Problem Set 5

Handed out: Wednesday, October 14,, 2015.
Due: 11:59pm, Wednesday, October 21, 2015

Introduction

In this problem set, you will design a simulation and implement a program that uses classes. As with previous problem sets, please don't be discouraged by the apparent length of this assignment. There is quite a bit to read and understand, but most of the problems do not involve writing much code.

Because there are many components to this problem set, we recommend testing your code incrementally (as you implement each method). That way, you will find out immediately if one method is not working as expected.

As always, please do not change any given function signatures.

Using Python's Random Module

You will be using Python's random module, so check out its documentation. Make sure you `import random` at the top of your file. Some useful function calls include:

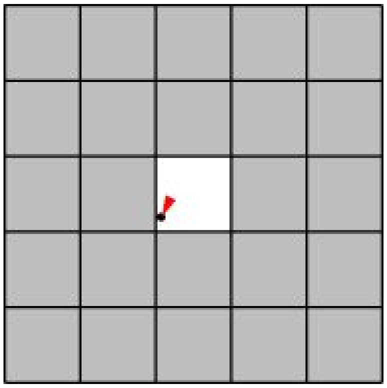
- `random.randint(a, b)` for integer inputs a and b, returns a random integer N such that a <= N <= b
- `random.random()` returns a float N such that 0.0 <= N < 1.0
- `random.seed(0)` starts the pseudorandom number generator Python uses at the same spot so that the sequence of random numbers it produces from different runs of your code will be the same. You may find using this is useful while debugging.

Simulation Overview

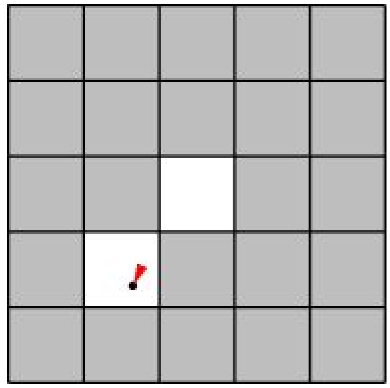
iRobot is a company (started by MIT alumni and faculty) that sells the Roomba vacuuming robot (watch one of the product videos to see these robots in action). Roomba robots move around the floor, cleaning the area they pass over.

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 using two different strategies. 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.

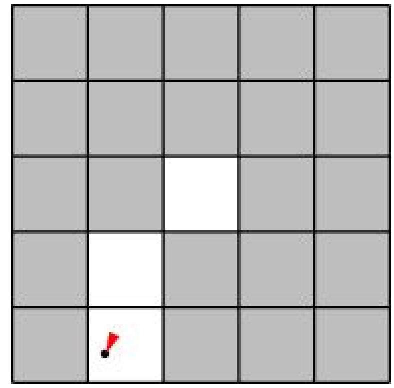
The robot starts out at some random position in the room. Its direction is specified by the angle of motion measured in degrees clockwise from "north." Its position is specified from the lower left corner of the room, which is considered the origin (0.0, 0.0). The illustrations below show the robot's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



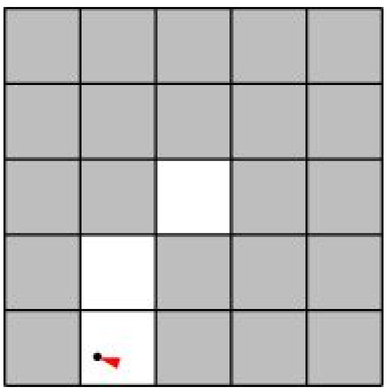
Time *t* = 0
The robot starts at the position (2.1, 2.2) with an angle of 205 degrees (measured clockwise from "north"). The tile that it is on is now clean.



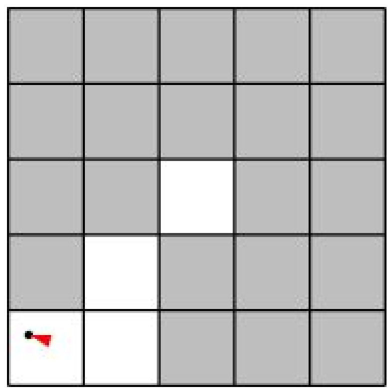
***t* = 1**
The robot has moved 1 unit in the direction it was facing, to the position (1.7, 1.3), cleaning another tile.



***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.

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 tiles. You will first implement the abstract class **RectangularRoom** in problem 1, and then you will implement the subclasses **EmptyRoom** and **FurnishedRoom** in problem 2.
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 **StandardRobot** and **SickRobot** in problems 3 and 4.

The simulation has specified **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*.

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).

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

Part I: The RectangularRoom and Robot classes

The first problem set task is to implement two classes, `RectangularRoom` and `Robot`.
In ps5.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`.

