CSM0120 - Programming for Scientists

Assessed Coursework 2

Samantha Pendleton

2017

Contents

1 Executive Summary

In this python assignment, I have set up various functions that have different purposes – altogether it is based on location and forecasts. There are numerous functions available, so users can use any for their purpose: plotting towns from a file onto a UK map, connecting to MET Norway API returning weather information, and plotting weather information for a user (location, wind speed, and wind direction).

Exercise 1: `plot_all_towns()`
This function takes in a file of latitude and longitudes of UK cities and plots them onto a UK map.

Exercise 2: `input_city_into_db()`, `input_user_into_db()`
Both functions input data from files (e.g. CSVs) into a specific table of a database.

Exercise 3: `forecast()`
Requests a latitude and longitude of a city, then searches and downloads the forecast, saves the XML. Data from MET Norway.

Exercise 4: `wind_forecast()`, `plot_winds()`
The first function takes in the results from `forecast()` and requests a tag and attribute, for example, "windSpeed" and "mps", it then returns a list of time-stamps and the data values of the tag/attributes. The second function then plots this: plotting "windSpeed" values against time.

Exercise 5: `user_loc()`, `loc_coord()`, `plot_user_windspeeddirection()`, `user_forecast_map()`
Exercise 5 has four different functions - `user_forecast_map()` calls in the other three functions as one final function, plus calls in another function `wind_forecast()` (to search for other tags, like "windDirection"); `user_loc()` needs an email and it'll return the city of that user's email; and `loc_coord()` needs a city and returns the co-ordinates - `plot_user_windspeeddirection()` plots the individual's co-ordinates on a UK map with speed depending on intensity, the icon is a triangle directed to degrees depending on wind direction.

2 Technical Overview

Through this assignment, I used various `PYTHON` features: I imported numerous packages, used `for-loops`, `if statements`, connecting to files plus databases, producing SQL statements, and plotting.

Exercise 1

This section has one function `plot_all_towns()`, which takes the parameter of a file-name, it imports the `UKMap` module and class. In which the function then plots city co-ordinates on a UK map. In the `main()` function, it demonstrates by calling in `plot_all_towns("latlon.csv")` and the results are as displayed:

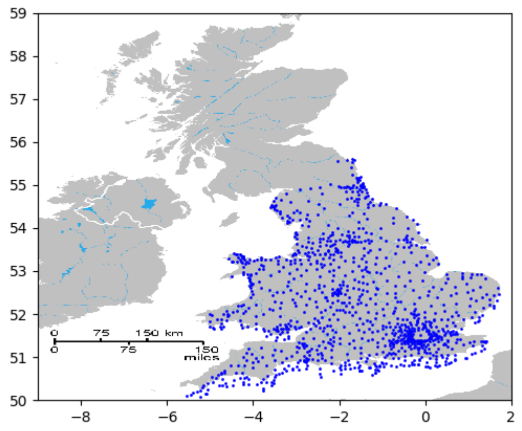


Figure 1: The plot of `plot_all_towns("latlon.csv") -` observe the co-ordinates of cities from the CSV file are plotted

Abbots Langley	51.70	-0.41
Aberaeron	52.23	-4.24
Aberdare	51.71	-3.45

Table 1: First three rows of "latlon.csv"

```
1 def plot_all_towns(filename):
2     mp = UKMap.UKMap()
3     coordinates = open(filename)
4     csvreader = csv.reader(coordinates)
5     for line in csvreader:
6         mp.plot(float(line[2]),float(line[1]))
7         mp.show
8     mp.savefig("map_\%s.png" \% (filename))
9     coordinates.close()
```

Listing 1: Function `plot_all_towns()`

Talking through the code:
2: uses the `UKMap()` class from the `UKMap` module.
3: opens the file, creates a connection.
5: using a `for-loop` to plot the co-ordinates from the file.
9: closes the connection of opening the file.

Exercise 2
For this section, the aim was to create a database with tables that will hold data from two CSVs, one mentioned previously (see table above) and a CSV of users: title, first name, last name, city, email, d.o.b and phone.

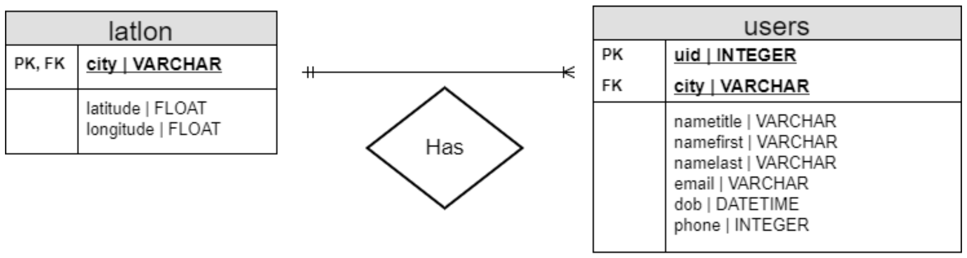


Figure 2: ER diagram of the designed database.

