Rail Fare Prices - Coursework 1 (Individual) - #001

Estimating rail fares from information about the location and connectivity of train stations.

Contents

---

1 Foreword and Summary

Please **READ THIS ASSIGNMENT CAREFULLY**. If you have any questions about the assignment, please post them on the Q&A forum in Moodle or email the module leaders. Do not post samples of your code concerning the assignment to public forums such as Moodle.

This assignment asks you to write some code for loading a dataset containing information about a railway network, and analysing fare prices between stations on the network. We will describe how the code must behave, but it is up to you to fill in the implementation. Besides this, you will also need to create some tests, and demonstrate your ability to use version control (via git).

The exercise will be semi-automatically marked, so it is *very* important that your solution adheres to the correct file and folder name convention and structure, as defined in the rubric below. An otherwise valid solution that doesn't work with our marking tool will **not** be given credit. We have provided a tool to check the structure of your submission file, details of which can be found below.

For this assignment, you can only use:

- The Python standard library,
- numpy,
- matplotlib, and
- pytest.

Your code should work with ***Python 3.10 or newer***. Some modules in the standard library that might be useful to keep in mind are:

- csv,
- string.

This document is laid out as follows:

- First, we provide the setting of the problem that you will be solving.
- Next, we outline the functionality that should be implemented, and any other tasks you should complete.
- Finally, to assist you in creating a good solution, we state the marking scheme we will use.

1.1 Requesting Clarification and Reporting Errors

Contact the module leader directly if you believe that there is a problem with the starting code, or the assignment text in general. Questions regarding the clarity of the instructions are also allowed - we recommend those happen on the Q&A forum in Moodle. However, please keep in mind the previous note that you should not post *samples of your code* to Moodle.

If you are reporting an error in the provided code, please provide information about your operating system version, Python version, and an exact sequence of steps to reproduce the error.

Any corrections or clarifications made after the initial release of this coursework will be written in the errata section and announced.

1.2 Errata

1.2.1 2023-11-01

Corrected typo and ambiguity in the name of the sample data file provided, paragraph 3, section: Goal.

- "`uk_network.csv`" was used in the opening sentence as if it was a file name, where it should have read "UK network".
- The sample data file was incorrectly referenced as "`uk_network.csv`" rather than "`uk_stations.csv`" in the following sentences.

Corrected text now reads:

Our marking script will be testing the functionality of your code over the UK network (provided) and other similar networks. You are welcome to use the `uk_stations.csv` file in your test cases, or write your own custom networks for your tests - in the latter case **make sure that you include the data files for your tests in your submission**. Be aware that the `uk_stations.csv` dataset is quite large, so you might find it difficult to pick out journeys or regions that serve as useful test cases.

1.2.2 2023-11-02

Updated the output figure that is displayed as the output of `rail_network.plot_fares_to('KGX', save=True, bins=10)`, section: Plotting.

- The previous figure was generated using a subset of the stations that were provided in the final `uk_stations.csv`.

Corrected image now corresponds to the final revision of the `uk_stations.csv` file as provided with the assignment and starting code.

---

2 Setting

You are attempting to model how train fares in the UK are computed, given data on the distribution and organisation of stations. In order to test your hypothesis on how the fares are computed, you have decided to write some code to make it easier to load, analyse, and visualise the data. You have been provided with a Jupyter notebook (`analysing_fare_prices.ipynb`) with a simple workflow that you are planning to run once your code is complete.

2.1 Station information

We have provided you with some data about the rail stations in the UK, which is provided in the `uk_stations.csv` file. For each station, you know;

- Its unique CRS code.
- Its name.
- Its region. These relate to the concept of "hub" stations, below.
- Its location (latitude and longitude, in degrees).
- Whether it is a regional "hub" station. These values are stored as either `0` (zero) or `1` (one) but should be interpreted as booleans. `0` (`False`) implies the station is *not* a hub station, and `1` (`True`) the station is.

The data provided is an amalgamation of two publicly available data sources about the UK rail network. The hub stations have been selected by the course co-ordinators for the purpose of this assessment, and are chosen based on a metric extracted from the aforementioned data sources. You can read more about how the dataset was created in the appendix.

2.1.1 Model assumptions

In the UK, rail stations are divided into *regions*. These regions loosely correspond to groups of geographically close stations, although the physical span of the regions themselves can vary widely. Each station in the national rail network belongs to exactly one region.

The "rail network" itself is made up of all the stations and their connections to each other. These connections in turn determine how a person is able to travel between any two given stations.

A subset of the stations in the national rail network are *hub station*s: these stations see a high volume of traffic and/or passengers passing through compared to the other stations in their region. To identify whether a station is a hub or not when reading the dataset, a value of `1` (`True`) will be present in the "hub" column in *if and only if* it is a hub.

