Problem Set 6: Simulating Robots

Handed out: Tuesday, April 2, 2013.
Due: 11:59pm, Tuesday, April 9, 2013.

Introduction

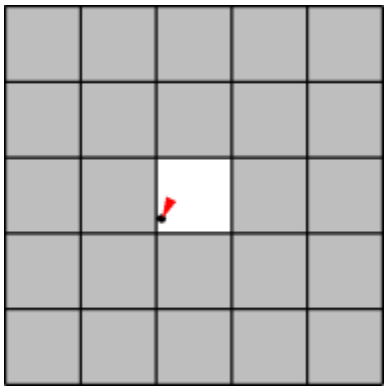
In this problem set you will practice designing a simulation and implementing a program that uses classes.
As with previous problem sets, please don't be discouraged by the apparent length of this assignment. There is quite a bit to read and understand, but most of the problems do not involve writing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

Getting Started

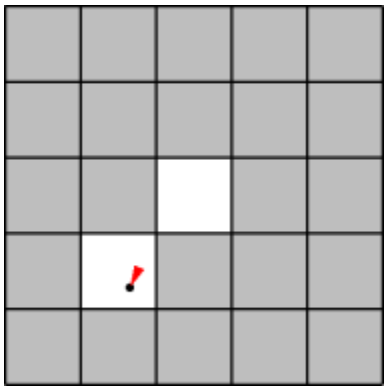
Make sure you have installed numpy and matplotlib successfully, according to the Getting Started document that was released with Problem Set 0.

Simulation Overview

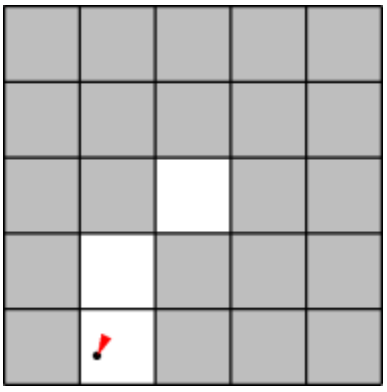
iRobot is a company (started by MIT alumni and faculty) that sells the Roomba vacuuming robot (watch one of the product videos to see these robots in action). Roomba robots move around the floor, cleaning the area they pass over. You will code a simulation to compare how much time a group of Roomba-like 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.
The robot starts out at some random position in the room, and with a random direction of motion. 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).



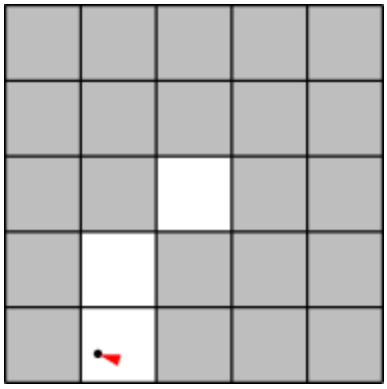
Time *t* = 0
The robot starts at the position (2.1, 2.2) with an angle of 205 degrees (measured clockwise from "north"). The tile that it is on is now clean.



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

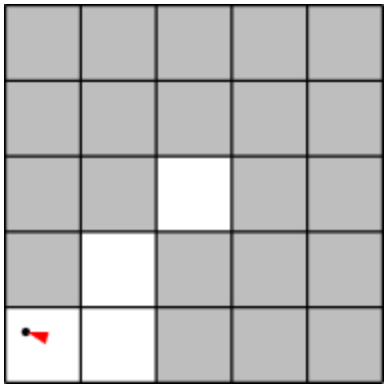


***t* = 2**
The robot has moved 1 unit in the same direction (205 degrees from north), to the position (1.2, 0.4), cleaning another tile.



***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.

Simulation Details

Here are additional details about the simulation model. Read these carefully.