As you can see, the table “latlon” has the Primary Key of city: all city titles are unique: none are duplicates. The other attributes “latitude” and “longitude” are both floats; when I stored and later retrived co-ordinates, they remained the same value (e.g. -0.14... was entered and retrieved as -0.14) – this entity/table can have 1 or multiple users. The table “users” has multiple attributes that suitably holds the data from the “users.csv” file. Users can only have one city from “latlon” and city is the foreign key.

I created the database with “SQLite Manager”[1] and then created functions to input the data from both files into their specific table on the database. Below is an example of one of the two functions I created for exercise 2.

```
1  def input_user_into_db(filename, db_conn):
2      conn = sqlite3.connect(db_conn)
3      c = conn.cursor()
4      in_file = open(filename)
5      csvreader = csv.reader(in_file)
6      header_row = next(csvreader)
7      for row in csvreader:
8          parameters = {"nametitle":row[0],
9                  "namefirst":row[1],
10                 "namelast":row[2],
11                 "city":row[3],
12                 "email":row[4],
13                 "dob":row[5],
14                 "phone":row[6]
15                 }
16         query = """INSERT INTO users (nametitle, namefirst,
     namelast, city, email, dob, phone)
17         VALUES (:nametitle,:namefirst,:namelast,:city,:email,:
     dob,:phone)"""
18         result = c.execute(query, parameters)
19         conn.commit()
20     conn.close()
```

Listing 2: Function `input_user_into_db()`

Talking through the code:
2: connects to the database using the function parameter.
6: ignoring first row, since “users.csv” has a header for attributes.
8: parameters are in a dictionary to identify attribute titles.
16: SQL query being put together.
17: adding the dictionary values from identifying keys.
18: executing the query using the parameters.
19: committing.
20: closing the connection.

[1] lazierthanthou; https://addons.mozilla.org/en-GB/firefox/addon/sqlite-manager/

Exercise 3

The function `forecast()` connects to MET Norway API and returns weather information. It requests a latitude and longitude parameters and then returns XML soup object. The following code is an example of the XML:

```xml
<?xml version="1.0" encoding="utf-8"?>
<weatherdata created="2018-01-05T16:06:04Z" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://api.met.no/weatherapi/locationforecast/1.9/schema">
 <meta>
  <model from="2018-01-05T17:00:00Z" name="EC.GEO.0125" nextrun="2018-01-05T20:00:00Z" runended="2018-01-05T07:35:43Z" termin="2018-01-05T00:00:00Z" to="2018-01-15T00:00:00Z"/>
 </meta>
 <product class="pointData">
  <time datatype="forecast" from="2018-01-05T17:00:00Z" to="2018-01-05T17:00:00Z">
   <location altitude="100" latitude="52.6700" longitude="-1.8100">
    <temperature id="TTT" unit="celsius" value="2.6"/>
    <windDirection deg="274.3" id="dd" name="W"/>
```

Listing 3: Soup Object from `forecast()`

`<<Data from MET Norway>>`

The function also saves the XML as "coordinate_forecast_%f_%f.xml" (with the %f being co-ordinates) and uses the `prettify()` function of the "`bs4`" package (new version of `Beautiful Soup`) to format it better: a more efficient tree structure.

```python
def forecast(lat, lon):
    url = "http://api.yr.no/weatherapi/locationforecast/1.9/?"
    payload = {"lat":+lat,
               "lon":+lon
               }
    response = requests.get(url, params=payload)

    if response.status_code == 200:
        soup = bs4.BeautifulSoup(response.text, "xml")
        f = open("coordinate_forecast_%f_%f.xml" %(lat, lon), "w")
        f.write(soup.prettify())
        f.close()
        return soup
    else:
        return None
```

Listing 4: Function `forecast()`

Talking through the code:
2: stating the URL in preparation for the request.
3: payload of latitude and longitude (function parameters).
8: if response HTTP status code = 200 (OK).
9: store response as an XML beautiful soup object.
11: using `bs4` function `prettify()`.