In reality there are a vast number of routes that can be taken between any two given stations. As such, we need to make some simplifying assumptions about the way in which stations are connected, and the journeys that passengers will be taking. Therefore, the model that you will be using in this assignment assumes the following:

- All routes between (connected) stations are of a length equal to the distance between the two stations.
- Every station within a particular region has a direct connection to every other station in the same region.
- Hub stations have direct connections to all other *hub stations*, even those in other regions, *in addition to* the stations within their own region.

To illustrate, let us consider the example network illustrated below.

There are two regions in this network; the Blue region (on the left) which contains the stations:

- Cobalt Mine,
- Sapphire Plaza,
- Bluebell Meadow (hub station),
- Brightwater (hub station),

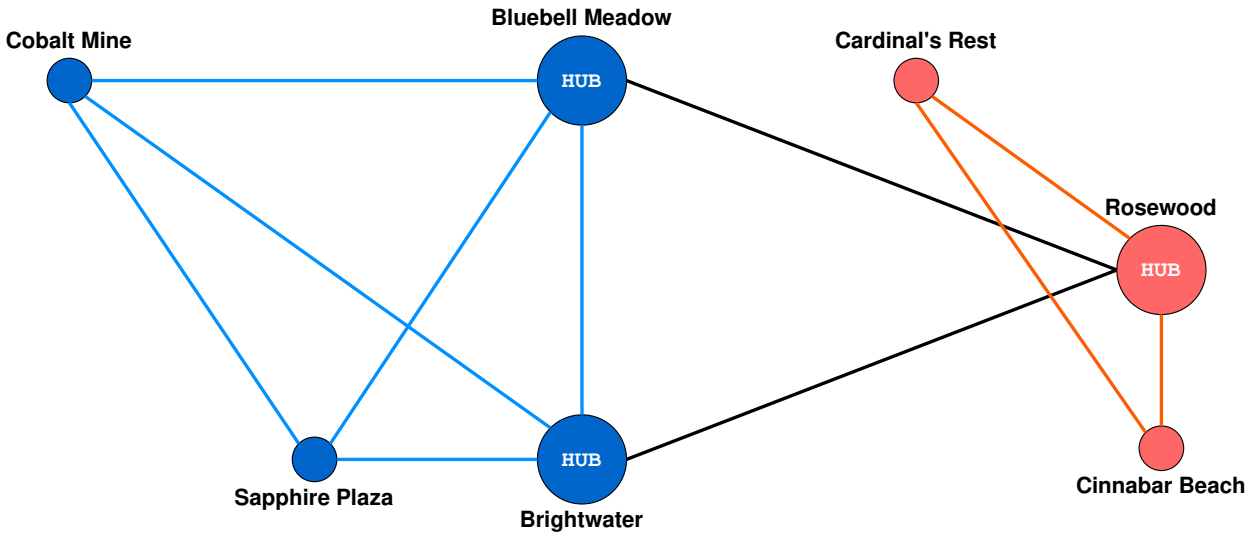

Figure 1: An example rail network that consists of 2 regions and three hubs.

and the Red region (on the right) containing:

- Cardinal's Rest,
- Rosewood (hub station),
- Cinnabar Beach.

From this information we can now work out how the stations connect to each other.

- Bluebell Meadow connects to Brightwater and Rosewood, because they are also hub stations. Bluebell Meadow also connects to Cobalt Mine and Sapphire Plaza, since these are in the Blue region.
- Brightwater - similarly to Bluebell Meadow - connects to Bluebell Meadow and Rosewood, because they are also hub stations. Brightwater also connects to Cobalt Mine and Sapphire Plaza, since these are also in the Blue region.
- Rosewood connects to Bluebell Meadow and Brightwater, since they are also hub stations. Rosewood also connects to Cardinal's Rest and Cinnabar Beach, since these are in the Red region.
- Cinnabar Beach and Cardinal's Rest only connect to each other and Rosewood (the other stations in the Red region),
- Cobalt Mine and Sapphire Plaza only connect to each other, Bluebell Meadow, and Brightwater (the other stations in the Blue region).

These connections are represented by the lines in the diagram above. The black lines in the diagram indicate connections that correspond to travelling *between regions*. It should be noted that travelling between hub stations *does not* imply that we are travelling between regions (see for example travelling between Brightwater and Bluebell Meadow).

2.1.2 Journeys and Fares

A *journey* from station `A` to station `B` consists of a number of legs (or "sub-journeys", or "parts"). One leg is one direct connection between stations.

