

Coursework 2: Tube Map

COMP70053 Python Programming (Autumn 2025)

Deadline: Friday 24th Oct 2025 (7pm BST)



1 Overview

In this coursework, you will implement classes that form part of a TubeMap network. In particular, you will attempt to compute the shortest path between two London Tube stations.

The objectives of this coursework are:

- to give you hands-on experience in implementing an existing algorithm;
- to allow you to practise working with advanced Python constructs such as dictionaries, lists, and sets, and reading from JSON files;
- to enable you to use object-oriented programming in a practical application;
- to give you some practical experience in using your own modules/packages.

This coursework covers topics in Core Lessons 1 to 9 of the course materials.

You are limited to the **Python Standard Library** for this coursework (i.e. anything listed on this page: <https://docs.python.org/3.12/library/>). For example, the `json` module may be used. Do **NOT** use any external libraries such as NumPy.

2 Preparation

You are supplied with some skeleton code (see section 4) and some raw data.

2.1 Understand the data

Examine `data/london.json`. The JSON structure represents the London Tube network, and has three keys:

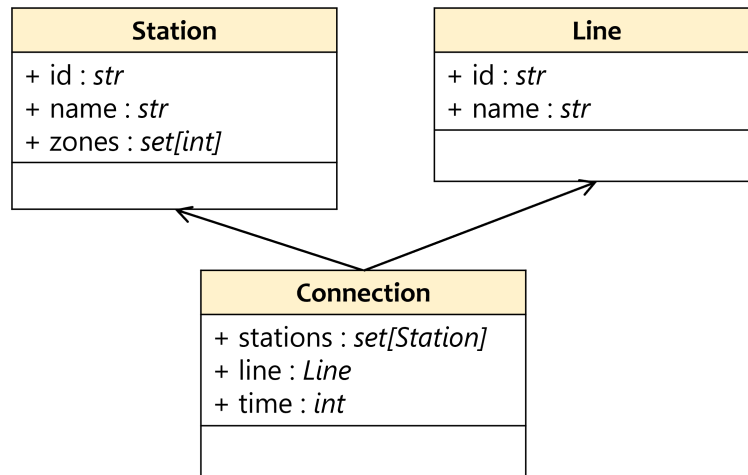
- **lines**: A list of lines on the Tube network (e.g. Victoria Line). Each line has an ID ("**line**") and a "**name**".
- **stations**: A list of stations on the Tube network (e.g. Gloucester Road). We are interested in the following properties: "**id**", "**name**", and "**zone**". The "**zone**" property represents the zone in which the station is situated (e.g. White City is in Zone 2). The zone may also be represented as a non-integer, for example Earl's Court is marked as **zone** "1.5". This indicates that the station is in both Zones 1 and 2 (i.e. at the boundary)¹.
- **connections**: A list of connections on the Tube network. A connection represents two adjacent **stations** on a specific **line**, and the **time** it takes to travel between the two stations on that **line**. A pair of **stations** may be connected via multiple **lines** (e.g. South Kensington and Gloucester Road are connected via the Piccadilly Line and the District Line). So, you will find two connection instances for this pair of stations ("**station1**": "99" and "**station2**": "236"), but on different **lines**. In some cases, even the **time** duration might differ.

You will be converting these from the JSON file into the equivalent **Station**, **Line** and **Connection** class instances (next section).

¹London commuting tip! If you live in West or Southwest London and travel via the Piccadilly or District Lines, you can save a lot on fares by stopping at Earl's Court (and walking ≈ 20 mins to Huxley) rather than Gloucester Road in Zone 1 (and walking ≈ 8 mins), since Earl's Court is considered to still be in Zone 2 if you travel from the west. The same is true if you travel from the west on the Central Line and stop at Notting Hill Gate and walk ≈ 30 mins to Huxley.

2.2 Understand the classes

Examine `tube/components.py`. You are given three class definitions: `Station`, `Line` and `Connection`.



Please read the docstrings to understand these classes. Also see the end of the file for examples of how we expect to use these classes.