- **Multiple robots**
  In general, there are $N > 0$ robots in the room, where N is given. For simplicity, assume that robots are points and can pass through each other or occupy the same point without interfering.
- **The room**
  The room is rectangular with some integer width w and height h, which are given. Initially the entire floor is dirty. A robot cannot pass through the walls of the room. A robot may not move to a point outside the room.
- **Tiles**
  You will need to keep track of which parts of the floor have been cleaned by the robot(s). We will divide the area of the room into 1x1 tiles (there will be $w * h$ such tiles). When a robot's location is anywhere in a tile, we will consider the entire tile to be cleaned (as in the pictures above). By convention, 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).
- **Starting Conditions**
  Each robot should start at a uniformly random position, and the tiles the robots are located at should be considered cleaned before any time steps have elapsed.
- **Robot motion rules**
  - Each robot has a position inside the room. We'll represent the position using coordinates (x, y) which are floats satisfying $0 \le x < w$ and $0 \le y < h$. In our program we'll use instances of the `Position` class to store these coordinates.
  - A robot has a direction of motion. We'll represent the direction using an integer d satisfying $0 \le d < 360$, which gives an angle in degrees.
  - All robots move at the same speed s, a float, which is given and is constant throughout the simulation. Every time-step, a robot moves in its direction of motion by s units.
  - If a robot detects that it will hit the wall within the time-step, that time step is **instead** spent picking a new direction at random. The robot will

attempt to move in that direction on the next time step, until it reaches another wall.

- **Termination**
  The simulation ends when a specified fraction of the tiles in the room have been cleaned.

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.

Part I: The RectangularRoom and Robot classes

You will need to design two classes to keep track of which parts of the room have been cleaned as well as the position and direction of each robot.
In `ps6.py`, we've provided skeletons for the following two classes, which you will fill in for Problem 1:

RectangularRoom

Represents the space to be cleaned and keeps track of which tiles have been cleaned.

Robot

Stores the position and direction of a robot.

We've also provided a complete implementation of the following class:

Position

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

Read `ps6.py` carefully before starting, so that you understand the provided code and its capabilities.

Problem 1

In this problem you will implement two classes.

For the `RectangularRoom` class, decide what fields you will use and decide how the following operations are to be performed:

- Initializing the object
- Marking an appropriate tile as cleaned when a robot moves to a given position (the function `math.floor` may be useful to you here)
- Determining if a given tile has been cleaned
- Determining how many tiles there are in the room
- Determining how many cleaned tiles there are in the room
- Getting a random position in the room
- Determining if a given position is in the room

For the `Robot` class, decide what fields you will use and decide how the following operations are to be performed:

- Initializing the object
- Accessing the robot's position
- Accessing the robot's direction
- Setting the robot's position
- Setting the robot's direction

Complete the `RectangularRoom` and `Robot` classes by implementing their methods in `ps6.py`.

(Although this problem has many parts, it should not take long once you have chosen how you wish to represent your data. For reasonable representations, *a majority of the methods will require only one line of code*.)

Note:
The `Robot` class is an *abstract* class, which means that we will never make an instance of it. In the final implementation of `Robot`, not all methods will be implemented. Not to worry -- its subclass(es) will implement the method `updatePositionAndClean()`.

Hint:
During debugging, you might want to use `random.seed(0)` so that your results are reproducible.

Part II: Creating and using the simulator

Problem 2

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

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 `updatePositionAndClean` but are for the most part the same as the original robots. Therefore, we'd like to use inheritance to reduce the amount of duplicated code.

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

Complete the `updatePositionAndClean` method of `StandardRobot` to simulate the motion of the robot after a single time-step (as described above in the simulation dynamics).

```
class StandardRobot(Robot):
    """
    A StandardRobot is a Robot with the standard movement strategy.

    At each time-step, a StandardRobot attempts to move in its current
    direction; when it hits a wall, it chooses a new direction randomly.
    """
    def updatePositionAndClean(self):
        """
        Simulate the passage of a single time-step.

        Move the robot to a new position and mark the tile it is on as having
        been cleaned.
        """
```

Before moving on to Problem 3, check that your implementation of Standard Robot works by uncommenting the following line under your implementation of StandardRobot. If you're running Python 2.7, you will need to change the import lines at the top of `ps6.py`. Make sure that as your robot moves around the room, the tiles it traverses switch colors from gray to white.

```
testRobotMovement(StandardRobot, RectangularRoom)
```

When you've checked that your robot moves correctly, make sure to comment out the above line.

