The CSC461 Python, Iterator pattern, Strategy pattern

DUE: Thursday, October 31th, at 8AM

The purpose of this assignment is to give you practice with python and the iterator and strategy patterns. This is also to give practice in bi-directional class associations.

Overview

You will be coding a traffic simulator. To keep the project size reasonable, there will only be one street with several blocks. Each block only has left traveling, and right traveling lane. Do NOT worry about cars being in the same position nor will there be a maximum capacity. Each block will be separated by a street light. You will also only add cars at the ends of the street. This is similar to a street that only has pedestrian crossings.

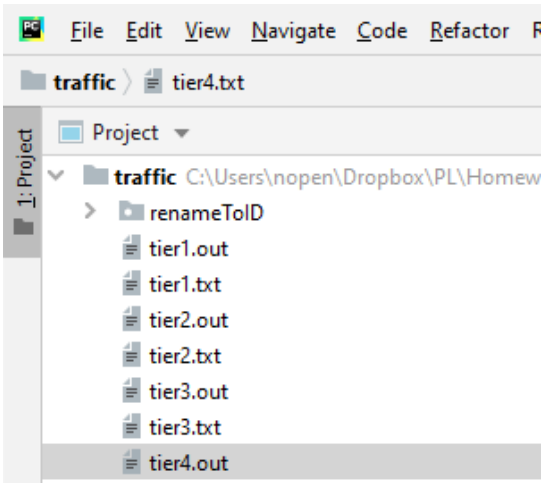
A car will travel the street based on its behavior which must be set using the strategy pattern. After each update, you will output the number of cars in each lane, whether the street lights are on or off, then the locations of the cars. This must be done using two concrete iterators. You must make these yourself using python's built-in support. Under no circumstances should there be anything similar to

```
for i, v in enumerate(self.__block):
    …
```

or

```
for v in iter(self.__block):
    …
```

I am providing you **__init__.py, ColorText.py** (used only to better mark discrepancies)**,** and **MainStarter.py** files that will add difference tests. This must be inside your **lastName_firstName** package. **MainStarter.py** has the structure to run normally and with the tests if any command line arguments are given to **__init__.py**. You must begin your code in the provided **main()** function. The test files are setup to be in the same folder (not inside) as your **lastName_firstName** package like the following:



We will first create the class diagram in class, and you must use this diagram to code your project.

Menu Functionality

The menu must be in the following format as the initial state:

```
1) Show Street
2) Make Road
3) Add Car
4) Show Cars
5) Update without Debug Info
6) Update with Debug Info
0) Quit

Choice:
```

Initially, the road will be composed of a single empty block of length 0.5 miles with a speed limit of 30 mph, and no lights.

There is a new line at the top of the menu.

The menu should be able to handle “e,” -1, “text” by printing “Invalid option”, and then reprint the menu. **Hint**: A *single* try catch block around your menu, plus a message in a specific type of exception, can make this input error and later input error requirements simple to handle.

Show Street

Show street should display the street in the following format:

1. Starting with Entry\Exit
2. Then a block
3. Then a light
4. Then a block (Repeat from 2 until no blocks are left)
5. Ending with Entry\Exit

To print the block, print the number of cars heading up and the number of cars heading down. To print a light, print “On” or “Off” according to its state, followed by the time till it turns. Then reprint the menu.

The initial output with choice 1 should be:

```
Entry:: -0-> <-0- ::Exit
```

The arrows will eventually refer to how many cars and their location which are currently traveling up and down along that one block.

After remaking a road with 3 blocks, with cycles of length 2 minutes and 3.5 minutes, with no cars, the result must be:

```
Entry:: -0-> <-0- ::Off:: -0-> <-0- ::Off:: -0-> <-0- ::Exit
```

This must be done with the iterator pattern with Python syntax (e.g. __iter__(self)). In other words, there should be a for-each loop in this menu option that returns each block and light, in order.

