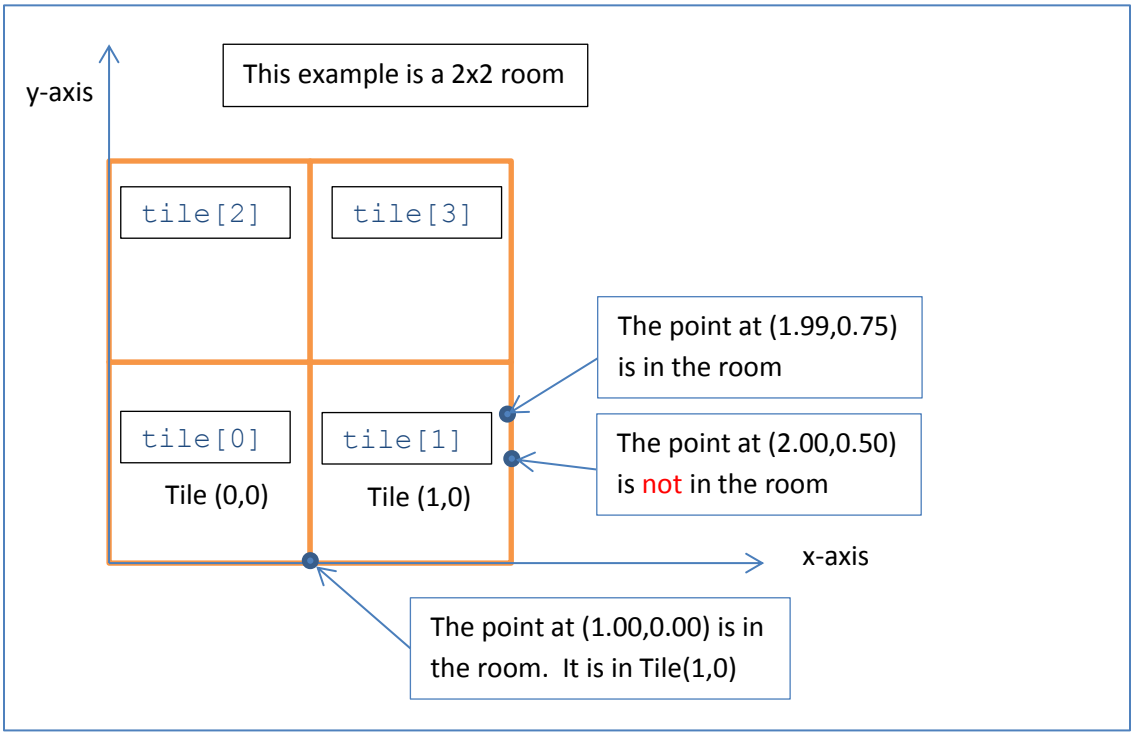
Solution for PS7:

This solution is for RectangularRoom thru Random (including Simulation). It does not include numpy/pylab or Data plots. My code is probably inelegant, but the reason for posting this annotated solution is to show you my reasoning and my method. Unfortunately, I didn't save all the tests that I ran on my code in the IDLE shell or with scripts. A very large part of my solution method involves repeated (and extensive) testing prior to submission to the autograder.

I assume you have the original ps7.py file available, as I won't reproduce the docstrings. A summary of the specifications for the problem set is in a different article in the wiki.

In general, my approach is to read the problem and specifications; outline possible solution approaches; select an approach; detail small chunks (much like they break up the PS's into smaller bits); figure out tests to see if specifications are met; then, write pseudocode, then write code, then, test the code before submission.

I wanted to use an array to represent my tiles, but it appeared that numpy or pylab array methods were not available in the autograder (spent 4 submissions testing various ways of using pylab arrays, none of which worked). Some of the other ways in which you could uniquely identify tiles are: Dictionaries (with tuples as the key), lists (with tuples as items in the list), or lists-of-lists. But I really wanted to do an array with Boolean entries. I chose to address my tiles using a list. A list is basically 1-D objects. By numbering the tiles consecutively (starting at 0) from bottom left to right, then, starting again at the left from the next row, I can do a 1-D to 2-D mapping. The sketch below illustrates how I chose to number and initialize my clean tiles.