Journeys are always performed as follows:

- If `A` and `B` are within the same region, the journey only has one leg as a passenger can go directly from `A` to `B`.
- Similarly, if `A` and `B` are both hub stations, the journey has one leg as a passenger can go directly between hub stations. Note that `A` and `B` might be in different regions in this case.

- If `A` and `B` belong to different regions and are not both hub stations; there are two further cases. First, we must identify the hub station within the same region as `A`, which we will call station `HubA` (which can be `A` itself). If there are multiple choices, we pick the hub station *closest* (by distance) to `A`. We also find the hub station `HubB` that is closest to `B` in the same way. Then;
  - If neither `A` nor `B` are hub stations, a passenger must first travel to `HubA`, then to `HubB`, and finally to `B`. In this case, the journey has 3 legs:
    1. `A` to `HubA`,
    2. `HubA` to `HubB`,
    3. `HubB` to `B`.
  - If `A` is a hub station, then `HubA` and `A` are the same station. In which case, the journey has 2 legs:
    1. `A` to `HubB`,
    2. `HubB` to `B`.
  - If `B` is a hub station, then `HubB` and `B` are the same station so we again can do a two-leg journey:
    1. `A` to `HubA`,
    2. `HubA` to `B`.
- **NOTE:** If `A` and `B` belong to different regions, and *at least one* of these regions does not have any hub stations in it, then a journey from `A` to `B` is impossible.

Journeys are written by listing the stations that a passenger passes through, in the order of travel. For example, a direct (or one-leg) journey *from* `A` *to* `B` is written `[A, B]`, whilst a journey with two legs would be something like `[A, HubA, B]`, and three legs `[A, HubA, HubB, B]`.

NOTE: If `A` and `B` are the *same* station, the journey is still written as `[A, B]`. This is important since (due to a clerical oversight) passengers used to be charged a fare for entering and leaving the same station, without going anywhere!

Let us use the example network from above to illustrate some example journeys:

- Travelling from Sapphire Plaza to Bluebell Meadow is a direct trip, written `[Sapphire Plaza, Bluebell Meadow]`, since they belong to the same region (the Blue region).
- Travelling from Rosewood to Sapphire plaza would be a 2 leg journey. Rosewood is a hub, so we can travel into the Blue region immediately. Brightwater is the nearest hub station to Sapphire Plaza, so the journey goes through there: `[Rosewood, Brightwater, Sapphire Plaza]`.
- The journey from Cinnabar Beach to Cobalt Mine has to utilise multiple hub stations. There is only one hub station in Red region, Rosewood. There are two hub stations in Blue region, but Bluebell Meadow is the one closest (distance-wise) to Cobalt Mine, so that is the one which is selected. Thus, the journey would be `[Cinnabar Beach, Rosewood, Bluebell Meadow, Cobalt Mine]`.
- Travelling from Bluebell Meadow to Cardinal's Rest is again a 2-leg journey; `[Cardinal's Rest, Rosewood, Bluebell Meadow]`. Despite the proximity between Cardinal's Rest and Bluebell Meadow, the lack of a direct connection due to their different regions forces a journey through Rosewood.

Each leg of a journey has an individual fare. Fare prices (in £-GBP) of direct travel between two connected stations are calculated using the formula:

$$\text{fare price} = 1 + d \times \mathrm{e}^{-\frac{d}{100}} \times \left(1 + \frac{\text{DR} \times \text{RHD}}{10}\right), \tag{1}$$

where

- e is the Euler number, approximately 2.71828. It is typically stored as a constant, or when raised to a power is called the exponential function.
- d is the distance between the two stations, (or equivalently the length of the direct connection between the stations).
- DR ("*Different Regions*") is equal to 1 if the stations belong to different regions, and 0 otherwise.

- RHD ("*Regional Hubs in Destination*") is equal to the number of hub stations *in the same region as the destination station*. This has the effect of making travel into busier regions more expensive than leaving those regions.

The fare price for a journey that has multiple legs is then the sum of the fare prices of the individual legs.

2.2 Goal

Your goal is to provide some code that will facilitate reading in this data, and starting to analyse it. The specific requirements are described in the next section. Your code must also check the validity of the data, as explained later.

We have already given you an outline of the code, which you have to complete. We have also given you a notebook, `analysing_fare_prices.ipynb`, with some examples of how we expect the code to work. This will let you test your code manually, however will not be marked. You will also need to write some (automated) unit tests to verify that the code itself is behaving as expected.

Our marking script will be testing the functionality of your code over the UK network (provided) and other similar networks. You are welcome to use the `uk_stations.csv` file in your test cases, or write your own custom networks for your tests - in the latter case **make sure that you include the data files for your tests in your submission**. Be aware that the `uk_stations.csv` dataset is quite large, so you might find it difficult to pick out journeys or regions that serve as useful test cases.