Problem 3

In this problem you will write code that runs a complete robot simulation.
Recall that in each trial, the objective is to determine how many time-steps are on average needed before a specified fraction of the room has been cleaned. **Implement the following function:**

```
def runSimulation(num_robots, speed, width, height, min_coverage, num_trials,
                  robot_type):
    """
    Runs NUM_TRIALS trials of the simulation and returns the mean number of
    time-steps needed to clean the fraction MIN_COVERAGE of the room.

    The simulation is run with NUM_ROBOTS robots of type ROBOT_TYPE, each with
    speed SPEED, in a room of dimensions WIDTH x HEIGHT.
    """
```

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

```
avg = runSimulation(10, 1.0, 15, 20, 0.8, 30, StandardRobot)
```

Then, in `runSimulation` 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 5.)
Feel free to write whatever helper functions you wish.
We have provided the `getNewPosition` method of `Position`, which you may find helpful:

```
class Position(object):

    def getNewPosition(self, angle, speed):
        """
        Computes and returns the new Position after a single clock-tick has
        passed, with this object as the current position, and with the
        specified angle and speed.

        Does NOT test whether the returned position fits inside the room.

        angle: float representing angle in degrees, 0 <= angle < 360
        speed: positive float representing speed

        Returns: a Position object representing the new position.
        """
```

For your reference, here are some approximate room cleaning times. These times are with a robot speed of 1.0.

- One robot takes around 150 clock ticks to completely clean a 5x5 room.
- One robot takes around 190 clock ticks to clean 75% of a 10x10 room.
- One robot takes around 310 clock ticks to clean 90% of a 10x10 room.
- One robot takes around 3322 clock ticks to completely clean a 20x20 room.

- Three robots take around 1105 clock ticks to completely clean a 20x20 room.

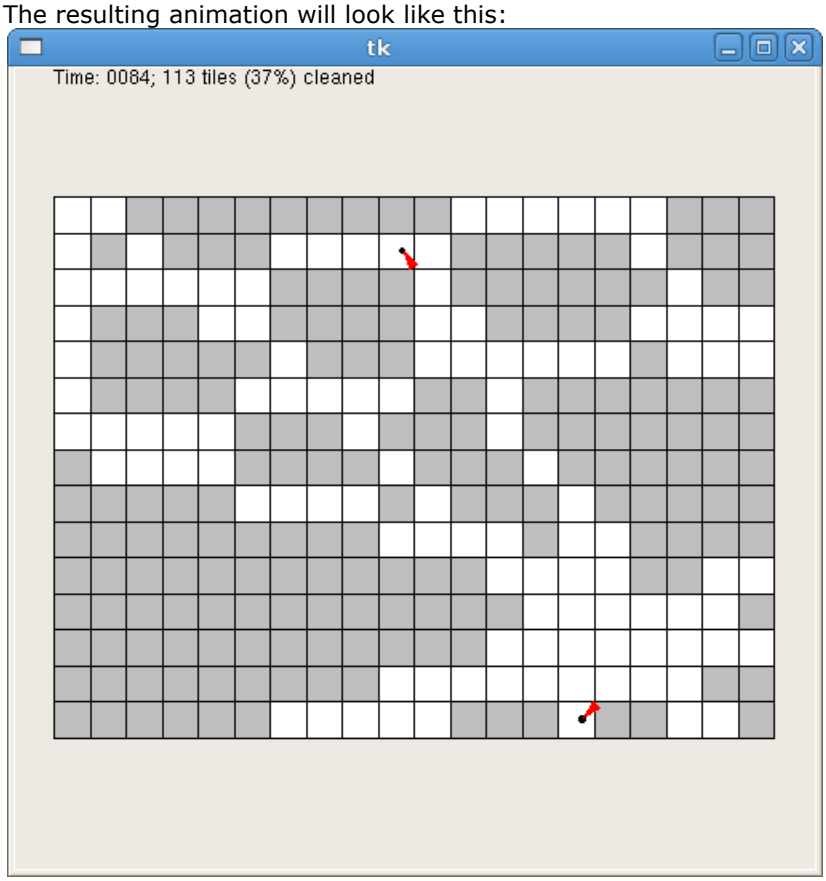
(These are only intended as guidelines. Depending on the exact details of your implementation, you may get times slightly different from ours.)
You should also check your simulation's output for speeds other than 1.0. One way to do this is to take the above test cases, change the speeds, and make sure the results are sensible.

