Conference travel emissions (Coursework 2)

Reading and analysing a conference travel dataset

Please read this assignment carefully.

This assignment asks you to write some code for loading and analysing a real dataset about the environmental costs of travelling to academic conferences. 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 git version control and GitHub.

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.

For this assignment, you can only use the Python standard library, matplotlib and pytest. Your code should work with Python 3.8 or newer.

First, we set out the problem we are solving. Next, we specify the target for your solution in detail. Finally, to assist you in creating a good solution, we state the marking scheme we will use.

1 Setting

You are part of the organising committee for an international academic conference that will host attendees from around the world. You and your colleagues would like to choose the conference location that minimises the CO2 emissions of the attendees' travels. They have therefore asked you to write some code to make it easier to load, analyse and visualise the data, so the committee can decide where the conference will be held. They've also provided a notebook with the steps that they are planning to run.

1.1 Attendee information

We have recorded some pieces of information about the cities that the conference attendees live in. For each city, we know:

- its name,
- its state (for some of them),
- its country,
- the number of conference attendees, and
- its location (latitude and longitude, in degrees).

The data provided is from a real conference (AGU 2019). It is held in a CSV file, and the first row of the file is a header containing the names of the columns. For more information about the whole dataset and its origin, see the appendix.

1.2 Goal

Your goal is to provide some code that will allow your fellow committee members to read in this data, and start analysing it. The specific requirements are described in the next section. Your code must also check the validity of the data, as explained there.

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

2 Your tasks

2.1 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 3.2"). Prefer 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. Refer to the notes for 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.

2.2 Main code

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

`cities.py` contains the definition of two classes, `City` and `CityCollection`. The first represents a single city, while the second brings together all the cities from a dataset. The second file, `utils.py`, is a place to put useful functions - there is a stub for one function you will need there.

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.

Users can create `City` objects and call methods on them. However, the user will likely interact primarily through the `CityCollection`, 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 `City` class as needed.

Below is the functionality your code should include.

2.2.1 Reading and validating data

To create a `City` object, one must know some basic properties: its name, the country it is in, the number of citizens wanting to attend the conference (the attendees) and its geographical coordinates. These are passed to the class's constructor:

```python
zurich = City('Zurich', 'Switzerland', 52, 47.22, 8.33)
san_francisco = City('San Francisco', 'United States', 71, 37.77, -122.41)
greenwich = City('London', 'United Kingdom', 15, 51.48, 0)
```

As this example shows, the name and country should be passed as strings, number of attendees as an integer, and the latitude and longitude as decimal numbers (in degrees). The constructor should additionally check for valid types and values for each of the properties (*i.e.*, attendees must be a positive number, and latitude and longitude should be restricted to the -90 to 90 and -180 to 180 ranges respectively), and throw errors 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 users understand what the problem is!

There are different ways you can write that `City` constructor, in this case, Python `dataclasses` is a good option (but not required).

The constructor of the `CityCollection` should take a list of `City` objects as an argument and hold it as a member variable named `cities`.

```python
list_of_cities = [zurich, san_francisco]
city_collection = CityCollection(list_of_cities)
city_collection.cities == list_of_cities # this should be True
```

Additionally, users should also be able create a city collection by passing in the file containing basic information ("attendee_locations") to the `read_attendees_file` utility function. The argument to the `read_attendees` file 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.

This is an example of how you should be able to create a collection:

```python
from pathlib import Path

file_path = Path("attendee_locations.csv")
collection = CityCollection(file_path)
```

2.2.2 Analysis (City)

The `City` class should have two functions to aid the analysis,

```python
# the distance in km from Zurich to San Francisco
zrh_to_sfo = zurich.distance_to(san_francisco)
# the total CO2 emitted by the 52 researchers from Zurich travelling to San Francisco
zrh_to_sfo_co2 = zurich.co2_to(san_francisco)
```

The first function, `distance_to()` allows to calculate the distance (in km) from a city 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.

The second function `co2_to` should allow you to calculate the total emissions for researchers from a certain `City` to travel to a conference held in another `City` - the host city. To compute this, we will assume that for distances up to 1000 km, people will use public transport, and therefore emit 200 kg/CO2/km/person. For distances larger than 1000 km, but up to 8000 km, they will travel using a short-haul flight, and emit 250 kg/CO2/km/person. For longer distances, they will fly long-haul and emit 300 kg/CO2/km/person.

