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_f16.py*.

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

A) Read the Style Guide

Make sure you consult the Style Guide as we will be taking point deductions for violations (e.g. non-descriptive variable names and uncommented code).

B) 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:

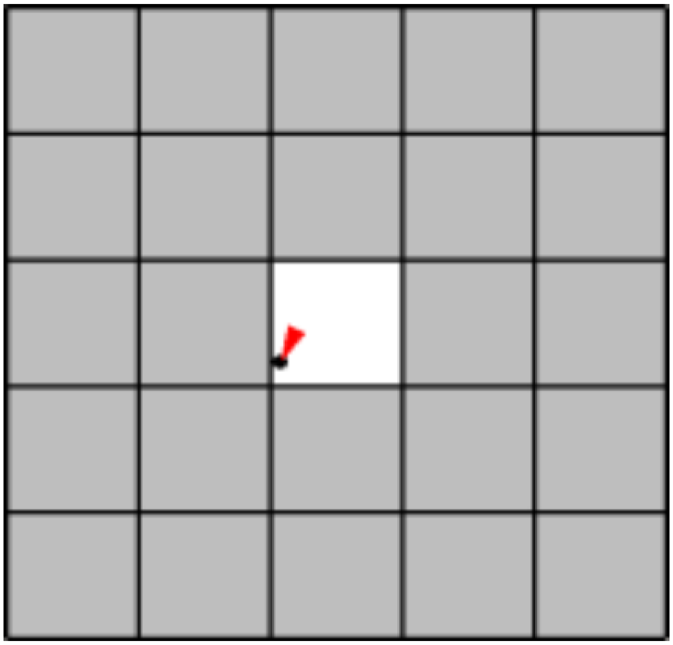
- *random.randint(a, b)* for integer inputs a and b, returns a random integer N such that a <= N <= b
- *random.random()* returns a float N such that 0.0 <= N < 1.0
- *random.seed(0)* starts the pseudorandom number generator Python uses at the same spot so that the sequence of random numbers it produces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

C) Simulation Overview

iRobot is a company 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.

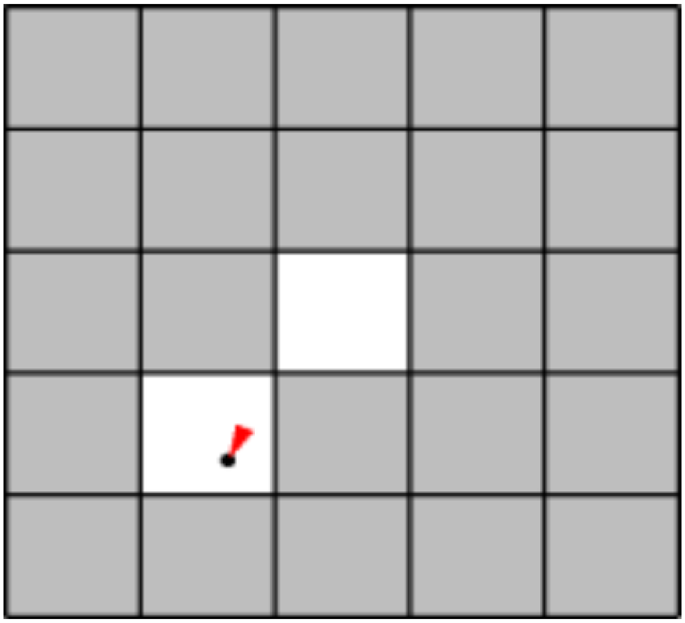
You will code a simulation to compare how much time a group of Roomba-like robots will take to clean the floor of a room using two different strategies. The following simplified model of a single robot moving in a square 5x5 room should give you some intuition about the system we are simulating. A description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