---

3 Your tasks

3.1 Git

To track your changes as and after you make them, you should work in a git repository. You should make commits as you go along, and give them meaningful descriptions.

The goal is that someone (you or others) can look at your commit history in the future and get a rough idea of what changes have happened. The messages should guide them in finding when a particular change was made – for example, if they want to undo it or fix a related bug. Therefore, avoid vague messages (*e.g.*, "Fixed a bug", "Made some changes") or ones that don't describe the changes at all (*e.g.*, "Finished section 1.2.1"). Prefer the use of concrete messages such as "Check the type of the arguments" or "Add tests for reading data".

Your repository should contain everything needed to run the code and tests (see below), but no files that are not necessary. In particular, you should not commit "artifact" files that are produced by your code. Please refer to the course notes for information on how to exclude such files from your repository!

You can work on one or multiple branches, as you prefer. We will only mark the state of the code at the latest commit on `main` branch of the repository you submit.

3.2 Main code

3.2.1 Outline of code provided

We have given you an outline of the code which you must fill in. The code is split over two files.

`railway.py` contains the definition of two classes, `Station` and `RailNetwork`, as well as the function `fare_price` which computes our approximation to the cost of a rail fare between two stations.

The `Station` class represents a single station, while the `RailNetwork` brings together all the stations from a dataset. The second file, `utilities.py`, is a place to put useful functions - there is a stub for one function you will need there.

Users should be able to create `Station` objects and call methods on them. However, the user will likely interact primarily through the `RailNetwork`, by creating it from a CSV file and retrieving some analysis results. Some of the methods of that class will, internally, call methods on the `Station` class as needed.

We have pre-populated the `RailNetwork` class with a couple of functions that will help you to visualise the rail networks that you have loaded in, and any journeys that are to be performed. These functions are purely to aid in your visualisation of the networks that you load in, and there is no need to make use of them if you do not wish to. You may remove these functions from your final submission if you so wish (but will not be penalised either way).

- You can invoke the `RailNetwork.plot_network` method to draw the rail network you have loaded, once you have setup the `regions` property.
- After you have written the `journey_planner` method, you can also invoke `RailNetwork.plot_journey(start, destination)` method to draw the journey that your code found between the start and destination stations.

You are free to write any additional functions or methods that you think are useful for your solution in any of these or additional files. However, your implementation must follow the interface we specify below; that is, the methods and functions we require should still be callable in the way we describe.

We now outline the functionality your code should provide.

3.2.2 A function for fare prices

As a starting point, you should write the body of the `fare_price` function, which computes the fare price using eq. 1 above, and returns it. You should also add a suitable docstring and `return` statement (and

type-hints if you deem them necessary). **Do not** change the call signature of the `fare_price` function, it must be callable as

```
fare_price(distance, different_regions, hubs_in_dest_region)
```

in your final submission. The `distance`, `different_regions`, and `hubs_in_dest_region` parameters should correspond to suitable variables outlined in eq. 1.

To avoid having to type out the value of the Euler number, you should import either the constant value or an appropriate function (see the exponential function) from a suitable library.

3.3 Reading and validating data

3.3.1 Creating `Station`s

To create a `Station` object, one must know some basic properties: its name, region, CRS code, geographical coordinates, and whether it is a hub station. These are passed to the class's constructor:

```
brighton = Station("Brighton", "South East", "BTN", 50.829659, -0.141234, True)
kings_cross = Station("London Kings Cross", "London", "KGX", 51.530827,
 ↪  -0.122907, True)
edinburgh_park = Station("Edinburgh Park", "Scotland", "EDP", 55.927615,
 ↪  -3.307829, False)
```

This information should be placed into the `name`, `region`, `crs`, `lat`, `lon`, and `hub` members of the `Station` class respectively. As this example shows;

- The name, region, and CRS code should be passed and stored as strings,
- The latitude and longitude are passed in as decimal numbers (in degrees),
- The hub flag should be passed as a boolean value.

The constructor should check for valid types and values for each of the properties.

- Latitude and longitude should be restricted to the -90 to 90 and -180 to 180 ranges respectively.
- The CRS code must be a 3-character string that consists of only uppercase letters (`ABC...XYZ`).

Appropriate errors should be thrown if this is not the case. The error messages should be informative, and you should use an appropriate error type. Avoid messages like "Something has gone wrong" - your goal is to help the user understand what the problem is!

There are different ways you can write the `Station` constructor. Python `dataclasses` might be a good option (but are not required).

