The CSC461 Python, Iterator pattern, Strategy pattern

DUE: As stated on D2L

Checklist this time!

The purpose of this assignment is to give you practice with python and the iterator and strategy patterns. This is also to give you practice in bi-directional class associations. This assignment hits D in SOLID,* really hard*. <u>If you do not make your classes reusable, the later tiers will be a nightmare!</u>

We will create a class diagram during lecture. You **must** use this diagram to code your project. This will be tested only on **Python 3.9.** Yes, python 10 was released, after we started on Python this semester!

Overview

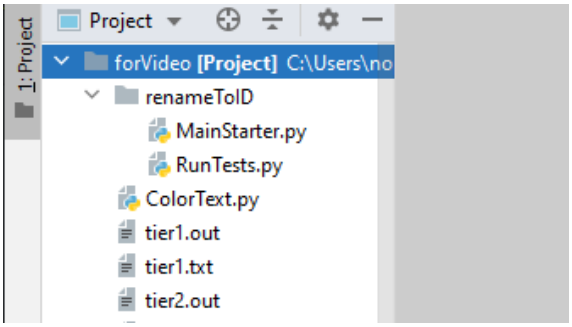
You will be coding a multi station gondola simulator. These do exist: https://en.wikipedia.org/wiki/Mi_Telef%C3%A9rico . To keep the project size reasonable, there will only be one gondola with several stations. Each stretch has one forward traveling, and one reverse traveling cable. Do NOT worry about gondola cars being in the same position nor a maximum capacity. Each stretch will be separated by a station. You will also only add cars at the starting point of the gondola. A car will travel the gondola system based on its behavior, which must be set using the strategy pattern.

Reminder: private/public variable rules still apply in python! You can search for this with regex and self\.[a-zA-Z]+[^()[] ?=

I am providing you with **RunTests.py, ColorText.py** (used only to better mark discrepancies)**,** and **MainStarter.py** files that will add difference tests. **RunTests.py** and **MainStarter.py** must be inside your **lastName_firstName** package. Your grade will be based on these tests. **ColorText.py** is at the same level as your package folder.

MainStarter.py has a **main()** function that the tests use to start up your project. Therefore, you must begin your code in the provided **main()** function. **RunTests.py** works like it did in Java. Run from this file and the tests will run. Run from **MainStarter.py**, and it will run as normal.

The test files and **ColorText.py** are set up to be in the same folder (not inside) as your **lastName_firstName** package like the following:



This is so I can use the same files for everyone when grading.

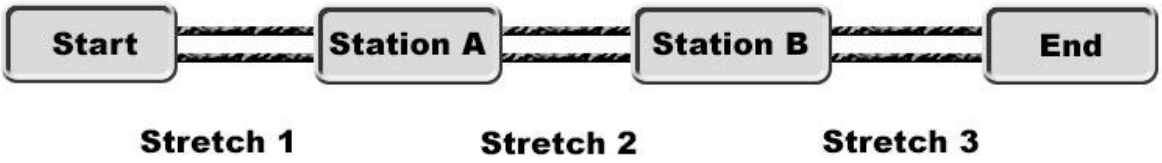
Menu Functionality

The menu must be in the following format as the initial state (there is a space after Choice:>):

```
//new line here
1) Show Gondola System
2) Add Car
3) Update with Debug Info
4) Set Station People
5) Show Station Details
0) Quit

Choice:>
```

The gondola will be composed as shown in the diagram below:



Each stretch is 0.3 miles long, and gondolas move at 0.1 miles per 0.5 minutes. Updates will occur in 0.5 minute intervals. Gondola cars can hold up to 20 people.

The menu must be able to handle integers out of range by printing “Invalid menu option,” and then reprint the menu.

Any other later invalid input should print “Invalid input” and continue.

Hint: A *single* try catch block around your menu, plus a message for specific types of exceptions, can make this input error and later input error requirements simple to handle. **Second hint**: Since this can block your error call stack from displaying, which helps with debugging, commenting it out temporarily is fine!

Show Gondola System

If run immediately after starting, this option should display

```
Time: 0 min
----Start
    Traveling Left:
    Traveling Right:
----Station A
    Traveling Left:
    Traveling Right:
----End
```

Eventually, this will display station information and the cars.