2.2.3 Analysis (CityCollection)

First, the `CityCollection` class should provide some simple information around the information it holds. This will be accessible by calling the methods `countries` and `total_attendees` of a `CityCollection`. How these functions should behave is explained in code comments below

```python
# should return a list of *unique* countries that
# the cities in the collection belong to
city_collection.countries()

# should return the number of all the attendees
city_collection.total_attendees()
```

Further, we'd like to have a way to compute information of the consequences of holding the conference in a specific host city, *e.g.*, Zurich.

```
# returns the total distance travelled by all attendees
city_collection.total_distance_travel_to(zurich)

# returns a dictionary mapping the attendees' country to the
# distance travelled by all attendees from that country
# to the host city
city_collection.travel_by_country(zurich)

# returns a dictionary mapping the attendees' country to the
# the CO2 emitted by all attendees from that country
# to the host city
city_collection.co2_by_country(zurich)

# returns the total CO2 emitted by all attendees
# if the conference were held in Zurich
city_collection.total_co2(zurich)
```

The `summary` method should not return any value. Instead, it should print out in the screen the information below.

```
city_collection.summary(zurich)
```

The output has to follow exactly this sample:

```
Host city: Zurich (Switzerland)
Total CO2: XXX tonnes
Total attendees travelling to Zurich from YYY different cities: ZZZ
```

where `X`, `Y` and `Z` should be integers, and the total CO2 has to be provided in tonnes (1,000 kg) rounded to zero decimal places.

In the final analysis step, we'd like to have a `CityCollection` method called `sorted_by_emissions()` that returns a sorted list of city names and CO2 emissions. The elements should be sorted by the total CO2 emitted by all attendees if that city were to be chosen as the host city. The first item in the list should have the lowest total emissions, and the last item the highest total emissions.

2.2.4 Plotting

For your colleagues to get a quick overview of which countries are the ones with larger CO2 emission to attend the conference, they would like to be able to visualise it.

Your task is therefore to add a method to the `CityCollection` class called `plot_top_emitters` which plots the CO2 emissions/country on the Y-axis against the countries on the X-axis. Only the `n` countries with the most emissions should be plotted individually, while emissions for the other countries should be summed and put into an aggregated data point for "All other countries". The function should also take a boolean argument called `save`, which specifies whether to save the plot to a file (if `save` is `True`) or show it (if `save` is `False`). By default, `n` should be `10` and `save` should be `False`.

```
# This should create a plot with 8 data points
# (the 7 top emitters + the sum of the others) and
# save it as "./buenos_aires.png"
city_collection.plot_top_emitters(buenos_aires, 7, True):
```

The saved file should be named as the host city, all in lower case and replacing any possible spaces by underscores (`_`).