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
Stores the *x*- and *y*-coordinates of a robot in a room.

Read ps5.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.**

Problem 1 - Implementing RectangularRoom and Robot

In this problem, you will implement the `RectangularRoom` and `Robot` abstract classes. An abstract class will never be instantiated, and is instead used as a template for other classes that inherit from it. Abstract classes implement methods that are common to their subclasses. Not all methods for the subclasses are implemented in the abstract classes. In the skeleton code provided, the abstract classes contain methods which should only be implemented in the subclasses. If the comment for the method says “do not change,” please **do not change it**.

Part A: RectangularRoom

Implement the `RectangularRoom` abstract class, according to the specifications given in `ps5.py`. Make sure to think carefully about what kind of data type you want to use to store the floor tiles.

Room Implementation Details:

- Initially, the entire floor is **uniformly** dirty. Each tile should start with an integer amount of dirt, specified by *dirt_amount.*

- 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.

Part B: Robot

Implement the `Robot` abstract class according to the specifications given in `ps5.py`.

Robot Implementation Details:

- **Representation**
  - Each robot has a **position** inside the room. We'll represent the position using an instance of the `Position` **class**, which will have associated coordinates *x* and *y* (**floats** satisfying $0 \le x < w$ and $0 \le y < h$).
  - A robot has a **direction of motion**. We'll represent the direction using a **float** *direction* satisfying $0 \le direction < 360$, 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` and `Robot` classes by implementing their methods in ps5.py. Remember that these classes will never be instantiated; we will only instantiate their subclasses.

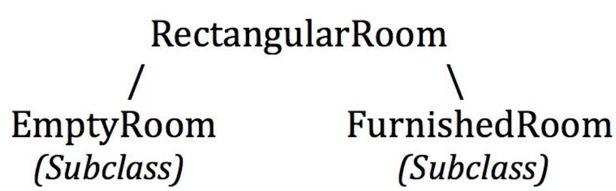
Hints

- A majority of the methods should require only one line of code.
- The `Robot` class and the `RectangularRoom` classes are abstract classes, which means that we will never make an instance of them. Instead, we will instantiate classes that inherit from the abstract classes.

- In the final implementation of these abstract classes, 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`).
- Given a `Position` specified as **floats** (x, y), how can you determine which tile the robot is cleaning? 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).

Problem 2 - Implementing EmptyRoom and FurnishedRoom

In the previous problem, you implemented the `RectangularRoom` class. Now we want to consider additional kinds of rooms: rooms with furniture (`FurnishedRoom`) (thanks IKEA!) and rooms without furniture (`EmptyRoom`). These rooms are implemented in their own classes and have many of the same methods as `RectangularRoom`. Therefore, we'd like to use **inheritance** to reduce the amount of duplicated code by implementing `FurnishedRoom` and `EmptyRoom` as **subclasses** of `RectangularRoom` according to the image below:



Think about how the methods you need to implement differ for the two classes. and how you can use methods already implemented in the parent class `RectangularRoom`. Note: failure to take advantage of inheritance will result in a deduction.

Additionally, be careful in determining whether a position is valid. Recall that in the case of `FurnishedRoom`, a robot cannot be in a position (in a tile) that has furniture.

Finally, in the `FurnishedRoom` class, we have implemented a method to add a rectangular furniture piece to the room for you. **Do not change this method.**

Complete the `EmptyRoom` and `FurnishedRoom` classes by implementing their methods in ps5.py.

Hint : Read the code we have provided carefully to understand how `FurnishedRoom` differs from `EmptyRoom` and `RectangularRoom`. How are the furnished tiles stored?

Problem 3 - StandardRobot 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_and_position_and_clean`.

Ordinarily we would consider putting all the robot's methods in a single class. However, later in this problem set, we'll consider robots with alternate movement strategies, to be implemented as different classes with the same interface. These classes will have a different implementation of update_and_position_and_clean, but are for the most part the same as the original robots. We will again make use of **inheritance** to reduce the amount of duplicated code.

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 StandardRobot class inheriting from it (which contains its own movement strategy).

The movement strategy for StandardRobot 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 by the robot's capacity.
- Otherwise, rotate the robot to be pointing in a random new direction. **Don't clean the current tile or move to a different tile**.

Complete the update_position_and_clean method of StandardRobot 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 4, check that your implementation of StandardRobot works by uncommenting the following line under your implementation of StandardRobot. This will test if your robot moves correctly in an EmptyRoom. When you've checked that your robot moves correctly, **make sure to comment out the** test_robot_movement **line**.

```
test_robot_movement(StandardRobot, EmptyRoom)
```

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

Make sure that as your robot moves around the room, the tiles get lighter (from black to white as shown below) 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 exit the room, it should never clean the tile if it also had to choose a new direction, etc.)

