<div align="center">

National University of Singapore

School of Computing

CS1010X: Programming Methodology

Semester II, 2016/2017

Mission 7

Rogue Train

</div>

Release date: 06 March 2017

Due: 26 March 2017, 23:59

Required Files

- mission07-template.py

- station_info.csv

- train_schedule.csv

- breakdown_events.csv

Background

"Not again!", you groan as the train comes to a sudden halt. The minutes tick by and the train still isn't moving. You're going to be late for your favourite class, CS1010X :(

After you finally get to class, Prof says he has some data on the recent spate of train breakdowns! With your newly developed programming skills, he is confident that you can track down the source of these disruptions and make the Circle Line great again!

This mission consists of **5** tasks.

Learning Objectives

This mission will cover accessing files in Python as well as the `map` and `filter` functions we saw in the lecture. We will also see how data abstraction can help us reason about and manipulate our data.

Mapping and filtering functions are useful since they are expressions and so they can be used inside other expressions. They allow you to decompose a complex operation into a series of smaller operations. They are convenient when the indices of the elements are not important in your function.

Warnings

GovTech has open sourced the code it used for its own analysis of Circle Line disruptions.

You can find their blog post at:

`https://blog.data.gov.sg/how-we-caught-the-circle-line-rogue-train-with-data-79405c86ab6a`.

While this mission is based on the same scenario, the actual algorithms and data structures we use in this mission are different.

Copying their code will not work.

Friendly Reminders

- Ensure that your code is runnable by your tutor by commenting out parts that do not work
- Add comments to your code if you think what you are doing is not clear/obvious!
- Where possible, you should reuse the ADTs and their getters/setters from the previous subtasks. Failure to do so would be treated as breaking abstraction and will be penalized.

Data Files

Train Station Data

The data for all of Singapore's train stations is provided to you in `station_info.csv`.

Each row consists of:

Station Code A short code (e.g. CC1) that uniquely identifies a train station.

Station Name The full station name. This is not unique. E.g. Dhoby Ghaut is an interchange and has 3 rows (NE6, CC1 and NS24).

Line Name The full name of the line that the station belongs to.

Train Schedule Data

`train_schedule.csv` contains records of all train movements.

We call each row the train *schedule event* and it consists of:

Train Code A string that uniquely identifies an individual train.

Is moving 1 if the train is moving, 0 if it is stationary.

Station from The station code where the train is stopped at, or has just departed from.

Station to The station code of the next station the train is headed to.

Date The date on which the breakdown event occurred, in DD/MM/YYYY format.

Time The time at which the breakdown event occurred, in HH:MM format.

Breakdown Data

breakdown_events.csv contains records of all the breakdowns.

It is in the same format as train_schedule.csv and thus each row is a schedule event. However, since a breakdown has occurred at that event we call each row in this file the *breakdown event.*

Python's datetime Module

We will be using Python's datetime module to handle dates and times in this mission. The documentation for the datetime module is available at https://docs.python.org/3/library/datetime.html.

Here are some examples that you might find useful:

```python
# Creating a datetime object which represents 28 Feb 2017, 1.05 pm.
my_datetime = datetime.datetime(2017, 2, 28, 13, 5)
#=> datetime.datetime(2017, 2, 28, 13, 5)

# Retrieving some properties from my_datetime
my_datetime.year
#=> 2017

my_datetime.month
#=> 2

# Printing as a string
my_datetime.ctime()
#=> 'Tue Feb 28 13:05:00 2017'

# Calculating time differences
my_second_datetime = datetime.datetime(2017, 2, 28, 13, 23)
difference = my_second_datetime - my_datetime
difference.total_seconds()
#=> 1080.0
```

Task 1: Introduction to ADTs (18 marks)

Abstract Data Types (ADT) allow us to define data in terms of its behaviour from the point of view of a user of this data.

The tasks and examples in the following section will help to illustrate this.

In this mission, we mostly store the data in tuples. However, the beauty of ADTs is that we can change the way we store the data. As long as we interact with the data through the abstraction, the rest of our code does not need to change.

Demonstration: Station ADT

The station ADT is specified below.

<div align="center">

Station ADT

(station_code, station_name)

Setters

make_station(station_code, station_name) → station

Getters

get_station_code(station) → station_code

get_station_name(station) → station_name

</div>

The Station ADT is provided for you. Here is a sample run.

```
station = make_station('NS9', 'Woodlands')

get_station_code(station)
#=> 'NS9'

get_station_name(station)
#=> 'Woodlands'
```

Task 1a: Train ADT (1 mark)

The train ADT is specified below.