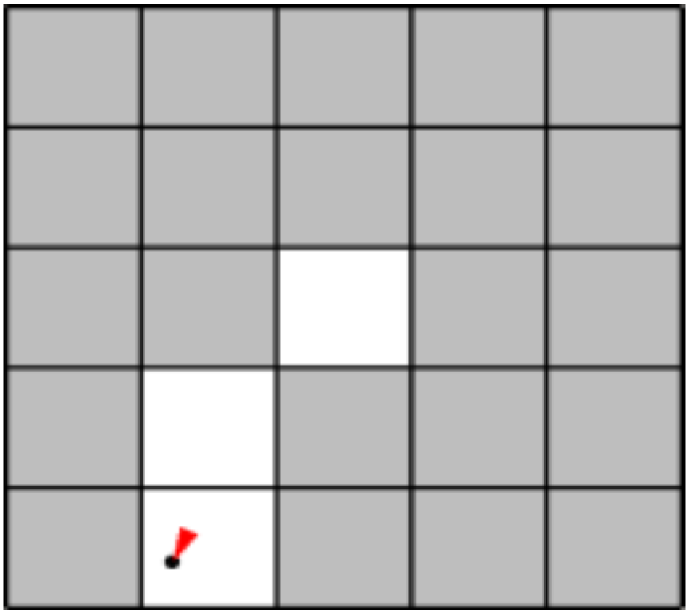
Time *t* = 0

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.



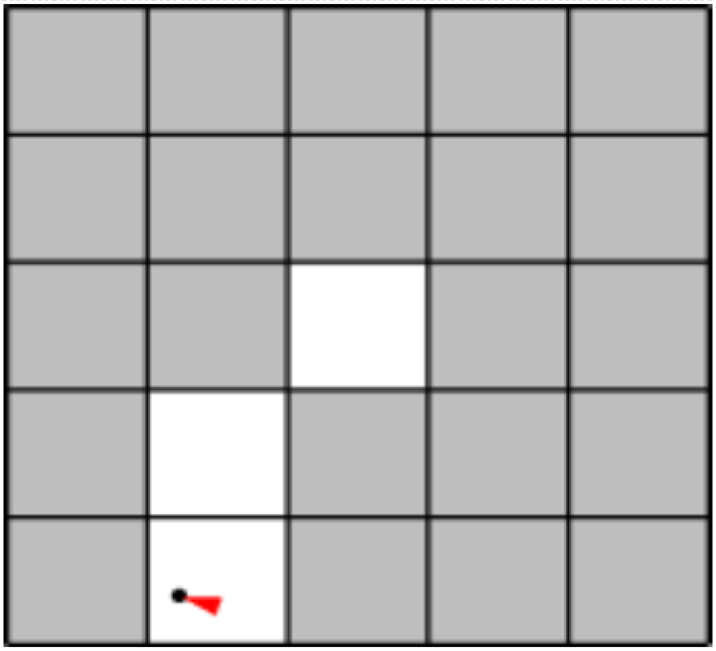
***t* = 1**

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



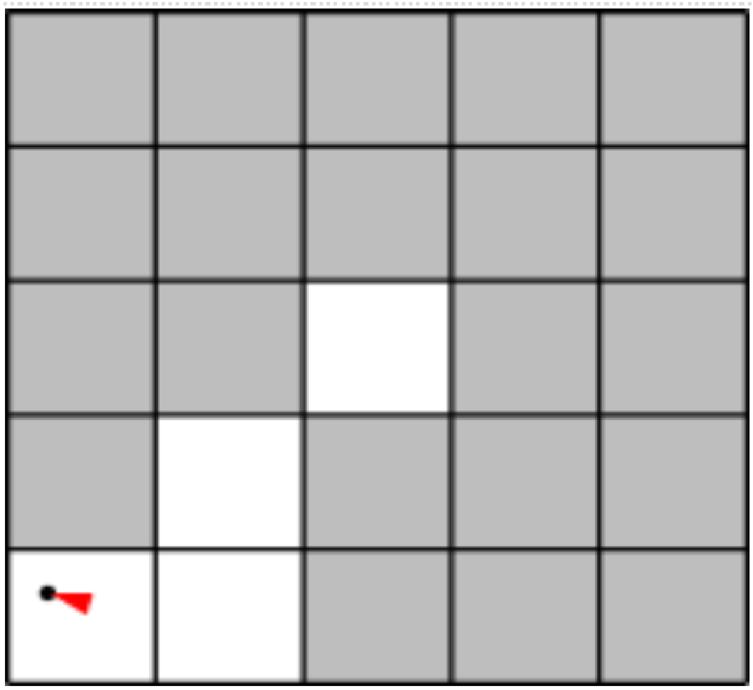
***t* = 2**

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.