Exercise 4

I created numerous functions for the different exercises. I could have simply created one function for each exercise however the implementation choice of multiple functions allows for functionality and efficiency: for exercise 4, instead of combining the two functions into one, I created two separate functions in which one takes in a soup object and searches for tag/attribute values, and the other plots those values. This method is better compared to one function that'll search the soup object for "windSpeeds" and "Times" then immediately plotting. My method makes it so users can call the `wind_forecast()` function whenever – as long as they have a soup object from exercise 3 and will input a tag and attribute parameters. Exercise 5 has separate functions too: `user_loc()` and `loc_coord()`, these two can use used separately allowing users to simply enter an email and gain the city result, or enter a city and gain the co-ordinates.

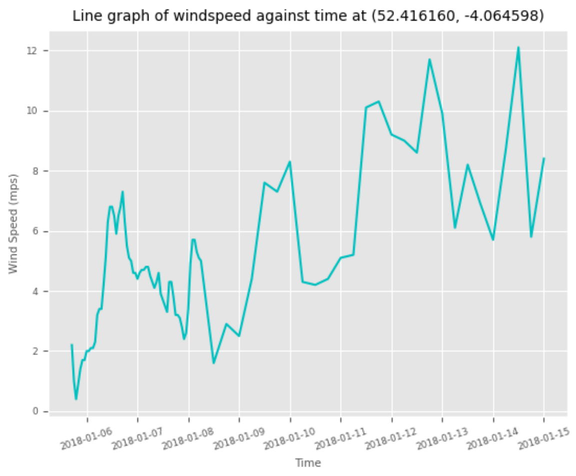


Figure 3: Result plot of `plot_winds()`.

Exercise 5

Furthermore, originally I hard-coded the `wind_forecast()` function (it only accepted a soup object and had hard-code of searching for "windSpeeds" only), however, as I progressed onto exercise 5, I altered the function to accept a tag and attribute (e.g. "windDirection", "deg"), this allows a user to save information of any tag and attribute data as a variable: - this variable will be two lists within a list: time-stamps plus the values. This method is much more efficient as it allows users to call this variable whenever.

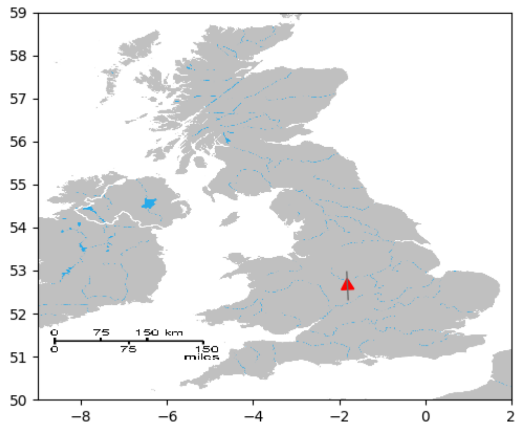


Figure 4: Result of `plot_user_windspeeddirection()`.

Above is the plot of the user's forecast, To get the colours correct for the plotting of wind speed, I mapped the variable onto the colour scheme.

```
colors = ['blue', 'green', 'yellow', 'orange', 'red']
top = max(user_wind[0]) + 1
step = top / len(colors)
speed = user_wind[0][0]
x = int(speed/step)
..
mp.plot(coord2, coord1, marker=(3, 0, int(user_wind[1][0])),
    color=colors[x], markersize=10)
```

Listing 5: Code for colour coordinating plots.

Exercise 6

For additional work, I would continuously save plots throughout script. This allows all plots produced to be saved, 'matplotlib' also has the feature of saving plots, users can edit the output to be logarithmic or alter colours, then save with a different title.

For exercise 5, when creating the wind direction icon, it was difficult to quite know which direction the wind speed actually was, so I added an additional line on the plot to properly show the direction.

```
mp.plot(coord2, coord1, marker=(3, 0, int(user_wind[1][0])),
    color=colors[x], markersize=10)
mp.plot(coord2, coord1, marker=(2, 0, int(user_wind[1][0])),
    color='dimgrey', markersize=20)
```

Listing 6: Code to show marker direction for wind direction.

For the wind speed against time plot, `plot_winds()` that I demonstrated in `main()`, I used to display the graph with the 'ggplot' style:

```
with plt.style.context('ggplot'):
  plot_winds(times, windspeeds, coords)
```

Listing 7: Code to show assigning a style to a plot.

I created a function for temperature to extend exercise 4, this small function adds a neat feature to the code.