Here is the full format when completed:

1. Print the time (Time: # min), then a new line.
2. Print the station information.
    a. Start with `----`.
    b. Then print the name with a width of 10 characters.
    c. If there is a gondola waiting to go to the left, print "Left: " followed by [<ID>: <time>min <held>/20](<waitCount>).
        i. Substitute <ID> with the car's ID formatted with a left aligned 4 character width.
        ii. Substitute <time> with the time until the doors close formatted to 1 decimal place.
        iii. Substitute <held> with the number of people on the gondola.
        iv. Substitute <waitCount> with the number of gondolas waiting to enter.
        v. End with 3 spaces.
        vi. For example, if those values were 1, 1.5, 0, and 3 the result would be
        `[ ID:1   1.5min 0/20 ](3)`
    d. Repeat with the cars going to the Right on the same line.

3. For the following stretch,
    a. Print 4 spaces, then "Traveling Left: " followed by [<ID>: <location>] for each car, separated by a space. The cars must be in the order added to the cable.
        i. Substitute <ID> with the car's ID formatted with a left aligned 4 character width.
        ii. Substitute <location> with the distance from the last station formatted to 1 decimal place.
        iii. End with a space.
        iv. For example, if those values were 1 and 0.1, the result would be
        `[ ID:1   0.1 ]`
    b. Repeat with "Traveling Right" on the next line.
4. Repeat from 2, until the last station is reached.

Printing from the contents of the system must be done with the iterator pattern. You must make these yourself **using python's built-in support**. This includes using the resulting supported for each loop to confirm it is working. Under no circumstances should there be anything similar to

```
for i, v in enumerate(self.myLine.getAList()):
    …
```
or
```
for v in iter(self.myLine.getAList()):
    …
```
or
```
for v in self.__myStationList:
    …
```
The last one is less obvious, but you are using the array's built-in iterator rather than the one you made. If I put a break point inside your __iter__, it would not be hit in that last case. `for v in self` should be used instead.

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

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

I'll be putting break points at these lines (including inside __iter__ and __next__) to ensure they are hit. ***Some help:*** Override the str() function for the tracks/stations as well as the cars.

Add Car

To add a car, ask the user to input whether the car is small, large, or adapting. Add 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. Therefore, later tiers will have IDs starting above 1.

You MUST use the following format to ask about the behavior type:

```
Which type: 0-->Short, 1-->Long, 2-->Adapting:>
```

If the user input a number other than presented, output "Invalid Input". As an example, after adding one car, option 1 will show

```
Time: 0 min
----Start     Right: [ ID:1   1.0min 0/20 ](0)
    Traveling Left:
    Traveling Right:
----Station A
    Traveling Left:
    Traveling Right:
----End
```

Car Behavior Types

You must implement the strategy pattern for the 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

Short

A car will wait 1 minute at a station.

Long

A car will wait 2 minutes at a station.

Adapting

A car will wait 1 minute at a station if it has <10 people, and will wait 2 minutes if it has 10+ people after unloading\loading.

- Tag ONCE, the cautious function with GRADING: SMALL.
- Tag ONCE, the normal function with GRADING: LARGE.
- Tag ONCE, the rush hour function with GRADING: ADAPTING.
- Tag ONCE, the line that set the function with GRADING: SET_BEHAVIOR.
- Tag ONCE, the line function is used during an update with GRADING: USE_BEHAVIOR.

I'll be putting break points on these to ensure they are hit.

Please note, implementing this task is spread across tiers.

Update

On an update, update all the stations, stretches, and cars by 0.5 minutes.

When updating, a station must have direct access to its adjacent stretches, and a stretch must have direct access to adjacent stations. This is to provide support for a later expansion (and it is also much easier to code long term)

The following tasks must be done **in the same order** presented when updating:

1. Update all stations
    a. If the car has timed out (closes its doors), move it to the next stretch.
    b. If there are any waiting cars, move the next one to the station.
    c. This is true for both directions.
2. Update all stretches
    a. Move cars in the same order they were added to the stretch.
    b. A car is moved to a station if it has reached the end or is past the length of the stretch, **after** its location has been updated.
        - If being moved to a station, move direction to the waiting position (open doors) if the space is available in the direction it is moving.
        - If the space is not currently available, place the car in the waiting list, in the direction it is moving, in the order of arrival.
        - In the case of a start or end station, immediately flip the direction of the car when reached. For example, if a car reached the end station, it will be waiting to go left since there are no stations remaining to the right.
        - This is true for both directions.