D) 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 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 **FaultyRobot** in Problems 3 and 4.

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

E) Helper Code
We have provided two additional files: *ps3_visualize.py* and *ps3_verify_movement27.py*. These Python files contain helper code for testing your code and for visualizing your robot simulation. Do not modify them. If one of these files throws an error, it is because of an error in your code implementation. To test your code, run *ps3_tests_f16.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 two abstract classes, *RectangularRoom* and *Robot*. An abstract class will never be instantiated, and is instead used as a template for other classes that inherit from it. Abstract classes define methods that are shared by their subclasses. These methods can be implement ed in the abstract classes, but they can also be left unimplemented and instead implemented in their subclasses.

In the skeleton code provided, the abstract classes contain 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_test_f16.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*.

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 ≤ x < w and 0 ≤ y < h

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*.

Note that room 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!

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.

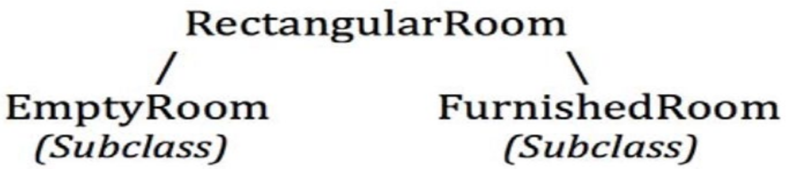
Complete the ***RectangularRoom*** and ***Robot*** abstract classes by implementing their methods according to the pecifications in *ps3.py*. Remember that these classes will never be instantiated; we will only instantiate their subclasses.

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.
- The *Robot* class and the *RectangularRoom* class 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 the m (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). Given a *Position* specified as **floats** (x, y), how can you determine which tile the robot is cleaning?
- 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 abstract class *RectangularRoom* has a *get_random_position* method that may be useful for this.
- Consider using math.floor(x) from the math module instead of int(x) to round floats to whole numbers so that numbers are always rounded down and points are ensured to be inside the room

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 the *add_furniture_to_room* method to add a rectangular furniture piece to the room for you. You do not need to call this method; the provided test code will call it for you. **Do not change this method.**

Complete the *EmptyRoom* and *FurnishedRoom* classes by implementing their methods in ps3.py.

Hints:

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

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 corresponding to that position by the robot's capacity. The position is valid if it is in the room and if it is unfurnished. Do not worry about the robot's path in between the old position and the new position and whether there is furniture in that path.
- 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 *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*:

test_robot_movement(StandardRobot, EmptyRoom)

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**.

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 move to a position outside of 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* by uncommenting the following line:

test_robot_movement(StandardRobot, 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.**

Problem 4: Implementing FaultyRobot
Oh no! It turns out iRobot churned out a bad batch of robots. Due to a problem with their vacuums, these robots randomly forget to clean a tile and will change direction. You have been asked to design a simulation to determine how badly this affects the time it takes a robot to clean a room.

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

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

The movement strategy for a *FaultyRobot* is as follows: 1.

1. Check if the robot is faulty at this timestep.
2. If the robot is faulty, it does not clean the tile it is currently on, and have randomly update its direction.
3. If the robot is not faulty, 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_faulty* inside *FaultyRobot* for you that you should use in order to determine if the robot gets faulty. Initially the robot is faulty with probability p = 0.15. As with *StandardRobot*, 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 *FaultyRobot* implementation and watch the visualization to make sure it is doing the right thing.

test_robot_movement(FaultyRobot, EmptyRoom)

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, 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.

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*.

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!

Your code should:

1. Simulate the robot cleaning process for the specified number of trials (*num_trials*).
2. Simulate the robot(s) cleaning the room until a specified fraction of the room's tiles are clean (*min_coverage*). *min_coverage* is the fraction of clean tiles to total tiles in the room.
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. Again, you may find the provided *get_new_position* method of *Position* helpful.