<div align="center">

Train ADT

(train_code,)

Setters

make_train(train_code) → train

Getters

get_train_code(train) → train_code

</div>

We have written the setter. Write the getter. A sample run is provided.

```
train = make_train('Train 0-0')

get_train_code(train)
#=> 'Train 0-0'
```

Task 1b: Line ADT (7 marks)

Design a Line ADT to represent a train line. A Line consists of its name and the sequence of Stations along the line.

The specification of the Line ADT is given below, along with a description of the functions. Decide how to store the data, then implement all the functions.

<div align="center">

Line ADT

Setters

make_line(name, tuple_of_stations) → line

Getters

get_line_name(line) → name

get_line_stations(line) → tuple_of_stations

Related Functions

get_station_by_name(line, station_name) → station or 'Not found'

get_station_by_code(line, station_code) → station or 'Not found'

</div>

- make_line takes in a line name and a tuple of stations, and returns a Line. The sequence of stations in the tuple should remain fixed.

- get_line_name takes in a Line and returns its name.

- get_line_stations takes in a Line and returns a tuple of stations in its original sequence.

- get_station_by_name takes in a Line and a station name, and returns the Station if it exists in the Line. If there is no such Station, then it returns the string 'Not found'.

- get_station_by_code takes in a Line and a station code, and returns the Station if it exists in the Line. If there is no such Station, then it returns the string 'Not found'.

Task 1c: TrainPosition ADT (6 marks)

A TrainPosition represents a Train's position on a Line. Note that the Line object is not stored in the TrainPosition.

TrainPosition ADT

(is_moving, from_station, to_station)

Setters

make_train_position(is_moving, from_station, to_station) → train_position

Getters

get_is_moving(train_position) → True or False

get_direction(line, train_position) → 0 or 1

get_stopped_station(train_position) → station or 'Train is not stationary'

get_previous_station(train_position) → station or 'Train is not moving'

get_next_station(train_position) → station

We have implemented the setter for you. Note that the parameter is_moving should be a **boolean**. Implement the getters. Refer to the ADT specification and the description of the functions below.

- get_is_moving(train_position) takes in a TrainPosition and returns True if the train is moving, False otherwise.

- get_direction(line, train_position) takes in a Line and a TrainPosition and returns an integer indicating which way the train is facing.

  0 means that the train is going along the line in ascending order (e.g. CC1 → CC2) while 1 means it is going in descending order (e.g. CC2 → CC1).

  While station codes contain running integers, you should not rely on it to determine the direction. Instead, you should use the Station sequence stored in the Line object.

- get_stopped_station(train_position) takes in a TrainPosition and returns the Station that the train is currently stopped at. If the train is stationary, it is currently stopped at the from_station. If the train is not stationary, return the string 'Train is not stationary'.

- get_previous_station(train_position) takes in a TrainPosition and returns the Station that the train just departed from. If the train is not moving, return the string 'Train is not moving'.

- get_next_station(train_position) takes in a TrainPosition and returns the next Station that the train will arrive at.

Setters and getters are not always as simple as setting and retrieving attributes from a tuple. In this task, your getters had to perform some logic to decide whether to return the attribute in the tuple or the string ('Train is not stationary/moving').

Working with a clearly defined ADT specification allows you to change the underlying implementation without changing the code that relies on it.

Task 1d: ScheduleEvent ADT (4 marks)

Finally, design a `ScheduleEvent` ADT that represents a single schedule event or breakdown event[1]. It should reference the `Train`, the `TrainPosition` where the breakdown took place, and the time of the occurrence.

<div align="center">

ScheduleEvent ADT

Setters

`make_schedule_event(train, train_position, time)` $\rightarrow$ `schedule_event`

Getters

`get_train(schedule_event)` $\rightarrow$ `train`

`get_train_position(schedule_event)` $\rightarrow$ `train_position`

`get_schedule_time(schedule_event)` $\rightarrow$ `time`

</div>

Here, `time` is a Python `datetime.datetime` object. In this task, you only have to store and retrieve it without modification. In a later task, you will need to manipulate the `datetime.datetime` object.

Task 2: Data Parsing (9 marks)

The `read_csv` helper function is provided to help you with this task.

