National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2024/2025

Mission 4
Rogue Train

Release date: 7th October 2024
Due: 20th October 2024, 23:59

Required Files

- mission04-template.py
- IO.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, CS1010S :(

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 **three** tasks.

Learning Objectives

This mission will cover the `map` and `filter` functions we saw in the lecture. We will also be making use of ADT structures defined in the last mission to help us reason about and manipulate our data.

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.

Due to the extensive size of the dataset in `IO.py`, it is recommended to avoid opening the `IO.py` file directly as it may lead to computer performance issues or crashes.

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.**
- Use only `tuple` as your compound data structure. No `list`, `set`, `dict` etc.

Abstract Data Types (ADT)

The following consists of the ADTs that have been implemented for you. Sample executions of `Station`, `Train` and `Line` are provided in Recitation 6.

Station

A `Station` consists of its code and name.

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`

- `make_station` takes in a station code and station name, and returns a `Station`.
- `get_station_code` takes in a `Station` and returns its code.
- `get_station_name` takes in a `Station` and returns its name.

Train

A `Train` consists of its code.

Train ADT

(train_code,)

Setters

`make_train(train_code)` → `train`

Getters

`get_train_code(train)` → `train_code`

- `make_train` takes in a train code, and returns a `Train`.
- `get_train_code` takes in a `Train` and returns its code.

Line

A `Line` consists of its name and the sequence of Stations along the line.

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 None`

`get_station_by_code(line, station_code)` → `station or None`

`get_station_position(line, station_code)` → `Number`

- `make_line` takes in a line name and a tuple of stations (in order), and returns a `Line`.
- `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 `None`.
- `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 `None`.
- `get_station_position` takes in a `Line` and a station code, and returns the index of `Station` with the given code inside the `Line`. The index starts from 0. If there is no such `Station`, then it returns -1.

ScheduleEvent

A `ScheduleEvent` represents an incident and consists of a train, its position and the time of the event.

ScheduleEvent ADT

Setters

`make_schedule_event(train, train_position, time)` → `schedule_event`

Getters

`get_train(schedule_event)` → `train`

`get_train_position(schedule_event)` → `train_position`

`get_schedule_time(schedule_event)` → `time`

Here, `time` is a Python `datetime.datetime` object.

- `make_schedule_event` takes in a train, its position and the time, returning a `ScheduleEvent`.
- `get_train` takes in a `ScheduleEvent` and returns the train involved.
- `get_train_position` takes in a `ScheduleEvent` and returns the position of the incident.
- `get_schedule_time` takes in a `ScheduleEvent` and returns the time of the event.

A sample execution for this ADT is given in the "Breakdown Data" section.

Dataset

All datasets have been imported into `IO.py` or the template file as variables for you to use, in the appropriate ADTs. You do not need to worry about the data importation (unless we say otherwise!). This section is for you to understand the data you are working with. A comma-separated value (CSV) file is a technique to store data, where (as the name implies) the values of the table are separated with commas, in place of borders.

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.

This data has been imported and parsed in `IO.py`, stored in the variable `LINES`. A sample run is shown below:

```
>>> LINES = (
  ('North South Line', (('NS1', 'Jurong East'), ...)),
  ('East West Line', (('EW1', 'Pasir Ris'), ...,)),
  # Line ADT as elements of LINES =>
  ...
)

>>> LINES[0]
('North South Line', (('NS1', 'Jurong East'), ..., ))
# Line ADT => (Line Name, tuple of stations)

>>> get_line_name(LINES[0])
'North South Line'
# Line Name

>>> get_line_stations(LINES[0])
(('NS1', 'Jurong East'), ('NS2', 'Bukit Batok'), ...,
  ('NS28', 'Marina South Pier'))
# Tuple of stations => (Station ADT, Station ADT, ...)

>>> get_station_by_code(LINES[0], 'NS1')
('NS1', 'Jurong East')
# Station ADT => (Station Code, Station Name)
```

a

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 "True" if the train is moving, "False" 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 position of the train was recorded, in DD/MM/YYYY format.

Time The time at which the position of the train was recorded, in HH:MM format.

Similar to the train stations, `train_schedule.csv` stores the events, and it is parsed using `parse_events_in_line` (how this function is implemented is described in Task 2). A sample execution is similar to that of `BD_EVENTS`.

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*.