Given the problem description of `Tile(m,n)`, where `0≤m<width` and `0≤n<height` (`m,n` are integers), and the limitation of 1D indices in lists, you can see that an index `[m+n*width]` will *uniquely* address every tile in a room (assuming you represent your tiles as a list). I like the simplicity of my approach, but it's not the only way to solve this problem.

You only need a 1-D list of length (width*height) initialized with "False." You can do this using nested loops, a single loop, or a list comprehension. The reason I chose to initialize with "False" is that I had already read the

docstring for `getNumCleanTiles()`: I knew that a Boolean True translates to 1, and a False translates to 0. A simple "sum" method gets me the number of clean tiles. I could have initialized the list using int=1, but I think that makes it more difficult to read. In my code, I called the list `self.tile`; a name `self.Are_Tiles_Clean` would be more descriptive.

```
# === Problem 1
class RectangularRoom(object):
    ''' See original file for the docstring'''
    def __init__(self, width, height):
        '''See original file for the docstring'''
        self.width=width
        self.height=height
##      tempvar=[]
##      for x in range(width):
##          for y in range(height):
##              tempvar.append(False)
##      self.tile=tempvar  #Create array of "False"
        self.tile=[False for i in range(width*height)]
##      Using list comprehension is more elegant
```

The remainder of the RectangularRoom is here:

```
    def cleanTileAtPosition(self, pos):
        """Docstring in original"""
        xloc=int(math.floor(pos.getX()))
        yloc=int(math.floor(pos.getY()))
        self.tile[yloc*self.width+xloc]=True #set tile under "pos" to True

    def isTileCleaned(self, m, n):
        """Docstring in original"""
        return self.tile[n*self.width+m]  #2D address for 1D list

    def getNumTiles(self):
        """Docstring in original"""
        return self.width*self.height

    def getNumCleanedTiles(self):
        """Docstring in original"""
##      sum of list of Booleans returns count of "True" in list
        return sum(self.tile)

    def getRandomPosition(self):
        """Docstring in original"""
        xloc=random.random()*self.width
        yloc=random.random()*self.height
        return Position(xloc,yloc)

    def isPositionInRoom(self, pos):
        """Docstring in original"""
        #raise NotImplementedError
        if pos.getX()<self.width and pos.getX()>=0 \
           and pos.getY()<self.height and pos.getY()>=0:
            return True
        else:
            return False
```

A couple of notes on `getRandomPosition`:

1. The "Position" class is ***any*** (x,y) coordinate, where (x,y) are floating point values.
2. random.random() returns a value bounded by (0,1). However, per the lectures, ***actual*** values of 0 and 1 are highly unlikely (infinitesimally unlikely!). So, `random.random()*width` will span the room width, and also meet the specification $0 \leq x < width$.
3. Don't round the position! (See note 1 above). The Position method has a display method (__str__) which rounds nicely to 2 decimal places and displays for you.

This passed all the tests I could think of (and of course, also passed the autograder suite). (Well, I did misunderstand the room limits at first! But as soon as I saw the error message, I realized the < width boundary!)

Now, on to Robot class. Here is the code:

```
class Robot(object):
    """Docstring in original"""
    def __init__(self, room, speed):
        """Docstring in original"""
        self.room=room
        self.speed=speed
        self.direction=int(360*random.random()) #initialize direction first
        self.position=room.getRandomPosition()  #then, position
        self.room.cleanTileAtPosition(self.position)

    def getRobotPosition(self):
        """Docstring in original"""
        return self.position

    def getRobotDirection(self):
        """Docstring in original"""
        #Note that I chose to implement self.direction in __init__
        #as an integer.  Otherwise, I would return(int(self.direction))
        return self.direction

    def setRobotPosition(self, position):
        """Docstring in original"""
        self.position=position

    def setRobotDirection(self, direction):
        """Docstring in original"""
        self.direction=direction
        #Note that direction is expected to be an integer
        #from 0 to 359 degrees inclusive

    def updatePositionAndClean(self):
        """Docstring in original"""
        raise NotImplementedError # don't change this!
```