Task 2a: Train Lines (4 marks)

`station_info.csv` contains basic information about all the train stations in Singapore.

You can assume that stations on the same line will be on consecutive rows, and that the stations are listed in their actual order on the line.

In the template file, you will find a partial implementation of the function `parse_lines` which takes in a filename, reads the CSV file and generates a tuple of lines. In other words, the output of `parse_lines` should look like (`line1, line2, ...`) where `line1` and `line2` are also tuples.

Complete the function by inserting your code where the comments instruct you to. Your code should only be at the indentation level of the comments. Remember to use the getters and setters defined in Task 1.

The template file also has some code which calls your function and extracts the data for the Circle Line. Uncomment those lines after you have finished this task. You will need the `CCL` global variable for the rest of the tasks.

---

[1]Recall from the description of the data files that an event is a schedule event if read from `train_schedule.csv` and a breakdown event if read from `breakdown_events.csv`

Task 2b: Schedule Events (5 marks)

Complete the `parse_events` function. It should return a tuple of `ScheduleEvents`.

Note that the data fields from a CSV file are read as strings by the `read_csv` helper function. Also make sure that you follow the ADT specifications of `Train`, `TrainPosition` and `ScheduleEvent` correctly.

We will use the `parse_events` function to first read the breakdown events. The template file already has the code to do this. Uncomment those lines once you are done with this task.

Task 3: Data Cleaning (8 marks)

Task 3a: Breakdown Events Filtering (5 marks)

Unfortunately, the breakdown events in Task 2 were manually keyed in by an SMRT employee who was also moonlighting as a CS1010X grader and hasn't had enough sleep in the past few weeks.

Some of the data is invalid! The "from" and "to" stations are not even adjacent to each other on the same line.

SMRT has also been conducting their own tests outside operating hours and the breakdown events from those tests are included in the file too. We don't want to include these tests in our analysis, so we have to remove them too.

We only want to keep breakdown events that match the following criteria:

1. "From" and "To" stations are adjacent on the Circle Line.

   Note that while station codes contain running integers, you should not rely on the integer to determine adjacency. Instead, use the `Station` sequence stored in the CCL `Line` object.

2. Breakdown event occurs during operating hours (7am – 11pm), inclusive of 7am and 11pm.

This sounds like a job for `filter`! `filter` takes in a predicate and an iterable, such as a tuple.

`get_valid_events` is provided for you. Your task is to write the predicate function `is_valid` such that `get_valid_events` returns a tuple of valid breakdown events, as defined in the criteria above.

After you are done with your task, uncomment the code provided in the template file so that the filter can be applied and all valid breakdown events can be stored in the global variable `VALID_BD_EVENTS`.

Task 3b: Computing Location IDs (3 marks)

Implement the function `get_location_id` that takes in a `ScheduleEvent` and uses its `TrainPosition` to compute a location ID that represents the location of the train corresponding to the `ScheduleEvent`.

Here's how we will define the location ID:

Each station on the Circle Line corresponds to an integer according to its sequence. For example, CC1 → 0, CC2 → 1. To get the sequence integer, use the Station sequence stored in the CCL global variable from Task 2(a).

If the schedule event was recorded when the train was stationary, the location ID will be the integer which corresponds to the station the train was stopped at.

If the schedule event was recorded when the train was in between two stations, the location ID will be denoted as (0.5 + the lower of the two station numbers). For example, if the schedule event was recorded between Stadium (CC6) and Mountbatten (CC7), the location ID is 5.5.

Task 4: Data Filtering (6 marks)

In this task, we will write some functions to filter the train schedule. Before we do that, we would need to read the entire train schedule data. Uncomment the code in the template file so that the entire train schedule is read using the parse_events function from Task 2(b) and stored in the global variable FULL_SCHEDULE. Note that this operation might take some time.

Task 4a: Filter by Time (2 marks)

Implement the function get_schedules_at_time that takes in a tuple of ScheduleEvents and a Python datetime.datetime.

Your function should return a tuple of ScheduleEvents which occur at the given time.

Task 4b: Filter by Location (2 marks)

Implement the function get_schedules_near_loc_id that takes in a tuple of ScheduleEvents and a location ID as defined in Task 3(b).

Your function should return a tuple of ScheduleEvents whose location IDs are a maximum of 0.5 away from the given location ID.