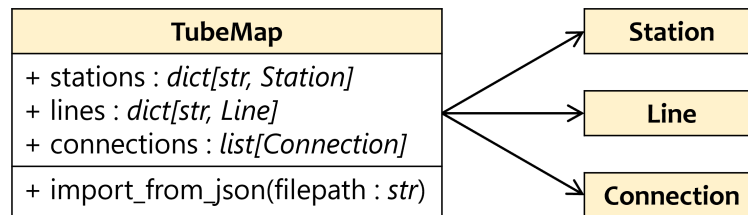
Notes:

- Take note of the expected data type of each attribute (e.g. `Connection.time` must be an `int`, `Station.id` and `Line.id` should be `str`) Our LabTS tests are quite strict in enforcing the exact data types. **YOU HAVE BEEN WARNED!**
- As mentioned, some **Stations** may belong to multiple **zones**. For example, Earl's Court station is in both Zones 1 and 2. In this case, `zones = {1, 2}`.

3 Tasks

3.1 Task 1: Complete the TubeMap class

Examine the class definition of `TubeMap` in `tube/map.py`. The class comprises a collection of `Stations`, `Lines`, and `Connections` between `Stations` on particular `Lines`:



Your first task is to complete the `import_from_json(self, filepath)` method of the `TubeMap` class in `tube/map.py`. The method imports the tube map data from a JSON file indicated by `filepath` and populates the `TubeMap`'s attributes (`stations`, `lines`, `connections`). You can assume that the JSON file will be in the same format as `data/london.json`. Data from previous imports should not be retained.

Please read the docstrings for a detailed description of the attributes, and what we expect the method to do when it receives an invalid `filepath` as an argument.

A function `test_import()` is provided to you at the end of the file so that (i) you have an example usage of the `TubeMap` object and (ii) you have a basic test for `import_from_json()` using `data/london.json`.

You can test your implementation by running `python -m tube.map` from the **root** of the project directory.

Hints:

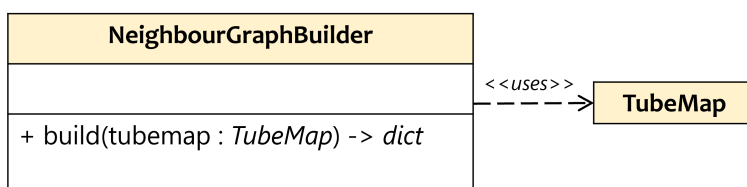
- The data types in the JSON file may differ from the expected data types in the classes! Make sure these are converted to the appropriate type.
- The "line" property in the JSON file (for the key "line") represents the `Line.id` attribute.

3.2 Task 2: Implement the NeighbourGraphBuilder class

Now that you have your `TubeMap` instance, you will be able to compute the shortest path from one station to another. For this, you will need to know which stations are connected to each other. You already have this information from the `connections` attribute in `TubeMap` from earlier. You should now encode and index this information into a graph data structure to make it easier for you to compute the shortest path later.

For your second task, complete the `NeighbourGraphBuilder` class in `network/graph.py`.

The class must have a `build()` method as a minimum. The method takes a `TubeMap` instance as input. It returns a nested `dict` representing the graph, or an empty `dict` if the input is invalid.



The returned nested `dict` is like a 2D grid, except that the indices are the station `ids` of two adjacent stations (these are strings). The value is a **list of `Connection` instances** taken from `TubeMap.connections`. Examples are provided in the docstrings in `network/graph.py`.

Important note: We assume that the connection between stations is bidirectional. For example, if *Baker Street* station (id "11") and *Marylebone* station (id "163") are connected, then you should set the value of both `graph["11"]["163"]` and `graph["163"]["11"]` to the **same list of `Connection` instances between the two stations**.

You can test your implementation by running `python -m network.graph` from the **root** of the project directory.

