January 22, 2021 Clarke A. Homan. (cahoman@gmail.com)

MIT 6.0002 PS 3 WRITE UP

1. INTRODUCTION

Problem Set 3 for the MIT 6.0002 online course has the student complete a program provided by the course that deals with the simulation of a floor cleaning robot.

The problem set description document describes the goals of the exercise as:

> *"In this problem set, you will design a simulation and implement a [Python] program that uses classes to simulate robot movement."*

The cleaning robot travelled across a virtual rectangular "room", which consisted of M x N square tiles that each had a predetermined amount of dirt. The simulation's room could be empty of furniture or could have a piece of furniture that covered a subset of tiles adjacent to each other. Additionally, the simulation also included a scenario where the robot was randomly faulty; occasionally the robot would "fail" to clean the tile it moves to and simply reset its current direction.

The goal of the of the program is to simulate the robot's behavior given the room type (empty or furnished) and the type of robot (nan-faulty and faulty).

This program utilizes Python 3 (I am currently using Python 3.8.2).

2. SUMMARY DESIGN APPROACHES

Although much of the PS3 program (ps3.py) that was provided by the course was completed, several classes modules were left to the student to complete. Additionally, I created a class structure for the modelling of tiles that extended to both furnished and unfurnished tiles. Their design descriptions are listed below.

A. Completed "Position" Class

i. This class represents a x-y position within a two dimensional "room".

ii. Following methods were implemented:

1. *__init__() constructor*
2. *Get_x() and Get_y() get methods to retrieve object x and y positions*

B. Created "Tile" Class

i. I decided that a tile within a room would be modelled as an object, that would contain tile properties. The original problem did not model tiles this way, but I felt that modelling the tiles as an object that I could then subclass subsequently to model specific tile characteristics would be beneficial. The tile properties

include residual dirt amount remaining on the tile before or after being processed by the robot. The tile's status of being completely cleaned (no residual dirt). Additionally, each tile would contain its center point (x-y coordinates). I also added a sequence number (tile number) when created in case a specific tile needed to be determined.

ii. The tile class included the following methods:

1. *__init__() Constructor that would be passed in the center location (passed in as a 2 element list array. The initial amount of dirt was passed, along with the flag variable "tileCleaned" that is defaulted to "False" when created.*
2. *SetCenter() and getCenter() methods to set and get a tile's center position. Both methods interact with the caller using the Position object being passed.*
3. *SetDirtAmount() and getDirtAmount() methods set and get the tile's current dirt amount property*
4. *RemoveDirtAmount() method attmpts to remove a specified dirt amount from the tile's dirtAmount property. If the current dirt amount of the tile is less than or equal to the amount being removed, then the tile's dirtAmount property is set to 0 and the tile's "tileCleaned" flag property is set to "True". If the current dirt amount of the tile is greater than the amount being removed, the tile's dirtAmount property is decremented accordingly.*
5. *setTileCleane() and getIsTileCleaned() set and get the Boolean value of the tile's "tileCleaned" property*
6. *getTileNumber(): returns the tile's internal tileNum property that was set when the tile was created.*

C. Completed "RectangularRoom" Class

i. All rooms are rectangular specified by a Height and Width parameters. A room consists of a set of tiles stored as a Python dictionary data structure. The tiles dictionary uses the tile's position as expressed as a tuple to uniquely identify the tile within the dictionary. Each dictionary element contains a tile object for the tile, where each tile's method can be accessed from the dictionary element.

ii. A room has the following internal properties: self.width, self.height, self.dirt_amount, self.tiles which is a dictionary of tiles.

iii. RectulangularRoom methods include:

1. *__init__(): Class Constructor. Sets room properties and creates tiles dictionary database.*
2. *Clean_tile_at_position(): Given a current position and robot dirt capacity, this method will determine if the position is valid (within the room) and find the tile within the room whose center position (both x and y) are closest to the position given. Although this determination could have been*

simplified by simply rounding down to the nearest x and y integer tile position, the method used allows the position to be handled as a real. Perhaps this may be over-engineering and could be simplified. Once the tile is selected, the method of tile.removeDirtAmount() is used to remove the robot's dirt capacity from the tile.

3. *Is_tile_cleaned(): Given a x and y (in this case m and n) coordinates, the tile is queried as to whether it has any remaining dirt amount (getDirtAmount) equals 0 and if its "cleaned" flag (getIsTileCleaned) is True. If both are True, then True is returned. Otherwise, False is returned.*
4. *Get_num_cleaned_tiles(): Loops thru all of the time and queries it to see if the tile is cleaned using the getIsTileCleaned() method and the tile's dirt amount is set to 0. This function could be simplified by using the preceding Is_tile_cleaned() method which performs this same validation instead of duplicating the functionality within this function. This should be cleaned up. Once a tile is determined to be cleaned, a running count is incremented. Once all tiles are processed, the running count is returned.*
5. *Is_position_in_room(): Given a Position object, this method determines if the position's coordinates are within the room. Returns the Boolean status of this determination.*
6. *Get_dirt_amount(): Given a m and n coordinates, determine if the coordinates are valid within the room and if so, retrieve the located tile's dirt amount. This function's interface was provided by the course. I would have designed the coordinate datum interface to use the Position object instead of separate m and n components, but I implemented what was asked.*
7. *Get_tile_center(): Given a Position location object, determine if the position is within the room and if so, return the corresponding tile's center position.*
8. *Get_random_direction(): Using random.uniform with a range of 0.0 to 360.0, return a randomly selected direction (float)*
9. *Get_num_tiles(): Not implemented in this class, left to be implemented in subsequent subclasses.*
10. *Is_position_valid(): Not implemented in this class, left to be implemented in subsequent subclasses.*
11. *Get_Random_Position(): Not implemented in this class, left to be implemented in subsequent subclasses.*