Task 4c: Filter by Time and Location (2 marks)

Let's put the two functions from Tasks 4(a) and 4(b) together. Implement the function get_rogue_schedules that takes in a tuple of ScheduleEvents, a Python datetime.datetime and a location ID.

Your function should return a tuple of the ScheduleEvents which occur at the given time and whose location IDs are a maximum of 0.5 away from the given location ID.

Your code must make use of the two functions written earlier. **ZERO** marks will be awarded for solutions that do not call get_schedules_at_time and get_schedules_near_loc_id, or call and discard the results without using them.

Task 5: Finding the Rogue Train (10 marks)

We've now finished designing our ADTs, reading in the data, removing invalid entries and writing functions to help filter the train schedule data. What was it all for?

SMRT and GovTech have a hypothesis that the breakdowns are caused by a rogue train.

Let's find that rogue train!!

Strategy

We will examine each of the breakdown events, one at a time, and see which other trains were nearby at the time when the breakdown event occurred. We will assign a "blame score" of 1 to a train each time it is found to be near a breakdown event. Once we have the total "blame score" of all the candidate trains, we will verify the rogue train hypothesis. If the rogue train hypothesis holds, we would go ahead and find the rogue train.

In order to help record and manage the "blame score" for each train, we have provided a Scorer ADT that you would use throughout Task 5.

Scorer ADT

You do not need to understand how it works, but you do need to know how to use it.

- `make_scorer` returns a Python dictionary. It has already been called for you. In your code, use the global variable `SCORER` which is an instance of the Scorer ADT.

- `blame_train` takes in the `SCORER` object and a train code. It increments the blame score of the given train (identified by the train code) by 1.

- `get_blame_scores` takes in the `SCORER` object and returns a nested tuple of train codes and their corresponding blame scores.

Here is a sample run.

```
SCORER = make_scorer()    # this is already done for you in the template

# Blame a bunch of trains for the breakdowns
blame_train(SCORER, 'Train A')
blame_train(SCORER, 'Train B')
blame_train(SCORER, 'Train C')
blame_train(SCORER, 'Train A')
blame_train(SCORER, 'Train A')

# Retrieve their blame scores.
get_blame_scores(SCORER)
#=> (('Train A', 3), ('Train B', 1), ('Train C', 1))
```

Task 5a: Calculate Blame Scores (4 marks)

Write a function `calculate_blame` that takes in the full train schedule tuple and the tuple of valid breakdown events. The function then goes through all the valid breakdown

events, finds which trains were in the vicinity at the time of the breakdown, and assigns 1 point of blame to each nearby train.

Step by step:

1. For each valid breakdown event, filter train schedule for trains which were nearby at the time when the breakdown event occurred.

2. For each train in the vicinity, blame the train using the `blame_train` function of the already-defined scorer ADT. Remember to use the global `SCORER` variable here.

Note that your function should not return anything. Only update the global `SCORER` variable via `blame_train` function calls.

Hint: Make use of functions you have written in previous tasks, as well as the ADTs already defined. Also remember that with `get_rogue_schedules`, there can be multiple `ScheduleEvent` involving the same train.

Once you are done with this task, uncomment the code in the template file below the `calculate_blame` function definition so that the total blame score for all the candidate rogue trains is calculated using `calculate_blame`.

Task 5b: Find Max Score (2 marks)

Write a function `find_max_score` that takes in the `SCORER` object. Using the `map` function, and Python's built-in `max` function, **return** the maximum score.

You should not write your own loops for this task. However, you can use indexing to get the blame score from each (`'Train Code'`, `blame_score`) tuple.

Task 5c: Verify the Rogue Train Hypothesis (2 marks)

Uncomment the code provided under Task 5c in the template file to view the blame scores for all candidate rogue trains. Do you think the hypothesis that the breakdown events are caused by a single rogue train holds? Explain.

Note that ZERO marks would be awarded for answers without reasonable explanation.

Task 5d: Find the Rogue Train (2 marks)

Now that we know the maximum "blame score" and have also verified the rogue train hypothesis, we can finally find the rogue train (programmatically).

Write a function `find_rogue_train` that takes in the `SCORER` object and the maximum score found in Task 5(b). The function should **return** the <u>train code</u> of the rogue train whose score matches the maximum score.

Conclusion

Congratulations! You have found the rogue train! SMRT has pulled the train from service and you can now be on time for CS1010X!

Till the next breakdown . . .