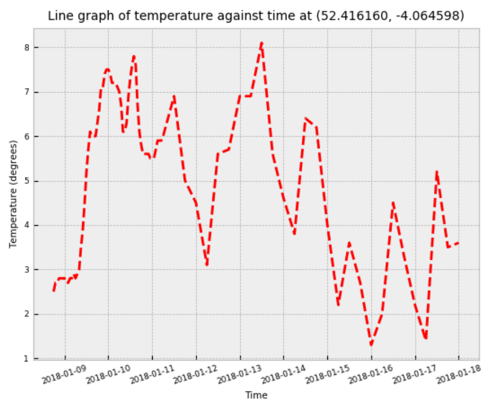


Figure 5: Result of `plot_temp()`.

3 Software Testing

In this section of the report, I will be describing the type of tests I conducted both throughout the project plus at the end. To begin with, the functions work as stated: output is correct and when testing my code as a whole, I used new files, or entered values I knew wouldn't work.
All of these functions depend on the input parameters, to explain: in my examples in `main()`, I used parameters that I know would work and so I would create my own files, or enter my own data to test if it'll work on different things.

Testing individual Functions
Exercise 1

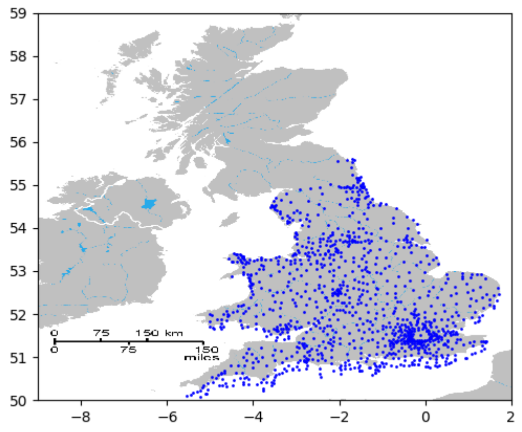


Figure 6: The plot of `plot_all_towns("latlon.csv")` - observe the coordinates of cities from the CSV file are plotted

I found it quite difficult to test exercise 1 function, `plot_all_towns()`, as the result and plot looked accurate, as seen above. As stated in the assignment brief, we can see that there is some distortion when plotting due to the fact that the Earth is spherical and maps are 2-dimensional.

I tested my own CSV file (see below): `"latlon_test.csv"`, and from the plot below, we can obverse that the results are as expected: *Cardiff* isn't visible due to incorrect values.

Liverpool	53.43	-2.93
Aberystwyth	52.38	-4.05
Manchester	53.47	-2.27
Cardiff	-3.19	51.5

Table 2: First three rows of ”latlon.csv”

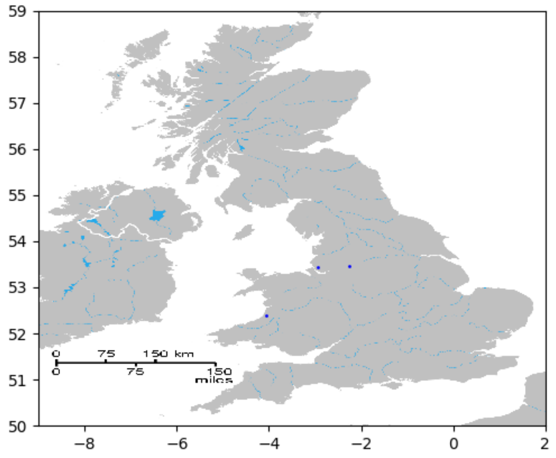


Figure 7: The plot of `plot_all_towns("latlon_test.csv")` - observe the co-ordinates of cities from the CSV file are/are not plotted.

Exercise 2
For this section, I was unsure how to set up some tests due to my code being very specific, but I checked the file row count and them checked the SQLite Manager system for the value count too. In a Linux Terminal, I used `wc -l users.csv` to check row count, result was 15001; in SQLite Manager it stated there were 15000 records - meaning the code works and skips the header as planned. I did the same for the other CSV file: `wc -l latlon.csv`, results were: 1095; SQLite Manager results were the same.

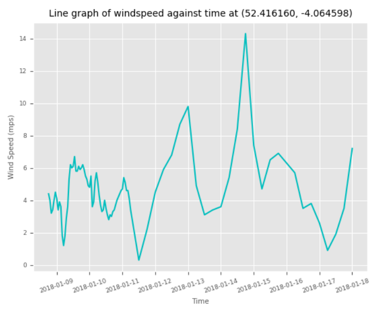
To expand on testing, I originally noticed `conn.commit()` may have been in the wrong place as it took quite a while for records to be submitted into the database, after removing it from the `for-loop` and changing it to above the `conn.close()` and noticed the data entered the database table a lot faster.