You should also test if your robot moves correctly in a FurnishedRoom. You should not have to change the implementation of update_position_and_clean for this to work. Remember to comment this line out when you are done testing. 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 on a furnished (red) tile or outside of the room. When you've checked that your robot moves correctly, **make sure to comment out the** test_robot_movement **line**.

```
test_robot_movement(StandardRobot, FurnishedRoom)
```

Problem 4 - Implementing SickRobot

Oh no! It turns out iRobot churned out a bad batch of robots. Due to a problem with their vacuums, these robots barf up the dust they love to eat instead of cleaning it off of a tile. They're rather intense barfers, so a robot will end up changing the direction it faces after spewing. You have been asked design a simulation to determine how badly this affects the time it takes a robot to clean a room.

Note: sickness is determined for each timestep. If a robot is sick at one timestep, it may or may not be sick at the next timestep.

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

The movement strategy is as follows:

1. Check if the robot gets sick at this timestep.
2. If the robot gets sick, add the robot's capacity dirt to the tile it is currently on and randomly update its direction.
3. If the robot does not get sick, treat it like `StandardRobot` - have it move to a new position and clean if it can. If it cannot validly move to the next position, instead change its direction.

We have written a method `gets_sick` inside `SickRobot` for you that you should use in order to determine if the robot gets sick. Initially the robot gets sick with probability $p = 0.05$.

Hint: you can still use `room.clean_tile_at_position` to dirty the room, as long as you are using a negative of the robot's capacity.

Test out your new class. Perform a single trial with the new `SickRobot` implementation and watch the visualization to make sure it is doing the right thing.

```
test_robot_movement(SickRobot, EmptyRoom)
```

Part II: Creating and Using the Simulator

Problem 5

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, in Problem 6 you'll comment on the results of your simulation.

Implement `run_simulation(num_robots, speed, capacity, width, height, dirt_amount, min_coverage, num_trials, robot_type)` according to its specification. Use an `EmptyRoom` for this problem.

Your code should:

1. Run the simulation for the specified number of trials (`num_trials`).
2. Simulate the robot(s) cleaning the room until a specified fraction of the room (`min_coverage`) is cleaned.
3. Keep track of the number of time steps (clock ticks) it takes in each trial to reach `min_coverage`.
4. Output the average number of time steps needed to clean the room.

The first six parameters of `run_simulation` should be self-explanatory. For the time being, you should pass in `StandardRobot` for the robot_type parameter, like so:

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

Then, in `run_simulation` you should use `robot_type(…)` instead of `StandardRobot` (...) 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. We have provided the `get_new_position` method of `Position`, which you may find helpful.

For your reference, here are some **approximate** room cleaning times. These times are with a `StandardRobot` and `EmptyRoom`, speed of 1.0 with capacity of 1 (i.e. cleaning 1 unit of dirt per tile per step).

- One robot takes around 310 clock ticks to completely clean a 5x5 room with 3 units of dirt on each tile.
- One robot takes around 575 clock ticks to clean 80% of a 10x10 room.
- One robot takes around 727 clock ticks to clean 90% of a 10x10 room.
- One robot takes around 5478 clock ticks to completely clean a 20x20 room.
- Three robots take around 1861 clock ticks to completely clean a 20x20 room.

These are only intended as guidelines. You may get times slightly different from ours, because there is a random element to the simulation.

You should also check your simulation's output for speeds other than 1.0. One way to do this is to take the above test cases, change the speeds, and make sure the results are sensible.

Hint : Don't forget to reset the necessary variables at the end of each trial.

Problem 6

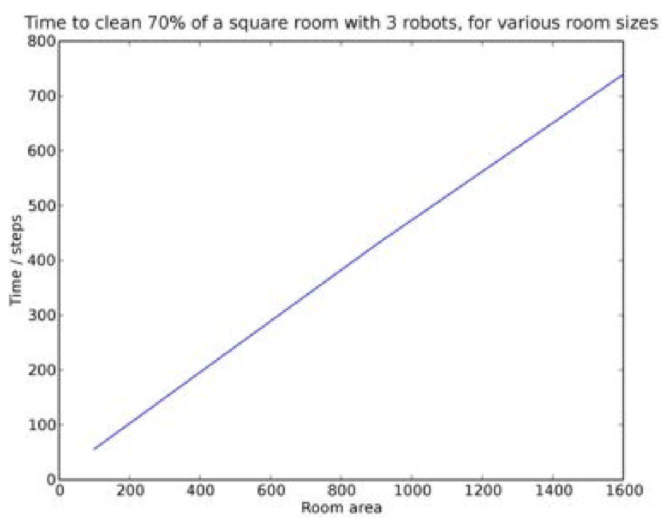
Now, use your simulation to answer some questions about the robots' performance. In order to do this problem, you will be using a Python package called `pylab` (aka `matplotlib`). If you want to learn more about `pylab`, please read this tutorial. Don't worry about learning `pylab` for 6.0001, but it will be covered in 6.0002.