D. Completed "Robot" Class

i. Since multiple robots can be set loose to a common room, I decided that the room was modeled as a Robot class variable "_room", accessible to all instances of the Robot. A class variable called "_isRoomDefined" was also used to prevent multiple robot room redefinition. The 1st instantiated robot object would create

the room and set it to the class variable. Subsequent robot object would be prevented from creating duplicate rooms, by inspecting the status of the "_isRoomDefined" class variable set by the 1st robot after creating the room. Class methods of getting (isRoomDefined) and setting (setRoomDefined) were created to provide robots orderly methods of obtaining this status.

ii. Robot methods also include:

1. *__init__() Class Constructor: The class constructor receives the robot's room, speed, capacity and robot number (defaults to 0) as parameters which it sets the object's instance variables to, range and type checking each. Also the robot instance checks to see if the class variable _room has been already populated using the room's isRoomDefined() class method. If not, the robot sets the passed in room as the robot class's _room database. Also the init() method sets the robot to a random position, using the room class's get_random_position() method. It does the same with the robot's current direction using the room class's get_random_direction().*
2. *Get_robot_position(): yields the robot's current position within the room.*
3. *Get_robot_direction(): yields the robot's current direction.*
4. *Set_robot_position(): set's the robot's current position within the room.*
5. *Set_robot_direction(): set's the robot's current direction.*
6. *Update_position_and_clean(): Not implemented in this class, left to be implemented in subsequent subclasses.*

E. Completed EmptyRoom (RectangularRoom) Class

i. An empty room is one without any furniture. It is a subclass of the RectangularRoom class, thus it inherits all of the attributes and methods of its base class.

ii. The EmptyRoom class methods include:

1. *Get_num_tiles(): returns the number of tiles within the room*
2. *Is_position_valid(): given a position object passed to it, this method validates that the position is within the room and returns its boolean status.*
3. *Get_random_position(): overloaded from RectangularRoom, using the random library's random.random and random.random methods generates a random position in floating point between 0.0 and 4.999999… for both width and height position dimensions.*
4. *Is_room_cleaned(): overloaded from RectangularRoom, EmptyRoom specific method that compares the number of cleaned tiles within the room to the overall number of tiles within the room. If the numbers are equal, returns True, otherwise False.*

F. Created FurnishedTile (tile) Class

i. A FurnishedTile is a subclass tile. It adds the class attribute of hasFurniture, which is used to indicate that the tile has furniture placed upon it, thus the robot will not be able to clean it.

ii. The FurnishedTile class methods include:

1. *__init__() Class constructor. The class contructor first instantiates the underlying Tile object with its center position and dirtAmount. Additionally, this method sets the subclasses attribute of whether or not it has furniture on it.*
2. *GetHasFurniture() and setHasFurniture(). These methods will retrieve or set the FurnitureTile's object's hasFurniture attribute.*

G. Completed FurnishedRoom(RectangularRoom) Class

i. FurnishedRoom is a subclass of RectangularRoom. It has all of the properties and methods of its base class, but comprehends the inclusion of a single piece of furniture within the room. The furniture is rectangular and occupies one or more adjacent tiles within the room. The program originally provided modelled the furnished tile identification as a list maintained within the class, where each furnished tile x, y location was added to the furnished tiles list as a tuple. Y design decision was to make tiles a 1st class citizen of the design as a object within the robot's object database and to extend the tile object to comprehend it being furnished. Thus a list of furnished tiles is unnecessary as the tile objects maintain this information themselves.

ii. The FurnishedRoom class methods include:

1. *__init__() Class Constructor. This class constructor for FurnishedRoom was provided by the course pre-implemented and was not supposed to be modified. Due to my tile object design change, I had to update the __init__() constructor with a class specific __init__() constructor that comprehends furnished tile objects. Although this introduces the software maintenance risk of an additional constructor, the benefit of utilizing the FurnishedTile design modification was determined "worth it". The FurnishedTile's __init__() constructor duplicates much of the RectangularRoom __init__() code but introduces the furniture placement parameters of furniture width and height and its lower left corner tile placement. The tiles database maintains which of the tiles are covered with the furniture.*
2. *Is_room_cleaned(). This method returns True if all of the tiles not covered with furniture reporting that their dirt has been cleaned off by the robot. It calls helper methods of get_num_cleaned_tiles() and get_num_tiles, which return the number of cleaned tiles and the total number of tiles in the room that are not furnished.*