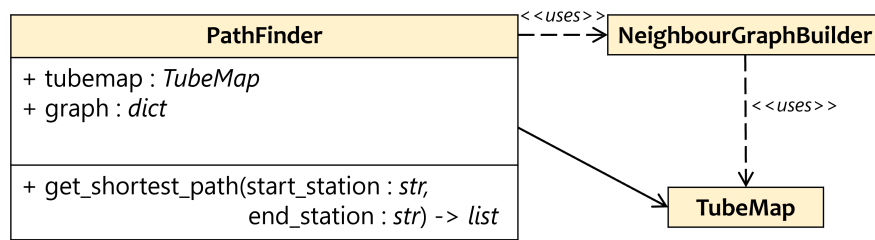
3.3 Task 3: Implement the Pathfinder class

Given that you can now generate a neighbour graph, you can use the graph to compute the shortest path between two stations. When we say ‘shortest path’, we mean the path that takes the least time.

For your third task, complete the Pathfinder class in network/path.py. More specifically, complete the `get_shortest_path()` method that takes in the start and end station names as string inputs. It should compute the shortest path (in terms of duration) to get from the start station to the end station.

The method should return a list of `Stations` representing **one** shortest path between the start and end stations. If either of the provided station names does not exist, the method should return `None`. If the start and end stations are the same, return a list with an instance of this station itself.

The dependencies between `PathFinder`, `NeighbourGraphBuilder` and `TubeMap` are summarised in the class diagram below:



A widely known algorithm to compute the shortest path on a graph is Dijkstra’s algorithm. You can find many tutorials on YouTube explaining the algorithm (e.g. <https://www.youtube.com/watch?v=GazC3A40QTE>). If you prefer, you can also find relevant explanations, illustrations and pseudocode on Wikipedia: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Pseudocode. It’s sufficient to just implement Dijkstra’s algorithm for this coursework; there is no need to try to implement a more advanced algorithm.

You can test your implementation by running `python -m network.path` from the **root** of the project directory.

4 Skeleton code

You are provided with a git repository containing the skeleton code. You will use `git` to push your code changes to the repository. To obtain the code, clone the repository using SSH, replacing `<login>` with your College username.

- `git clone git@gitlab.doc.ic.ac.uk:lab2526_autumn/python_cw2_<login>.git`

Test your code with LabTS (<https://teaching.doc.ic.ac.uk/labts>) after pushing your changes to GitLab. Make sure your code runs successfully on LabTS without any syntax errors. You will lose marks otherwise. These automated tests are purely to ensure that your code runs correctly on the LabTS servers. **You are responsible for testing your own code more extensively.** The final assessment will be performed using a more extensive set of tests not made available to you.

Multiple versions of tests

You may notice that LabTS may run multiple versions of tests in some cases. For example, `NeighbourGraphBuilder` relies on `TubeMap`, but which `TubeMap`? Your implementation of `TubeMap` or ours? LabTS will test with both versions; you will only need to pass at least one of these variants to pass the test.

This arrangement is so that any mistakes in your implementation of `TubeMap` will not propagate to subsequent tests that rely on your implementation of `TubeMap`. In general, such propagated errors will be taken into account in the marking scheme. We will not penalise you twice.

5 Handing in your coursework

Go to LabTS (<https://teaching.doc.ic.ac.uk/labts>), and click on the “Submit this commit” button next to the specific commit you wish to submit. This will automatically submit your commit SHA1 hash to Scientia. Then go to Scientia to confirm that the coursework has been submitted.

6 Grading scheme

Criteria	Max	Details
Code correctness (Tasks 1+2+3)	4+3+7	Does your code run without any syntax errors? Does your code give the expected output for valid test cases, whether they be standard or edge cases? Does your program not end up in an infinite loop for certain inputs?
Code robustness	2	How well does your code pass our automated stress test with different kinds of input? Is it error free from invalid inputs?
Code readability	4	Did you use good coding style and naming conventions? Did you use meaningful names? Is your code highly abstracted, modular, and self-explanatory? Did you provide useful, meaningful comments where necessary?
TOTAL	20	Will be scaled to 8% of your final module grade.

Credits

Coursework designed and prepared by Luca Grillotti and Josiah Wang