Some comments on Robot class:

1. As many have noted, the autograder expects that you initialize your robot direction before you initialize its position. This probably has to do with how they compare a student's output with their code's output.
2. I personally chose to implement all my directions as integers. Without loss of generality, I'm sure I could have initialized as float, but force to integer in getDirection and setDirection. I chose to just treat as integers because the getDirection and setDirection docstrings (which I've deleted here) state that these are integers.

3. Many forgot to initialize the robot with "cleanTileAtPosition(where_the_robot_has_been_placed)" (me too!). The autograder does not always check this in the "Robot" class tests, but this comes back to bite you in the "StandardRobot" and "RandomWalkRobot" tests.

Remember: All robots will ***inherit*** the attributes and methods defined in class Robot(object)—so in the subclasses of Robot, they are initialized ***in the room*** with ***random direction, random (legal) position, speed***, and you can use ***setRobotPosition***, etc. So, StandardRobot and RandomWalkRobot should be very short pieces of code.

Now, here are the StandardRobot and the RandomWalkRobot (I coded both at the same time, then, coded the simulation after). Even though the code is very simple, you really should write pseudocode, to make sure you're matching specifications. Again, I did some artificial tests in the IDLE shell after writing the code, to see that the robots were moving in the correct direction; that the tiles under them were cleaned, and the robots would "bounce" randomly off of walls. At this point, I uncommented the testRobotMovement line, to see test animations, then, re-commented the testRobotMovement line.

A standard robot:

1. Attempts to move.
    a. If it can move, it moves. Then,
    b. Cleans the tile it lands on.
2. If it cannot move, it changes to a new random direction.

A random walk robot:

1. Attempts to move.
    a. If it can move, it moves. Then,
    b. Cleans the tile it lands on.
    c. Then, changes to a new random direction.
2. If it cannot move, it changes to a new random direction.

```
class StandardRobot(Robot):
    """Docstring in original"""
    def updatePositionAndClean(self):
        """Doc string in original"""
        newPos=self.position.getNewPosition(self.direction,self.speed)
        if self.room.isPositionInRoom(newPos):
              self.setRobotPosition(newPos)
              self.room.cleanTileAtPosition(newPos)
        else:
            NewDir=int(random.random()*360)
            self.setRobotDirection(NewDir)

class RandomWalkRobot(Robot):
    """Docstring in original"""
    def updatePositionAndClean(self):
        """Docstring in original"""
        newPos=self.position.getNewPosition(self.direction,self.speed)
        if self.room.isPositionInRoom(newPos):
            self.setRobotPosition(newPos)
            self.room.cleanTileAtPosition(newPos)
            NewDir=int(random.random()*360) #For shorter code, just de-indent
            self.setRobotDirection(NewDir)  #these two lines
        else: #The "else" block duplicates the last two lines in "if", so
            NewDir=int(random.random()*360) #just de-indent where indicated
            self.setRobotDirection(NewDir)  #and comment out from else below.
```

You can see it doesn't take many lines of code to implement either StandardRobot or RandomWalkRobot. The

real effort comes in actually ***understanding*** the problem set requirements, then, figuring out how to solve them. I spent more effort working out the details of my method for storing clean tiles and testing my methods. The rest of the room class and robot classes were very straightforward.

Let's now look at runSimulation: runSimulation is called with:
`runSimulation(num_robots, speed, width, height, min_coverage, num_trials, robot_type)`
where num_robots, all of the same robot_type, and same speed, are placed in a (rectangular) room with width and height. The simulation runs until min_coverage of tiles have been cleaned. The simulation is ***re-run*** for num_trials. The average (mean) time to clean is reported.

