Mission_MRT
The Rogue Train

Required Files

- `mission_mrt-template.py`
- `station_info.csv`
- `breakdown_events.csv`
- `train_schedule.csv`

Background

It’s Wednesday morning... "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 Computnig Lesson. :(

After you finally get to class, Ms Foo and Mr Tan say that they have some data on the recent spate of train breakdowns! With your newly developed Python programming skills, they are confident that you can track down the source of these disruptions and make the Circle Line great again!

The full mission consists of **5** tasks, but you will only do 4 of them now. (Total 40 marks)

Learning Objectives

This mission will cover accessing files in Python and we will also see how data abstraction can help us reason about and manipulate our data. We will use tuple to organise all the information.

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

- 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. You must learn not to break the abstraction !

Introduction to ADTs (20 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.

Task 1a: Train ADT (1 mark)

The train ADT is specified below.

Train ADT
(train_code,)

Setters
`make_train(train_code) → train`

Getters
`get_train_code(train) → train_code`

We have written the setter `make_train`. Write the getter `get_train_code`.
A sample run is provided as follows:

```
test_train1 = make_train ('Train 0-0')

get_train_code (test_train1)
=> 'Train 0-0'
```

Task 1b: Station ADT (2 mark)

The station ADT is specified below.

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`

We have written the setter `make_station` for the Station ADT.
Write the getters `get_station_code` and `get_station_name`.
Here is a sample run :

```
test_station1 = make_station ('CC2 ', 'Bras Basah ')
```

```
get_station_code (test_station1)
=> 'CC2'

get_station_name (test_station1)
=> 'Bras Basah'
```

Another two `test_station` are also created :

```
test_station2 = make_station('CC3', 'Esplanade')
test_station3 = make_station('CC4', 'Promenade')
```

Task 1c: MRT Line ADT (7 marks)

Now that you have seen how the train and station ADT look like, try to design a MRT 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 MRT Line ADT is given below, along with a description of the functions. Decide how to store the data, then implement all the functions.

MRT 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
```

- `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 `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`.

Write the setter `make_line` for the MRT Line ADT.
Write the getters `get_line_name` and `get_line_stations`.
Write the two related functions `get_station_by_name` and `get_station_by_code`.

```
test_line = make_line('Circle Line', (test_station1, test_station2,
                                      test_station3))
```

Here is a sample run :

```
get_line_name(test_line)
=> 'Circle Line'

get_line_stations(test_line)
=> (('CC2','Bras Basah'),('CC3','Esplanade'),('CC4','Promenade'))

get_station_by_name(test_line, 'Bras Basah')
=> ('CC2', 'Bras Basah')

get_station_by_code(test_line, 'CC4')
=> ('CC4', 'Promenade')
```

You should realise by now that the '`test_line`' contains tuples within a tuple and a simple `print_line` function will help to print the tuple '`test_line`' in a more readable format:

```
Details about the Line:
                       Circle Line
                       *     ('CC2', 'Bras Basah')
                       *     ('CC3', 'Esplanade')
                       *     ('CC4', 'Promenade')
```

Task 1d: Train_Position ADT (6 marks)

A `Train_Position` represents a train's position on a `Line`. Note that the `Line` object is not stored in the `Train_Position`.

Train_Position 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_stopped_station(train_position) → station or None`
`get_previous_station(train_position) → station or None`
`get_next_station(train_position) → station`
`get_direction(line, train_position) → 0 or 1`

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 `Train_Position` and returns `True` if the train is moving, `False` otherwise.

- `get_stopped_station(train_position)` takes in a `Train_Position` 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 `Train_Position` 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 `Train_Position` and returns the next `Station` that the train will arrive at.
- `get_direction(line,train_position)` takes in a `Line` and a `Train_Position` 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.

Hint : If `tup = [‘a’,’b’,’c’]`, then `tup.index[‘b’]` will return the index which is `1`.

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 1e: Event ADT (4 marks)

Finally, design an Event ADT to represent two kinds of event : a schedule event or a breakdown event. An Event should reference the `Train`, the `Train_Position` where the breakdown took place, and the `time` of the occurrence.

With the Event ADT, we will be able to read from the data file `train_schedule.`csv for all the schedule events and from the data file `breakdown_events.csv` for all the breakdown events.

Event ADT
(train, train_position, time)