3. *Add_furniture_to_room(). As originally provided by the course, this method was supplied pre-implemented with the intention that the student made no modifications. Because of the introduction of FurnishedTile design modification, this method was modified to record the furniture placement within the class instance variables and to call each tile's setHasFurniture method with True for those tiles covered by furniture instead of updating the furniture_tiles[] list for the class that was not used.*
4. *Is_tile_furnished(). This method receives a m, n position, checks to see if the location referred is valid using the _is_position_valid() method and if so, retrieves the referenced tile's HasFurniture property thru the getHasFurniture() method.*
5. *Is_position_furnished(). Receives a position object with arbitrary x and y dimensions, locates the tile being referenced and then retrieves the tile's HasFurniture property thru the getHasFurniture() method.*
6. *Is_position_valid(). Receives a position object and returns if the position is contained within the room, and the tile referenced is unfurnished using the Is_position_furnished() method.*
7. *Get_num_tiles(). Provide total number of valid tiles in room that are unfurnished. Currently the implementation uses a clunky dual loop where each tile position generated within the loop is then queried to be valid (within the room and unfurnished). I believe a more elegant and simple method would be to simply single loop thru the room's tiles database (using Python's dictionary keys() method) feeding each generated key into FurnishedRoom's is_position_valid() method.*
8. *Get_random_position(). FurnishedRoom specific next position generator. The robot calls this method to suggest it's next position choice. The next position choice is tested for being valid (an unfurnished tile). Uses random to generate a position (both x and y) but cycles thru the candidates until an unfurnished position is detected.*

H. Completed StandardRobot(Robot) Class

i. The StandardRobot is a subclass of Robot. This subclass adds the specific method of update_position_and_clean(). I've also added the is_robot_finished() method to the class

ii. The StandardRobot class instance methods include:

1. *update_position_and_clean() method first determines what the next valid position within the room could be. "Moves" to the valid position by updating the robot's current position using the Robot's class method of set_robot_position(). Once position has been updated, cleans the new tile it is at using room's clean_tile_at_position() method. As a diagnostic, I determined if the tile cleaned was previously dirty and print out the fact*

that this robot clean the tile. If the position that was to be moved to was not in the room, then the robot skipped the movement, changed its direction and did not clean any tile at this time quantum.

iii. is_robot_finished(). Calls the Robot's is_room_cleaned() method which returns the Boolean status of the room.

I. Completed FaultyRobot(Robot) Class

i. The FaultyRobot class is another subclass of the superclass Robot, much like the StandardRobot class. A FaultyRobot behaves similarly to a StandardRobot except that occasionally it will fail to clean the destination tile (tile the next move would take it to) and redirect its room direction to a randomly selected direction. The event of failure was modelled using the random.random number generator (producing real numbers between 0.0 and 1.0), where if the randomly generated number was less that the predetermined probability 'p', which in this case was set to 0.15, the event was considered to occur. FaultyRobot includes the static method of set_faulty_probability() which is used to set the FaultyRobot's class probability of failure to 'p'. This method is called externally by the program instantiating FaultyRobots.

ii. The FaultyRobot class instance methods include:

1. *Gets_faulty(). Determine if a FaultyRobot is going to get faulty at a particular timestep. Returns True if the robot gets faulty, otherwise False.*
2. *Update_position_and_clean(). Simulate the passage of a single time-step. Check if the robot gets faulty. If the robot gets faulty, do not clean the current tile and change its direction randomly. If the robot does not get faulty, the robot should behave like StandardRobot at this time-step (checking if it can move to a new position, move there if it can, clean tile. If it can't move, it picks a new direction and stay stationary. Accounts for furniture being at the destination tile (but not in tiles between its current position and destination position).*
3. *Is_robot_finished(). Return whether the robot has cleaned the room.*

3. RESULTS

The results of the program did accomplish what was required. The program successfully simulated the following scenarios:

- Single StandardRobot instance within a M x N EmptyRoom
- Multiple StandardRobot instance within a M x N EmptyRoom
- Single FaultyRobot instance within a M x N EmptyRoom
- Multiple FaultyRobot instance within a M x N EmptyRoom

- Single StandardRobot instance within a M x N FurnishedRoom
- Multiple StandardRobot instance within a M x N FurnishedRoom

- Single FaultyRobot instance within a M x N FurnishedRoom
- Multiple FaultyRobot instance within a M x N FurnishedRoom

4. CONCLUSIONS

I found this exercise interesting and I learned from the experiment. Perhaps the design of the code could be improved, which I noted in these notes. But in the end, the program performed as it was expected. I did not exercise the simulation test code, but I did verify each scenario manually.

One thing I like in particular with respect to the design of the code is the use of modelling the tiles dictionary database using tile objects as the value components of the keys instead of lists, which I feel give it more extensibility and clarity. Subclassing the furnished tile upon the standard tile class allowed the user of furnished tiles to use similar class method interfaces.