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

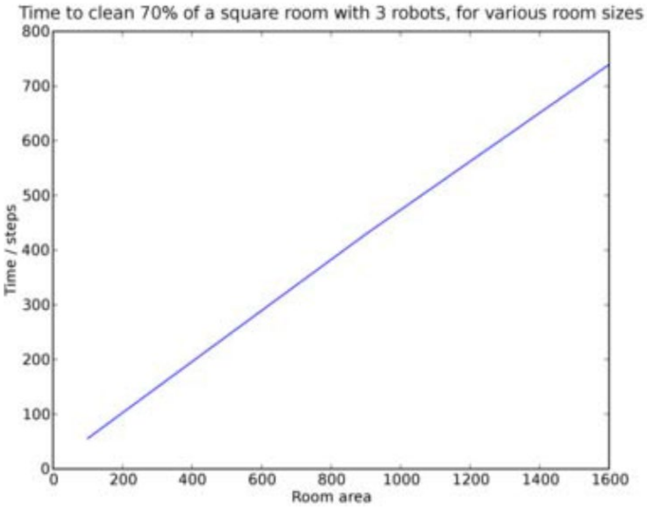
Problem 6: Running the Simulator

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 online
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.

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 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 answer question #1. Depending on your computer, it may take a few seconds 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 com pares how long it takes two of each type of robot to clean 80% of *EmptyRooms* 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 seconds for the plot to show up.

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.

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 = ps3_visualize.RobotVisualization(num_robots, width, height,is_furnished, delay)

   Pass in parameters appropriate to the trial, of course. *is_furnished* is a boolean that should be True if the room is furnished and False otherwise. "*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.

2. Then, during each time-step, after the robot(s) move, do the following to draw a new frame of the animation:

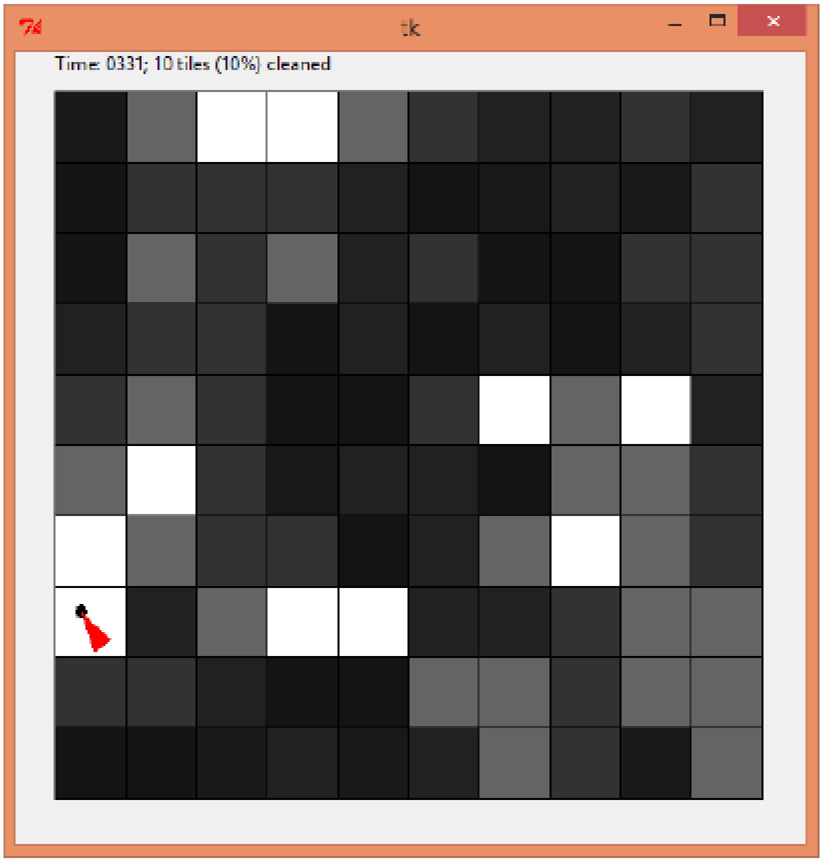
anim.update(room, robots)

   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:



Time: 0331; 10 tiles (10%) cleaned

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).

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 = ps3_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 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 **both** the *StandardRobot* and *FaultyRobot* 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. Make sure all the tests run.

Make sure to also delete any unused or commented out code.

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 3
# Name:
# Collaborators (Discussion):
# Time:
#
… your code goes here …
```