Do NOT interleave the stretch and station updates. The following is an example of the state if 3 cars were immediately added, and 4 updates have occurred:

```
Time: 2.0 min
----Start     Right: [ ID:3   1.0min 0/20 ](0)
    Traveling Left:
    Traveling Right: [ ID:2   0.1 ]
----Station A Right: [ ID:1   1.0min 0/20 ](0)
    Traveling Left:
    Traveling Right:
----End
```

Set Station People:

This task sets how many people will be getting on and off at each station (limit to the range 0 to 20), only upon initially entering the station. Remove people first, then add. This is permanent for simplicity. The prompts must follow this format

1. Print the station name
2. Print 4 spaces, then “Getting on:> “
3. Print 4 spaces, then “Getting off:> “
4. Repeat from 1 until all stations are complete

The following is an example:

```
Start
    Getting on:> 5
    Getting off:> 0
Station A
    Getting on:> 5
    Getting off:> 6
End
    Getting on:> 0
    Getting off:> 20
```

Show Station Details

This task shows more details in the station. To show the station details, follow these steps:

1. Print the station string from before.
2. On the next line, output 4 spaces, then “People getting on/off: <on>/<off>” substituting the values for getting on and off.
3. On the next line, output 4 spaces, then “Delayed on...”
4. On the next line, output 6 spaces, then “Left Side: ” followed by the cars using the string creation function developed earlier for the car cable position.
5. On the next line, output 6 spaces, then “Right Side: ” followed by the cars using the string creation function developed earlier for the car cable position.
6. Print a new line.
7. Repeat from step 1 until done.

In an empty system, displaying the station details from with the above would yield

```
----Start
    People getting on/off: 5/0
    Delayed on...
      Left Side:
      Right Side:

----Station A
    People getting on/off: 5/6
    Delayed on...
      Left Side:
      Right Side:

----End
    People getting on/off: 0/20
    Delayed on...
      Left Side:
      Right Side:
```

Assume 4 cars were immediately added, this would be the result:

```
----Start     Right: [ ID:1   0.5min 0/20 ](3)
    People getting on/off: 0/0
    Delayed on...
      Left Side: [ ID:2   0.0 ] [ ID:3   0.0 ] [ ID:4   0.0 ]
      Right Side:

----Station A
    People getting on/off: 0/0
    Delayed on...
      Left Side:
      Right Side:
```

```
----End
     People getting on/off: 0/0
     Delayed on...
       Left Side:
       Right Side:
```

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 station. It should work very similar to

```
for s in stationIter:
    print(s.getStationDetailsString())
    print()
```

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.

Additional restrictions

- You must copy in the checklist and complete it.
- 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.
- You must use the OOP diagram developed in 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 reasonably 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. If given a file that is not supposed to be changed, I will be overwriting the file with the original when I grade.
2. Check that you are not in violation of the additional deductions in the main tab.
3. Complete the checklist and paste into the top of MainStarter.
4. All of your Python files must use a package named your lastName_firstName (lowercase with no prefixes or suffixes).
5. 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!

6. Submit to D2L. The dropbox will be under the associated topic's content page.
7. *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 system	8
Menu working	2
Shows system properly	6
2. Menu framework	10
Prompts correct for remaining options	4
Invalid input working* (-2pt each)	6
3. Add Car	12
Able to add short car to one end	4
Car at station shows properly along with time (-50% for one missing)	4
Car str overridden	3
4. One Short Car Update (one way)	32
Car time starts and decreases properly	3
Car can move to next stretch at the right time	4
Car displays and moves forward correctly on stretch	3
Car enters next station at correct time and location	4
Next station handles time and release properly	4
Car reaches the end station properly	4
Display is correct and done with iterator**	10
5. One Long Car Update (one way)	20
Repeat of prior tier with long car ignoring iterator (50% points each)	12
Strategy pattern used properly**	8
6. Set people and Adapting (one way)	16
Updates station on/off count properly	5
Updates short/long car people count properly	5
Update adapting car properly with strategy**	6
7. Multi short/long car (one way) and station details	20
Waiting list works properly at start	4
Waiting list works properly at later stations	4
Show station details working properly**	8
Later added cars work	4
8. One short car, round trip	18
Able to return to cable at end	6
Returns at correct speed to start (-50% if stalled at mid station)	6
Multiple cycles permitted	6
9. Stress test	14
multiple, varied cars	14
Total	150

*Nothing else has to be working at this point as I'll end any input sequence request with an invalid option for this test case. This is mostly to check your prompt text early.

**Has a tag associated with it.