a list of `Robot` objects representing the current state of the room and the robots in the room.
4. When the trial is over, call the following method:
   `anim.done()`

The resulting animation will look like this:



Initially, all dirty tiles are marked as black. As the robot cleans each tile by its given capacity, the

color of the tile transits from black to gray to white, with white means completely clean.

The visualization code slows down your simulation so that the animation doesn't zip by too fast (by default, it shows 5 time-steps every second). Naturally, you will want to avoid running the animation code if you are trying to run many trials at once.

Delay:
For purposes of debugging your simulation, you can slow down the animation even further. You can do this by changing the call to `RobotVisualization`, as follows:

```
anim = RobotVisualization(num_robots, width, height, delay)
```

The parameter `delay` specifies how many seconds the program should pause between frames. The default is 0.2 (5 frames/second). You can raise this value to make the animation slower.

For problem 6, we will make calls to `run_simulation()` to get simulation data and plot it. However, you don't want the visualization getting in the way. If you choose to do this visualization exercise, before you get started on problem 6 *and* before you turn your problem set in, **make sure to comment out the visualization code out of run_simulation().**

Hand-In Procedure

1. Save

Save your code in a single file, named `ps3.py`.

2. Test

Run your file to make sure it has no syntax errors. Test your `run_simulation` to make sure that it still works with **all** of the `SimpleRobot, RobotPlusCat` and `BoostedRobot` classes. Make sure that plots are produced when you run the two functions in problem 6 and verify that the results make sense. Make sure all the tests run.

3. Time and Collaboration Info

At the start of your file, in a comment, write down the number of hours (roughly) you spent on the problems, and the names of the people you collaborated with.

4. Submit

Upload `ps3.py` to the 6.00 submission site. You may upload new versions of each file until the 9:00PM deadline, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.