3.3.2 Creating a `RailNetwork`

Next, you will need to provide the ability to create `RailNetwork`s. The `__init__` method for the `RailNetwork` class should take a list of `Station` objects as its input.

If the list provided contains two stations with the same CRS codes, a (helpful) error message should be provided indicating there is a fault in the input (as no two stations in the same `RailNetwork` may have the same "unique" identifier).

Otherwise, the `RailNetwork` constructor should store the `Station` objects in a `dict` member variable named `stations`. Given that the CRS codes of stations are guaranteed to be unique after checking the data in this way, the keys of the `RailNetwork.stations` variable should be the CRS codes.

```
list_of_stations = [brighton, kings_cross, edinburgh_park]
rail_network = RailNetwork(list_of_stations)
print(f"List of stations passed in: {list_of_stations}")
```

```
print(f"Stations in the network: {list(rail_network.stations.values())}")
print(f"Keys of rail_network.stations: {list(rail_network.stations.keys())}")
```

```
List of stations passed in: [Station(BTN-Brighton/South East-hub), KGX-London
  ↪ Kings Cross/London-hub), Station(EDP-Edinburgh Park/Scotland)]]
Stations in the network: [Station(BTN-Brighton/South East-hub),
  ↪ Station(KGX-London Kings Cross/London-hub), Station(EDP-Edinburgh
  ↪ Park/Scotland)]
Keys of rail_network.stations: ['BTN', 'KGX', 'EDP']
```

Additionally, users should also be able create a `RailNetwork` by passing in a file of the same format described above (such as the "uk_stations.csv" file) to the `read_rail_network` utility function. The argument to the `read_rail_network` function should be a `Path` object from Python's pathlib module. The file could be located anywhere on the user's computer; your code **should not** make any assumptions about its location. Additionally, the order of the columns may be different than the one shown in the sample file. Your `read_rail_network` function should be able to infer which column is which using the header information (first line).

For example:

```
from pathlib import Path

from utilities import read_rail_network

file_path = Path("../path/to/my/data/uk_stations.csv")
rail_network = read_rail_network(file_path)
```

3.3.3 `Station` class methods

The `Station` class should have a function, `distance_to`, to aid the analysis of fare prices. It should be callable in the following manner:

```
# the distance in km from Brighton to King's Cross
BTN_to_KGX = brighton.distance_to(kings_cross)
```

The `distance_to()` method should calculate the distance (in km) from the `Station` to another. To find the distance between two sets of coordinates, you should use the Haversine formula: the distance d between two points (ϕ_1, λ_1) and (ϕ_2, λ_2) (with latitudes ϕ_i and longitudes λ_i) is

$$d = 2R \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

where $R = 6371$ is the approximate radius of the Earth in km.

Additionally, you are required to set how Python represents the `Station` objects when printing or presenting outputs. This behaviour is controlled by two special methods; one of these special methods is `Station.__str__(self)`. This method returns a string, which is the text printed to the screen when you call `print` on a `Station` object! This in turn will let you execute `print(my_station)` and get a custom output with the important information, rather than the default printout. Similarly, there's another method that you'll need to find to produce the same output when instead of using `print`, the object is displayed by name.

When displayed using `print`, you should obtain the an output of the following format:

```
Station(<CRS>-<Station name>/<Station region>)
```

if the `Station` is **not** a hub station. If the station **is** a hub station, you should obtain

```
Station(<CRS>-<Station name>/<Station region>-hub)
```

instead. Note that the `-hub` text only appears if the `Station` is a hub.

Some example usage is given below:

```
print(brighton)
print(edinburgh_park)
```

```
Station(BTN-Brighton/South East-hub)
Station(EDP-Edinburgh Park/Scotland)
```

```
brighton
```

```
Station(BTN-Brighton/South East-hub)
```

3.3.4 Optional: Utility Functions

You are allowed - **but not required** - to add additional functions to the `Station` class if you think they will be useful. You will not lose or gain additional marks for including or excluding such functions, the idea is just here for you to explore. If you do choose to add an additional functions, don't forget to add appropriate tests for them!

One method you might find particularly helpful to write is the `Station.__eq__(self, other_station)`. This is the method that `==` uses, and it returns a `bool`. It may be useful to setup this method so that it compares the CRS codes of the two stations when the user tries to execute `Station1 == Station2`, and returns the result. CRS codes are unique, so the result of this comparison reflects whether any two `Station`s are equal (for the purposes of this assignment).