Tag ONCE, the __iter__(self) for this task with GRADING: INTER_STREET

Tag ONCE, the for-each loop for this task with GRADING: LOOP_STREET

Make Road

This remakes the road. It immediately erases the prior road. To make a new road, ask the user to input:

- How many blocks
- For each block with or without a light
    - The length of the block
    - The speed limit in miles per hour
- For each block with a light
    - The minimal length of the time for the light to turn in minutes

For example (you MUST follow this format):

```
Number of blocks: 2

_____Block 0_____
   Length of block in miles: #
   Speed limit of block in mph: #
   Length of cycle in minutes: #

_____Block 1_____
   Length of block in miles: #
   Speed limit of block in mph: #
```

There are 4 spaces in front of the block prompts. Do NOT display the road right after. Input checking will be limited to the following substituting in the value you received into *value*:

- The speed limit must be an integer and >= 5 with the failure response being "Speed limit must be at least 5 mph and a whole number. Got: *value*" **Hint**: Look up isdigit()
- The length of the block must be a positive float >= 0.5 with the failure response being "A block must be at least 0.5 miles. Got: *value*"
- The length of light cycle must be a positive float that is a multiple of 0.5 minutes (hint: % works on floats in Python) with the failure response being "Light cycle must be positive and an increment of 0.5 min. Got: *value*"

These should stop the function as soon as they are noticed. So, if you are making a new road, and there is an error, leaving it in an invalid state is acceptable.

Add Car

To add a block, ask the user to input:

- Which end of the road the car is to be added (0 for top and 1 for bottom)
- Whether the car is a slow driver, normal driver, or fast driverAdd an id for each car starting at 1, and increment automatically for each car added. Note: This ID start is never reset during the run of the program, so since the tests are run in order, later tiers will have IDs starting above 1. For simplicity, there will be instant acceleration and deceleration. You MUST use the following format to add a car:

```
Which end: 0-->left, 1-->right: #
Which type: 0-->slow, 1-->norm, 2-->fast: #
```

Driver Types

You must implement the strategy pattern for the driver behavior. This MUST be a set of functions, although these functions may be stored in a class. You MUST call a function to perform the behavior. The behavior types are:

Nervous driver

A nervous driver drives 5 mph under the speed limit

Normal driver

A nervous driver drives the speed limit

Aggressive driver

An aggressive driver drives 5 mph over the speed limit

Yes, this is a tiny example of the strategy pattern that easily could have been done without it. The point was to give you an easy practice problem.

Tag ONCE, the nervous driver function with GRADING: NERVOUS

Tag ONCE, the normal driver function with GRADING: NORM

Tag ONCE, the nervous driver function with GRADING: AGGRESSIVE

Tag ONCE, the first line in the call chain to set the driver function with GRADING: SET_BEHAVIOR

Tag ONCE, the where this function is called during an update with GRADING: USE_BEHAVIOR

Show Cars

To show the cars:

1. Output the block number starting at 0 at the top
2. Then the cars location for each block
    a. In order they were added
    b. First all cars going up
    c. Then all the cars going down

A car must display in the following format: `ID: location`. This must be done by overriding the `str()` function. The location of a car is its distance from the start of its lane in the block. The overall format must be:

```
_____Block 0_____
Block Length: #.#
Block Speed: #
Left:  id: loc    id: loc     …
Right: id: loc    id: loc     …

_____Block 1_____
Block Length: #.#
Block Speed: #
Left:  id: loc    id: loc     …
Right: id: loc    id: loc     …
…
```

The ID's will be blank if there are no cars. There are 5 _ on both sides of Block. The format for the location should be to 2 decimal places (e.g. 0.22). The format for the length should be to 1 decimal places. The format for the speed should be as an integer. There are 15 spaces reserved to show each car. There are 7 spaces reserved to show Left: and Right:.

This must be done with the iterator pattern, using Python's specific syntax. In other words, there should be a for-each loop in this menu option that returns each block. It should work very similar to:

```
for i, v in enumerate(streetIter):
    print("Block " + str(i) + "--------------------")
    print(v.getCarLocString())
```

Some help: Override the str() function for the lanes/blocks as well as the cars.