```
Setters
make_event(train, train_position, time) → event
Getters
get_train(event) → train
get_train_position(event) → train_position
get_event_time(event) → time
```

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

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`

You must include ‘`from datetime import *`’ or ‘`import datetime`’ in your code to use the Python’s `datetime` module. [Note : for the template you are using, we have included ‘`from datetime import *`’ in Task 2d.

Here are some examples that you might find useful:

Creating a `datetime` object which represents 28 Feb 2017 , 1.05 pm :

start_datetime = `datetime` (2017 , 2, 28 , 13 , 5)

start_datetime
=> datetime . `datetime` (2017 , 2, 28 , 13 , 5)

Printing a `datetime` object :

print(start_datetime)
=> 2017-02-28 13:05:00

start_datetime . `ctime ()`
=> 'Tue Feb 28 13:05:00 2017 '

Extracting information from a `datetime` object :

start_datetime . `year`
=> 2017

start_datetime . `month`
=> 2

start_datetime . `hour`
=> 13

start_datetime . `minute`
=> 5

Calculating time difference between two `datetime` objects :

end_datetime = `datetime` (2017 , 2, 28 , 13 , 23)

difference = end_datetime - start_datetime
difference . `total_seconds ()`
=> 1080 .0

Converting a string to `datetime` object :

```
datetime.strptime("06/01/2017", "%d/%m/%Y")
=> datetime.datetime(2017, 1, 6, 0, 0)
```

```
print(datetime.strptime("06/01/2017", "%d/%m/%Y"))
=> 2017-01-06 00:00:00
```

Combining date and time string before converting to `datetime` object :

```
date_time_string = "06/01/2017"+","+"06:44"
```

```
datetime.strptime(date_time_string, "%d/%m/%Y,%H:%M")
=> datetime.datetime(2017, 1, 6, 6, 44)
```

```
print(datetime.strptime(date_time_string, "%d/%m/%Y,%H:%M"))

=> 2017-01-06 06:44:00
```

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 "`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.

Breakdown Data

`breakdown_events.csv` contains records of all the breakdowns.

We call each row in this file the breakdown 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.

Task 2: Data Parsing (12 marks)

The `read_csv` helper function is provided to help you with this task. This helper function will read from the `station_info.csv` and return a tuple of tuples. Each tuple will contain

(`station_code`, `station_name`, `line_name`)

Task 2a: Print Station_Info Details (1 marks)

The output tuple from the `read_csv` helper function could be long and difficult to read. Write a `print_station_details` to print the tuple in an organised manner like the following :

```
Details about the Stations file:
('station_code', 'station_name', 'line_name')
('NS1', 'Jurong East', 'North South Line')
('NS2', 'Bukit Batok', 'North South Line')
('NS3', 'Bukit Gombak', 'North South Line')
('NS4', 'Choa Chu Kang', 'North South Line')
('NS5', 'Yew Tee', 'North South Line')
...
```

Task 2b: Train Lines (4 marks)

The `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.

Task 2c: Stations along a Lines (2 marks)

Continue from Task 2b, you will now write the code to extracts the data for the `Circle Line`. This looks like a filtering job! Uncomment those lines after you have finished this task. You will need the `CCL` global variable for the rest of the tasks.

Task 2d: Breakdown 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, including the Booleans are also read as string “True” and “False”. Also make sure that you follow the ADT specifications of `Train`, `TrainPosition` and `ScheduleEvent` correctly.

We will use the `parse_events` function to 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 (6 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 Computing Tuition Teacher 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: Examine the Breakdown Events (1 mark)

Use the previously written `print_event` function to display the `VALID_BD_EVENTS`

Task 4: Data Filtering (2 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 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 `Events` and a Python `datetime`. Your function should return a tuple of `Events` which occur at the given time.

Task 4b: Filter by Location (Omitted)

Task 4c: Filter by Time and Location (Omitted)

What have we done so far ?

We’ve designed our ADTs, we are able to read the data from the CSV files, we have removed all the invalid entries and have written a function 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 done that for all the breakdown cases, we will sum up the “blame score” of all the candidate trains.

If indeed there was a rogue train, as according to the hypothesis, then we will hunt down the train with the highest “blame score”.

To be continued …