3.3.5 Basic `RailNetwork` Information

The `RailNetwork` class should provide some simple information about the network it encodes. This will be accessible by calling the methods `regions`, `n_stations`, and `hub_stations` of a `RailNetwork`. How these functions should behave is explained below:

The `regions` **property** should return a list of the unique regions within the network.

```
# Should return a list of *unique* regions that
# the stations in the rail network belong to.
rail_network.regions
```

```
['East of England', 'North West', 'London', 'Scotland', 'South West', 'West
↪  Midlands',
 'Yorkshire and The Humber', 'North East', 'East Midlands', 'South East',
↪  'Wales']
```

The `n_stations` **property** should return the number of stations in the network.

```
# Should return the number of stations in the network.
rail_network.n_stations
```

```
2395
```

Lastly, we need a method that will list the hub stations in the network, and optionally within a particular region. This is the `hub_stations` method: if called without any arguments it should return a list of all the hub stations in the network. If the optional "`region`" argument (which is a string) is provided, the list that is returned should consist of only the hub stations *within* that region. If the region does not exist in the network, an error should be thrown.

```
# Should return a list of the hub stations in the network.
# If passed the optional argument "region", it should only return a list of hub
  ↪ stations in that region.
rail_network.hub_stations()
len(rail_network.hub_stations())
```

```
[Station(ABD-Aberdeen/Scotland-hub), ..., Station(VXH-Vauxhall/London-hub)]]
41
```

```
rail_network.hub_stations("North West")
len(rail_network.hub_stations("North West"))
```

```
[Station(CAR-Carlisle/North West-hub), Station(LIV-Liverpool Lime Street/North
  ↪ West-hub),
 Station(MAN-Manchester Piccadilly/North West-hub)]
3
```

3.3.6 Planning Journeys

Our next step is to write a method that will plan journeys between two stations in our network. First, we should write the `closest_hub` method for `RailNetwork` that takes a `Station` as an input and computes the nearest hub station within the same region.

```
# Returns the closest hub Station to Edinburgh Park that is in its same region
edinburgh_park = Station(
    "Edinburgh Park", "Scotland", "EDP", 55.927615, -3.307829, False
)
rail_network.closest_hub(edinburgh_park)
```

```
Station(STG-Stirling/Scotland-hub)
```

NOTE: As mentioned above, it is possible for the user to supply a datafile in which a region has *no* hub stations. This situation will mean that we cannot provide any stations as an output from `closest_hub`, and so the function should **throw an appropriate error** in such a situation.

Next, we need to write the `RailNetwork`'s `journey_planner` method. This method should take the CRS codes of a starting station (`start`) and destination station (`dest`). It should return the journey between the two stations, as a list of `Station`s as described in the journeys and fares section. If the CRS codes provided do not correspond to stations in the network, an error should be thrown.

A journey between two stations is only impossible if we want to travel between two regions, at least one of which does not have a hub station. These are the same conditions under which the `closest_hub` method throws an error, which `journey_planner` will have to call anyway. In this case, no additional error handling should be introduced here - that is, an error should be raised by `closest_hub` *after* being called from within `journey_planner`.

```
# Returns a list that describes the journey a passenger would
# take when travelling from Brighton (BTN) to King's Cross (KGX)
rail_network.journey_planner("BTN", "KGX")
```

```
[Station(BTN-Brighton/South East-hub), Station(KGX-London Kings
 ↪  Cross/London-hub)]
```

Similar to the `journey_planner` method, `RailNetwork` should also have a `journey_fare` method that returns the cost in £-GBP of a rail fare between two stations.

```
# Returns the cost (as a float) of a journey starting at Brighton
# and ending at King's Cross.
rail_network.journey_fare("BTN", "KGX")
```

```
90.38331633489044
```

If the `summary` argument is provided and has the value `True`, a summary of the journey and the cost should be printed to the screen in precisely the format below:

```
Journey from Station A (STA) to Station B (STB)
Route: STA -> Hub Station A (HBA) -> Hub Station B (HBB) -> B
Fare: £XX.YY
```

For example, the following journeys on the uk_stations `RailNetwork` would display:

```
EDP_to_BTN_fare = rail_network.journey_fare("EDP", "BTN", summary=True)
```

```
Journey from: Edinburgh Park (EDP) to Brighton (BTN)
Route: EDP -> STG (Stirling) -> BTN
Fare: £31.87
```

```
AVY_to_CDF_fare = rail_network.journey_fare("AVY", "CDF", summary=True)
```

```
Journey from Aberdovey (AVY) to Cardiff Central (CDF)
Route: AVY -> CDF
Fare: £36.16
```

```
ABW_to_TQY_fare = rail_network.journey_fare("ABW", "TQY", summary=True)
```

```
>> Journey from Abbey Wood (ABW) to Torquay (TQY)
Route: ABW -> FST (London Fenchurch Street) -> EXC (Exeter Central) -> TQY
Fare: £60.84
```

3.3.7 Plotting

For you to get an overview of the fare prices that your model estimates for rail journeys, you would like to be able to quickly visualise the distribution of fare prices to a specific station.