For the questions below, uncomment the function calls provided (they are the last two lines of the problem set) and run the code to generate a plot using pylab, and in the spaces provided for you to make comments, write a sentence or two about the results.

1. Examine `show_plot_compare_strategies` in ps5.py, which takes in the parameters `title`, `x_label`, and `y_label`. It outputs a plot comparing the performance of both types of robots in a 20x20 `EmptyRoom` 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 briefly compare the performance of the two types of robots.

2. Examine `show_plot_room_shape` in ps5.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 `EmptyRoom`s with dimensions 10x30, 20x15, 25x12, and 50x6 (notice that the rooms have the same area.) Uncomment the call to `show_plot_room_shape` and comment briefly on the results in the space provided.

Below is an example of a plot. This plot does not use the same axes that your plots will use; it merely serves as an example of the types of images that the `pylab` package produces.



As you can see, when keeping the number of robots fixed, the time it takes to clean a square room is basically proportional to the area of that room.

Hand-In Procedure

1. Save

Save your code in a single file, named ps5.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 **both** the `StandardRobot` and `SickRobot` classes. (It's common to accidentally break code while refactoring, which is one reason that testing is really important!). Make sure that plots are produced when you run the two functions in problem 5 and verify that the results make sense.

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. For example:

```
# Problem Set 5
# Name:
# Collaborators (Discussion):
# Time:
#
… your code goes here …
```

4. Submit

To submit a file, upload it to Stellar. You may upload new versions of each file until the 11:59pm 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.

Completely Optional: Visualizing robots (Cool and very easy to do. May also be useful for debugging. Comment out before turning in.)

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.

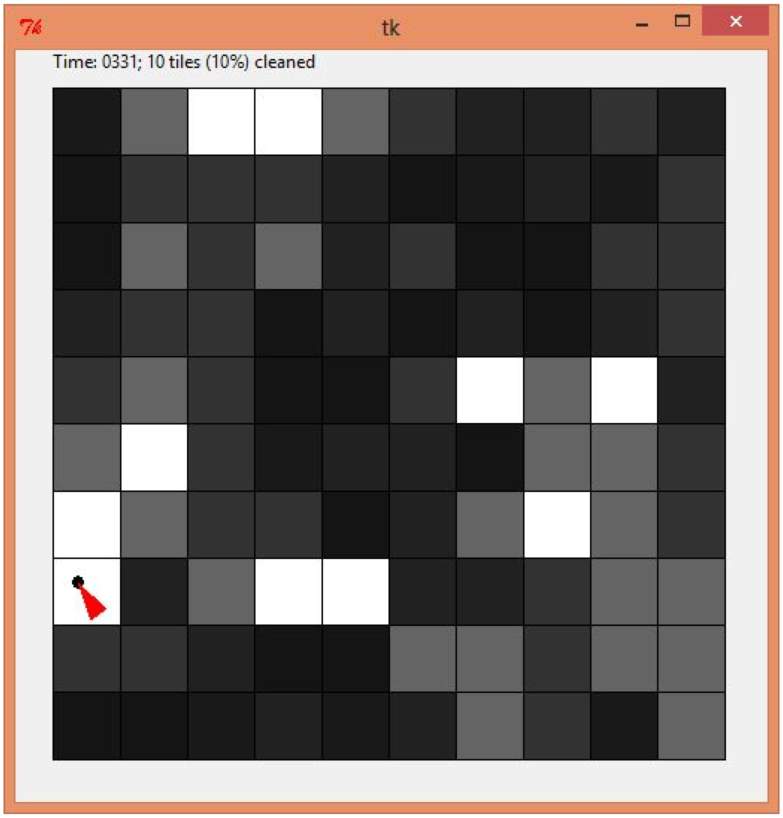
Here's how to run the visualization:

1. In your simulation, at the beginning of a trial, do the following to start an animation:
   `anim = ps5_visualize.RobotVisualization(num_robots, width, height, furniture_tiles, delay)`
   (Pass in parameters appropriate to the trial, of course. Note that delay is an optional

parameter.) This will open a new window to display the animation and draw a picture of the room.

2. 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.
3. When the trial is over, call the following method:
   `anim.done()`

The resulting animation will look like this:



NOTE: 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 (for example, when you are running the full simulation).

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 = ps5_visualize.RobotVisualization(num_robots, width, height,
furniture_tiles, delay)
```

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

For problem 5, 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 5 *and* before you turn your problem set in, make sure to comment out the visualization code out of run_simulation().