As with before, the events recorded in `breakdown_events.csv` has been imported and stored in the variable `BD_EVENTS`. A sample execution is given below:

```
>>> BD_EVENTS = (
  (('TRAIN 1-1',),
  (True, ('CC13', 'Serangoon'), ('CC12', 'Bartley')),
  datetime.datetime(2017, 1, 6, 6, 44)),
  ...
)

# Retrieve the first breakdown event, which is a ScheduleEvent ADT
>>> BD_EVENTS[0]
(('TRAIN 1-1',),
(True, ('CC13', 'Serangoon'), ('CC12', 'Bartley')),
datetime.datetime(2017, 1, 6, 6, 44)
)

>>> get_train(BD_EVENTS[0])
('TRAIN 1-1',)
# Train ADT => (train_code,)

>>> get_train_position(BD_EVENTS[0])
(True, ('CC13', 'Serangoon'), ('CC12', 'Bartley'))
# TrainPosition ADT => (Boolean, Station ADT, Station ADT)

>>> get_schedule_time(BD_EVENTS[0])
datetime.datetime(2017, 1, 6, 6, 44)
# datetime.datetime object
```

Task 1: TrainPosition ADT (6 marks)

Unfortunately, the data manager missed uploading the TrainPosition ADT, causing Prof's system to fail. However, you quickly realise that it can be represented similar to the other ADTs.

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 None`

`get_previous_station(train_position)` → `station or None`

`get_next_station(train_position)` → `station`

We have implemented the setter for you, where you may assume that the two stations are different. 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 and the given function to get the position of the `Station` in the `Line`.
- `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 `None`.
- `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 `None`.
- `get_next_station(train_position)` takes in a `TrainPosition` and returns the next `Station` that the train will arrive at.

Sample Execution:

```
>>> test_station1 = make_station('CC2', 'Bras Basah')
>>> test_station2 = make_station('CC3', 'Esplanade')
>>> test_train_position1 = make_train_position(False, test_station1, test_stat
>>> get_is_moving(test_train_position1)
False
>>> get_direction(test_train_position1)
0
>>> print(get_stopped_station(test_train_position1))
('CC2', 'Bras Basah')
>>> print(get_previous_station(test_train_position1))
None
>>> print(get_next_station(test_train_position1))
('CC3', 'Esplanade')
```

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 `None`.

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

Task 2: Data Filtering (6 marks)

Having defined the `TrainPosition` ADT, the profs have written a few functions to load the data.

The events are recorded in CSV files. `parse_events_in_line` takes in a CSV file as input, and turns each of the entries of the file (the schedule) into the neat and beautiful `ScheduleEvent` ADT and stores them in a tuple.

`is_valid_event_in_line` takes in a breakdown event and line as input, and filters events according to two criteria: the involved stations are adjacent (no stations between them) and the event is recorded during operating hours (7am - 11pm).

What do all these functions have in common? They depend on the functions you defined in the `TrainPosition` ADT to work properly. **DO NOT EDIT THE FUNCTIONS PROVIDED!**

In the next task, we will write some functions to filter the train schedule. Given the large size of dataset, please be prepared for longer runtime when executing your code.

Task 2a: Filter by Time (2 marks)

Implement the function `get_schedules_at_time` that takes in a tuple of `ScheduleEvents` and a Python `datetime.datetime`. You may assume that the `ScheduleEvent` given in `train_schedule` belongs to the correct `Line` being evaluated currently. Hence, you do not need to check if the `ScheduleEvent` belongs to the current `Line`.

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

Task 2b: Filter by Location (2 marks)

You are given the function `get_location_id_in_line` that takes in a `ScheduleEvent` and `Line` and uses the `ScheduleEvent`'s `TrainPosition` to find the location ID, that represents where the `TrainPosition` is on the `Line`.

E.g. `get_location_id_in_line(event, CCL)` will return the location id of the event. This assumes that the event belongs to the `CCL` line.

Here's how we will define the location ID:

Each station on the given Line corresponds to an integer according to its sequence. In fact, we have defined this function before. For example, CC1 $\rightarrow$ 0, CC2 $\rightarrow$ 1 in `CCL` global variable defined before.

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.

Implement the function `get_schedules_near_loc_id_in_line` that takes in a tuple of `ScheduleEvents` and a location ID.

Your function should return a tuple of `ScheduleEvents` whose positions are a maximum of 0.5 away from the given position in the given `Line`.

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

Let's put the two functions from Tasks 1(a) and 1(b) together. Implement the function `get_rogue_schedules_in_line` that takes in a tuple of `ScheduleEvents`, a Python `datetime.datetime` and a position.

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_in_line`, or call and discard the results without using them.

Task 3: Finding the Rogue Train (8 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 4.

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.
- `blame_train` takes in the `Scorer` 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` 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 3a: Calculate Blame Scores (4 marks)

Write a function `calculate_blame_in_line` that takes in the full train schedule tuple, the tuple of valid breakdown events, the current `Line` and a given `Scorer`. 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. Also remember not to double count the trains, as **one train can only be blamed once for one event**.

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 of each event, blame the train using the `blame_train` function of the already-defined scorer ADT.

The function should return the modified Scorer. Note that the `Scorer` should only be modified using the given Scorer ADT.

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_in_line`, 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_in_line` function definition so that the total blame score for all the candidate rogue trains is calculated using `calculate_blame_in_line`.

Task 3b: Find Max Score (2 marks)

Write a function `find_max_score` that takes in a `Scorer`. 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 3c: 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` and the maximum score found in Task 2(b). The function should **return** the train code 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 CS1010S!

Till the next breakdown . . .