Your task is therefore to add a method to the `RailNetwork` class called `plot_fares_to` which, taking the CRS code of a station as an input, produces a histogram plot of the fare prices from all ***other*** stations in the network to the provided station. If it is impossible to plan a journey between the input station and another station in the network, the function should skip computing this fare - ***it should not throw an error***. If the user passes in the optional argument `save`; the histogram plot should be saved to a file named as `Fare_prices_to_STATION_NAME.png`, otherwise the plot should be displayed to the screen. If there are spaces in the station name, you should replace them with underscores (_) when saving the plot. You should also be able to pass in additional keyword arguments to `plot_fares_to` which are then handed to the `matplotlib.pyplot.hist` plotting function.

```
# This should create a histogram showing the distribution of fare prices
# to London Kings Cross (KGX), saving the plot to the file
```

```
# "Fare_prices_to_London_King's_Cross.png".
# Additionally, the histogram should consist of 10 bins, since the bins
# parameter has been passed as a keyword argument.
rail_network.plot_fares_to('KGX', save=True, bins=10)
```

Figure 2: Sample output graph of the `plot_fares_to` method, when called by `rail_network.plot_fares_to('KGX', save=True, bins=10)`

3.4 Tests

In addition to the main code in `railway.py` and `utilities.py`, you should create some tests to verify that the main code is behaving correctly.

You should create a file called `test_railway.py` with unit tests written for the `pytest` framework. Running `pytest` in your submission directory should find the tests, and the tests should pass.

At a minimum, your tests should cover these cases:

- Errors with appropriate error messages are thrown when invalid values are encountered (see Reading and validating data section above)
- The `Station` methods work correctly
- For the `RailNetwork` class:
  - The methods and properties providing simple information about the collection work correctly.
  - The `hub_stations` and `closest_hub` methods work as intended, in the case of `hub_stations` including the case when it is passed optional arguments.
  - `journey_planner` correctly determines the journeys to be taken between stations.
  - `journey_fare` correctly determines the fare prices for journeys of different numbers of legs.

You will likely need several tests for each of the points above to cover the necessary cases. Fixtures and test parametrisations might be a good idea in some cases.

You *do not* need to write tests for the functions that are provided to you by us. Specifically, the following functions and methods *do not* require you to write tests for them:

- `RailNetwork.plot_network`
- `RailNetwork.plot_journey`

You are of course free to add any more tests that you think make your code more robust and help you with the development! Make sure that your submission includes all files that are needed to run the tests.

4 Directory structure and submission

You must submit your exercise solution to Moodle as a single uploaded gzip format archive. You **must** use only the `tar.gz`, not any other archiver, such as `.zip` or `.rar`. If we cannot extract the files from the submitted file with gzip, you will receive zero marks.

To create a `tar.gz` file you need to run the following command in a bash terminal:

```
tar zcvf <filename>.tar.gz <directory_to_compress>
```

The folder structure inside your `tar.gz` archive must have a single top-level folder, whose folder name is your UCL candidate number, so that on running

```
tar zxvf <filename>.tar.gz
```

this folder appears. This top level folder must contain all the parts of your solution (the repository with your files). You will lose marks if, on extracting, your archive creates other files or folders at the same level as this folder, as we will be extracting all the assignments in the same place on our computers when we mark them! We have created a `pip`-installable command-line tool for you to check the directory structure of your submission, should you wish to use it: instructions can be found here.

Inside your top level folder, you should have a `railway` directory. Only the `railway` directory has to be a git repository. Within the `railway` directory, you should have the files `railway.py`, `utilities.py` and `test_railway.py`, as well as any other files you need.

You should `git init` inside your `railway` folder, as soon as you create it, and git commit your work regularly as the exercise progresses. **We will only mark the latest commit on the `main` branch of the repository**, and will run `git switch main` before grading, so be careful about leaving changes on other branches or uncommitted changes to the `HEAD` of `main`. Due to our automated marking tool, only work that has a valid git repository, and follows the folder and file structure described above, will receive credit.

You can use GitHub for your work if you wish, although we will only mark the code you submit on moodle. Due to the need to avoid plagiarism, do not use a public GitHub repository for your work - instead, use the repository that you'll get access to by accepting this invitation:https://classroom.github.com/a/ote2eY1m After you accept the permissions, this will create a repository named `railway-<gh_username>`.