Tag ONCE, the __iter__(self) for this task with GRADING: INTER_CAR

Tag ONCE, the for-each loop for this task with GRADING: LOOP_CAR

Update

On an update, update all the cars and lights by thirty seconds. The following tasks must be done **in the same order** presented:

- A light may flip, which restarts its count down.
- At each intersection, see if a car can move to the next block.
    - A car can be removed if it is at the Exit location, and it currently is at or past the end of the block **before the car location update**.
    - A car can move to the next block if it is at a light, this light is on, and it currently is at or past the end of the block **before the car location update**.
    - **Only one** car may move past the intersection, per intersection, in one update.
    - If there is a tie, use the oldest car.
        - AKA check for cars from front to back in the list
- Move the cars forward according to their behavior along the block
    - Move the cars in the same order they were added to the block.
    - If a car tries to move past block when a light when it is off or the update only partially uses its distance, set its position to the length of the block. Essentially, a car's location should never be higher than the length of a block.

If being run with the "`with debug info`" print out the road AND the car locations. In theory, you can use the same functions as the "show road" and "show cars" options. Otherwise, only reprint the menu.

Additional restrictions

- You must display a car by overriding the **str()** function.
- You MUST put your code into a package named your lastName_firstName (lowercase with no prefixes or suffixes.
- There should not be a getter function for ANY list. You must use an iterator. This will cause you to lose all points for the iterator.
- You must use Python's commenting structure for functions. The autoformatting (epytext or restructured) version is up to you.
- The iterators must be done using Python's iterator structure. You must override __iter__() and __next__(). While this will depend on the class diagram, it is *highly* likely you will have two iterators for one class.

Grading Tiers

These tiers start with the simplest tasks, and go to the most involved. You must "reasonably" complete the lower tiers before gaining points for the later tiers. By "reasonably," I can reasonable assume you honestly thought it was complete. OOP line items fall into this category, since you cannot check those on your end.

Submission instructions

1. Check the coding conventions before submission.
2. If anything is NOT working from the following rubric, please put that in the bug section of your main file header.
3. If given a file that is not supposed to be changed, I will be overwriting the file with the original when I grade.
4. List any known bugs in the main file header comment.
5. All of your Python files must use a package named your lastName_firstName (lowercase with no prefixes or suffixes)
6. Delete the out folders, then zip your package folder into ***ONE*** zip folder named your lastName_firstName (lowercase with no prefixes or suffixes). Make sure this matches your package name!

If you are on Linux, make sure you see "hidden folders" and you grab the ".idea" folder. That folder is what actually lists the files included in the project!

7. Submit to D2L. The dropbox will be under the associated topic's content page.
8. *Check* that your submission uploaded properly. No receipt, no submission.

You may upload as many times as you please, but only the last submission will be graded and stored

If you have any trouble with D2L, please email me (with your submission if necessary).

Rubric

All of the following will be graded all or nothing, unless indication by a multilevel score.

Item	Points
Other deductions	
1. Initial Show street	8
Menu working	3
Invalid menu option working	2
"show street" working in default case	3
2. Make road	15
Prompts correct	3
Able to make at least two block and light	3
Able to make several blocks and lights	3
Properly catches invalid length, speed limit, and light, cycle (2 each)	6
3. Light updates	16
Update shows at least one light properly turning on and off	3
Update shows ALL lights properly turning on and off	3
Python Iterator created for blocks	5
Iterator used to display road	5
4. Add car	12
Car can be added to left of road	3
Car can be added to right of road	3
Multiple cars can be added to left of road	3
Multiple cars can be added to right of road	3
5. Show cars	15
Python Iterator created for car list	5
Iterator used to display car list	5
Formatting is correct	3
Able to add multiple cars left and right	2
6. Updating a car with strategy	14
Update works on cars for one block, both directions (2 each)	4
Functions made and properly used for strategy pattern (1 block)	4
Car move correct distance based on strategy(1 block)	3
Update with and without debugging works	3
7. Updating a car with no lights	6
Car able to exit road on one block in either direction (2 each)	6
8. Updating a car with lights	7
Car at minimum, able to move to next block at the end of the block, ignoring light, both ways	3
Car able to move to next block only when light is on, both ways	4
9. Ties	3
Ties favor the oldest car	3
10. Stress test	4
Multiple cars and multiple lights	4
Total	100