Let's write some pseudocode. Actually, let's first write some pre-pseudocode:

1. Need to initialize a room.
2. Need to initialize num_robots into that room.
3. Need to loop UpdatePositionAndClean for all the robots ***until*** min_coverage is reached.
4. Need to loop from 1-4 for num_trials. Remember to re-initialize the room when looping num_trials!
5. Need to report the average time taken over all trials.

Now, pseudocode:

1. TotalTime=0
2. for num_trials:
3. Initialize a (rectangular) room, and initialize num_robots into the room
4. TrialTime=0 (time each individual trial; reset to zero for next trial)
5. while (num_clean_tiles/num_tiles) < min_coverage:
6. for num_robots
7. UpdatePositionAndClean
8. TrialTime+=1
9. TotalTime+=TrialTime (accumulate total of all trials)
10. return (TotalTime/num_trials)

Some notes on the pseudocode: Beware divisions—num_clean_tiles, num_tiles, TotalTime, and num_trials are ***all*** integers. So, remember to "float" your variables correctly (before dividing!).

I tested my runSimulation code against the staff-provided "expected results" case. I created a script to test (in fact, while solving the entire problem set, I had several scripts—but unfortunately, I didn't save them. I can tell you that I tested each module against each published specification). Here is the script that I used to test runSimulation (it was my last test, so it's been saved).

```
from ps7 import *

print "Testing the various test cases"

#mean=runSimulation(num_robots, speed, width, height, min_coverage, \
#               num_trials,robot_type)

mean_1=runSimulation(1,1,5,5,1.0,80,StandardRobot)
print "Should get close to 150 clock ticks: ",mean_1
mean_2=runSimulation(1,1,10,10,.75,60,StandardRobot)
print "Should get close to 190 clock ticks: ",mean_2
mean_3=runSimulation(1,1,10,10,.90,40,StandardRobot)
print "Should get close to 310 clock ticks: ",mean_3
mean_4=runSimulation(1,1,20,20,1.0,25,StandardRobot)
print 'Should get close to 3322 clock ticks: ',mean_4
mean_5=runSimulation(3,1,20,20,1.0,25,StandardRobot)
print 'Should get close to 1105 clock ticks: ',mean_5
```

With that, here's my code for runSimulation (docstrings taken out, but pseudocode left in):

```
def runSimulation(num_robots, speed, width, height, min_coverage, num_trials,
                  robot_type):
    #pseudocode:
    def initRobots(): #helper function to initialize robots into a room
        room_size=RectangularRoom(width,height)
        R=[]
        for i in range(num_robots):
            R.append(robot_type(room_size,speed))
        return [R,room_size] #returns list R (list of robots) and room
    def fracDirty(room,min_coverage): #helper function for min_coverage
        '''Returns False if room is clean enough'''
        if (room.getNumCleanedTiles()/(float(room.getNumTiles())))\
           <min_coverage:
            return True
        else:
            return False
    #initialize room, total=0
    total_steps=0
    for x in range(num_trials):
    # for num_trials
        [R,testroom]=initRobots()
        #Initialize num_robots, TrialTime
        time=0
        while fracDirty(testroom,min_coverage): #while room is dirty
            time+=1                             #increment time
            for j in range(num_robots):         #let each robot update.
                R[j].updatePositionAndClean()   #and.clean
        total_steps+=time #done with trial "x"
    mean_time=float(total_steps)/float(num_trials)
    return mean_time
    # avg_time=total+ divided by num_trials
```

I chose to do a couple of helper functions: The "test for min_coverage" (def fracDirty) (mainly because I didn't want a hugely long conditional going longer than 1 line!", and the robot initialization code (def initRobots).

Note that the use of total_steps and time is somewhat redundant. I wrote it that way, because that's the way I wrote my pseudocode. I don't "change code on the fly," but that's a personal working style. When I write reports, I generally do an outline first, and stick to the outline. So, that's the same approach I take to writing programs.

So, this is my approach to solving PS7 (and a bit on my approach to problem solving in general). I hope you found this annotated solution useful.

hdt, Nov. 26 2012, edx MIT6.00x.