In summary, your directory structure as extracted from the `candidateNumber.tar.gz` file should look like this:

```
candidateNumber/
└── railway/
    ├── .git/
    ├── railway.py
    ├── utilities.py
    ├── test_railway.py
    └── <any other files or folders containing data you may need>
```

5 Getting help

This assignment is designed to check your understanding of the concepts we have covered in the class. You may find it useful to review the:

- Lecture notes,
- Classroom exercises,
- Other resources linked from Moodle,
- Or the official Python docs.

There are many places you can find advice for coding on the Internet, but make sure you understand any code that you take inspiration from, and whether it makes sense for your purposes.

You can ask questions about the assignment on the Q&A Forum on Moodle. If we receive repeated or very important questions, we will create a Frequently Asked Questions post to collect the answers, and keep it updated. The Errata section of this document will also be updated to reflect any common questions.

You can also email us your questions to the module leaders, or book an office hours slot.

5.1 Use of AI in this assignment

Use of AI tools like GitHub Copilot, ChatGPT, Bing chat, or Google Bard are allowed on this assignment in an assistive role manner. You are only allowed to use the free version of these tools or through an education license (such as Github Student development pack). However, you should be able to solve this assignment without the need of any AI tool.

If you choose to use AI tools to assist you, you are required to add an `AIusage.md` file within your submission providing a detailed explanation of why, where and how you used AI tools in the assignment. Describe the specific AI tool you used (including name and version), the purpose it served in your assignment, and any challenges you encountered while integrating it into your code, as well as any improvements or limitations they brought to your solution. Additionally, include at least one prompt you used and the provided code snippets obtained.

6 Marking scheme

Note that because of our automated marking tool, a solution which does not match the standard solution structure defined above, with file and folder names exactly as stated, may not receive marks, even if the solution is otherwise good. "Follow on marks" are not guaranteed in this case.

You can add more functions or files (e.g., fixtures) if you consider it's appropriate. However, you should not change the name of the provided files and functions, and should adhere to the names provided for the methods we have requested you write.

- **Version control with git (15%)**
  - Sensible commit sizes (5 marks)
  - Appropriate commit messages (5 marks)
  - "Artefacts" and unnecessary files not included in any commit (5 marks)
- **Data loading (10%)**
  - Reading stations from CSV (5 marks)
  - Correct instantiation of `Station` and `RailNetwork` (4 marks)
  - Correct representation of `Station`s (1 mark)
- **Analysis methods (27%)**
  - Implementation and docstring for `fare_price` method (2 marks)
  - Implementation of distance function (2 marks)
  - Simple data about the number of regions and stations in the network (2 marks)
  - Implementation of `hub_stations` method (3 marks)
  - Implementation of `closest_hub` method (3 marks)
  - Implementation of `journey_planner` method (7 marks)
  - Implementation of `journey_fare` method (3 marks)

  - Plots and associated methods (5 marks)

- **Validation (20%)**
  - Check `Station` member types (8 marks)
  - Check `Station` member values (6 marks)
  - Check `Station`s passed to `RailNetwork` have unique CRS codes (2 marks)
  - Errors have appropriate messages and type (4 marks)

- **Tests (23%)**
  - At least one test for the `fare_price` function (1 mark)
  - At least four negative tests, checking the handling of improper inputs to `Station` (4 marks)
  - A test to check that CRS codes loaded into a `RailNetwork` are unique (1 mark)
  - Test `Station` class `distance_to` method (2 marks)
  - Tests for `RailNetwork` simple information functions (3 marks)
  - Tests for the `hub_stations` and `closest_hub` methods (4 marks)
  - Tests for the `journey_planner` and `journey_fare` methods (6 marks)
  - Tests for the `plot_fare_to` method (2 marks)

- **Style and structure (5%)**
  - Good names for variables, functions and methods (3 marks)
  - Good structure, avoiding repetition when possible (2 marks)

Indicate what areas of your coursework you particularly would like feedback on by adding comments to a `Feedback.md` file within your submission.

7 Appendix: Additional information about the data

The data file distributed with the assignment is an amalgamation of two publicly available datasets:

- Obtained from the Office of Rail and Road (ORR) dataset of estimates of station usage between April 2020 and May 2021.
  - Link to the raw data
  - Station names, regions, and CRS code have been obtained from this dataset.
  - Other columns have been removed.
- Obtained from a public GitHub repository, itself derived from data used by Trainline EU.
  - Link to the raw data
  - Latitude and longitude coordinates have been obtained from this dataset.

The dataset distributed was created by matching entries in the two datasets and removing the unwanted columns. "Hub" station flags were then added manually by the course co-ordinators for the purpose of this assessment, however the assignment of such stations was motivated by the station usage data from the ORR.

The exercise makes some simplifying assumptions which may lead to inaccuracies, so be wary of drawing real, accurate conclusions!