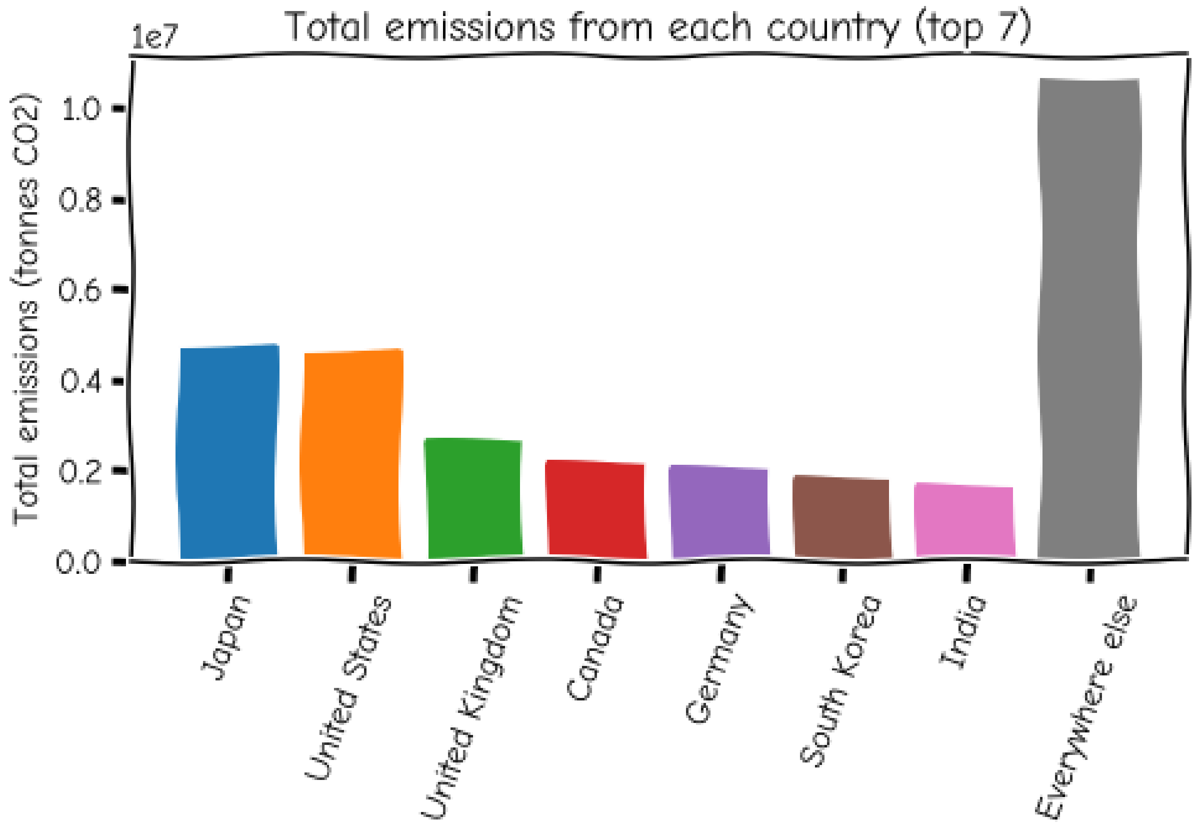

Figure 1: Sample output graph of the `plot_top_emitters` method showing the 8 columns for a random generated dataset. Your programme should produce axis and bars with straight lines.

2.3 Tests

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

You should create a file called `test_cities.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 `City` methods work correctly for all modes of transportation
- The `CityCollection` methods providing simple information about the collection work correctly
- The `CityCollection` methods providing information about a host-city-specific metric work correctly.
- The `sorted_by_emissions` method works correctly.

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 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 (for example, any sample data files if you use them).

2.4 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 on a bash terminal:

```
tar zcvf filename.tar.gz directoryName
```

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!

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

You should `git init` inside your `conference` 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**, so be careful about leaving changes on other branches or uncommitted. 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/KmoJ503a which, after you accept the permissions, will create a repository named `conference-travel-emissions-<gh_username>`.

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

```
candidateNumber/
└── conference/
    ├── .git/
    ├── cities.py
    ├── utils.py
    ├── test_cities.py
    └── <any other files you may need>
```

3 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, and 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. You can also email us your questions at `arc-teaching@ucl.ac.uk` or book an office hours slot.

4 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.

- **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 (9%)**
  - Reading cities from CSV (5 marks)
  - Correct instantiation of city and city collection (4 marks)
- **Analysis methods (28%)**
  - Sorting cities by emissions if they were chosen as host (4 marks)
  - Implementation of distance function (2 marks)
  - Implementation of CO2 function (2 marks)
  - Simple data about collection (2 marks)
  - Host city data queries (10 marks)
  - Summary of collection (3 marks)
  - Plots (5 marks)
- **Validation (20%)**
  - Check for input types (10 marks)
  - Check for input value (6 marks)
  - Errors have appropriate messages and type (4 marks)

- **Tests (23%)**
  - Test given city class methods (8 marks)
  - Tests for simple collection functions (3 marks)
  - Tests for host city-dependent collection functions (8 marks)
  - At least four negative tests, checking the handling of improper inputs (4 marks)
- **Style and structure (5%)**
  - Good names for variables, functions and methods (3 marks)
  - Good structure, avoiding repetition when possible (2 marks)

Appendix: Additional information about the data

The data file distributed with the assignment is the one used at An analysis of ways to decarbonize conference travel after COVID-19 article. The dataset (and its data analysis) are released as open source under the GPL license. Permission has been granted to be used for this assignment.

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