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 = ps6_visualize.RobotVisualization(num_robots, width, height)`
   (Pass in parameters appropriate to the trial, of course.) This will open a new window to display the animation and draw a picture of the room.
2. Then, during *each time-step*, before 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:



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 = ps6_visualize.RobotVisualization(num_robots, width, height, 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 `runSimulation()` 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 the visualization code out of `runSimulation()`.**

Problem 4

iRobot is testing out a new robot design. The proposed new robots differ in that they change direction randomly **after every other time step**, rather than just when they run into walls. You have been asked to design a simulation to determine what effect, if any, this change has on room cleaning times.

Write a new class `RandomWalkRobot` that inherits from `Robot` (like `StandardRobot`) but implements the new movement strategy of changing directions every 2 time steps. `RandomWalkRobot` should have the same interface as `StandardRobot`.

Test out your new class. Perform a single trial with the new `RandomWalkRobot` implementation and watch the visualization to make sure it is doing the right thing. Once you are satisfied, you can call `runSimulation`again, passing `RandomWalkRobot` instead of `StandardRobot`.

Problem 5

Now, use your simulation to answer some questions about the robots' performance.
In order to do this problem, you will be using a Python tool called pylab (aka matplotlib). To learn more about pylab, please read this 6.00 tutorial handout.

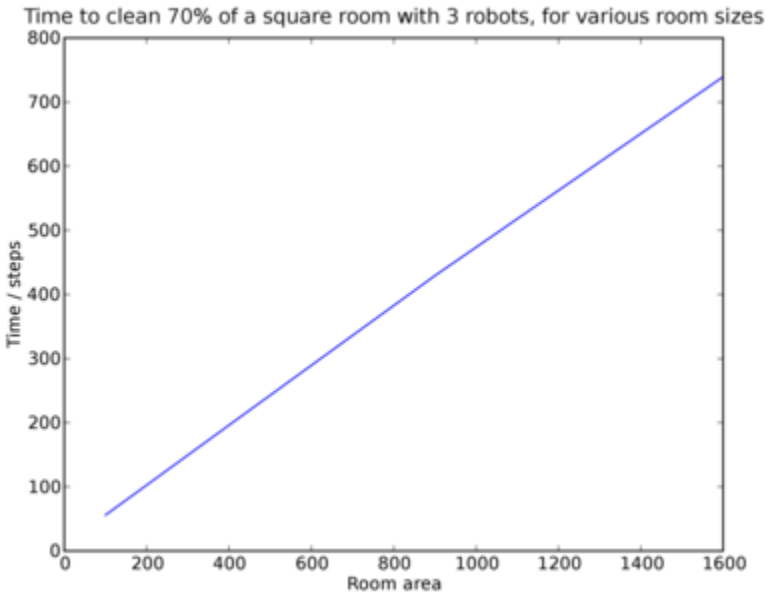
For the questions below, call the given function with the proper arguments to generate a plot using pylab, and in the spaces provided for you to make comments, write up 1) the function call you used, and 2) a short description of the results.

1. Examine **`showPlot1`** in `ps6.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 room with 80% minimum coverage, with a varying number of robots.
   Call `showPlot1` with appropriate arguments and write your function call as a comment under Problem 5, 1a. Additionally, briefly compare the performance of the two types of robots Problem 5, 1b.

2. Examine **`showPlot2`** in `ps6.py`, which takes in the same parameters as `showPlot1`.
   This figure compares how long it takes two of each type of robot to clean 80% of rooms with dimensions 10x30, 20x15, 25x12, and 50x6 (notice that the rooms have the same area.) After understanding the code, write a function call to `showPlot2` that generates

appropriate labels as a comment under Problem 5, 2a. Also comment briefly on the results in the space provided in Problem 5, 2b.

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 **`ps6.py`**.

2. Test

Run your file to make sure it has no syntax errors. Test your `runSimulation` to make sure that it still works with **both** the `StandardRobot` and `RandomWalkRobot` 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 6
# Name: Jane Lee
# Collaborators: John Doe
# Time: 3:30
#
... 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.