Exercise 3

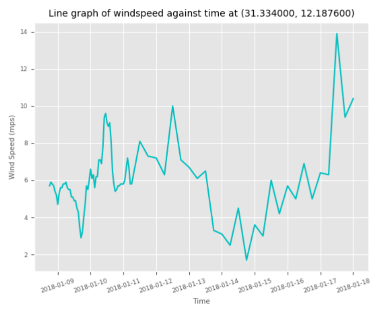
To test the function, `forecast()`, I used numerous co-ordinates and they all worked as planned. I also tested some coordinates that shouldn't work and as planned: they didn't work.

Exercise 4

Exercise 4 was a follow-up from exercise 3, it uses the returned soup object and then searches through the XML for an attribute value of a tag. I tested the plot using different co-ordinates and saw that the map differed.



(a) Co-ordinates (52.41616, -4.064598)

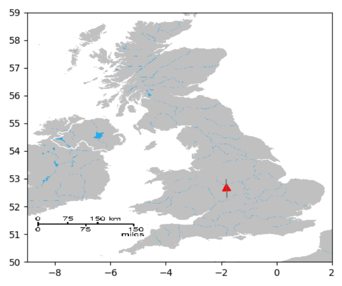


(b) Co-ordinates (31.334, 12.1876)

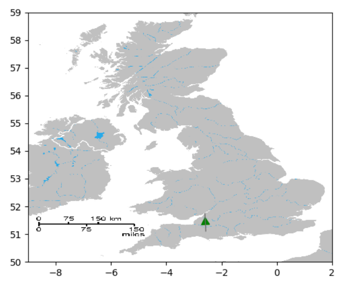
Figure 8: The plot of `plot_winds()`

Exercise 5

For exercise 5 `user_forecast_map()` I input different emails, as seen below, as we can see that the graph plots as expected. I considered that some user locations won't be in the database so I included: `if coord is not None: do something else: print("co-ordinates not in database to plot")`.



(a) `"marie.howard@example.com"`



(b) `"rebecca.baker@example.com"`

Figure 9: The plot of `plot_user_windspeeddirection()`

4 Reflections and Future Work

This assignment was very successful, I have achieved all requests for each section despite some areas that could be improved, which will be explained in more detail further on.

Reflecting

Exercise 1

I believe exercise 1 was performed at a high standard: the function read in the data of town then plotted it on a UK map using the `UKMap` module. I even expanded by adding an additional line of code to save the map in the working directory. Perhaps I could have saved the plot as a unique file-name, however, I believe it was sufficient enough.

Exercise 2

Exercise 2 was completely successfully, however, I am quite unsure if the functions could have been improved: rather than two separate functions, I wondered if possibly combining the two into one to create a function for inserting into a database could be achieved - since the functions I created are hard-coded, perhaps a function could be made to connect to the database plus file and somewhat create it's own parameters - this potential new function could request a file-name, database, and table name. However, this may be difficult as some files have a header row so code would be needed to do a sort of checking statement; for example, `if header_row == TRUE: next(csvreader)` - but how will it know of there is a header row? - We don't want to confuse actual data with headers.

Exercise 3

This section requests to use the `locationforecast` service from MET Norway API to download and save a forecast for a particular location (specifically in `main()`, Aberystwyth (52.41616,-4.064598) co-ordinates are used): a Beautiful Soup Object is returned as requested resulting in this section/function successfully completing the requirements. Naming the XML was successful: it saves the file title as `coordinate_forecast_%f_%f.xml` with `%f` being the co-ordinates.

Exercise 4

This section requests that a function is created to use the soup object from exercise 3, and search within it. It requests for the tag "windSpeed" and attribute "mps" plus times to then be plotted. Originally I hard-coded the function however later altered it to request a tag and attribute, allowing users to input the tag "windDirection" and attribute "deg" (used in exercise 5 for this purpose). Plotting was completed well: I used the data to plot a line graph, with a suitable title, x & y axis.
I think my expansion with the temperature graph was a unique additional, however this could have been a good additional to exercise 5, perhaps labelling the wind forecast with temperature.

Exercise 5

Furthermore, I believe my exercise 5 function was efficiently created as it has separate functions, with a final function that uses the others: it calls in the function from exercise 4. Having multiple functions allows users to use these separate functions for their own purpose.

I believe I could have spent more time testing my code, using `doctest()` and `pytest()` (a library for writing unit tests). I struggle quite a lot when it comes to testing code as I am unsure how to test, or what specifically to test: despite testing the function with different entry parameters, I struggle with the other aspects of testing: `exceptions`.

```python
import pytest
import doctest
def user_loc(email, db_conn):
    """
    >>> user_loc("rebecca.baker@example.com", "csm0120_database.
    sqlite")
     bristol
    """

if __name__ == "__main__":
    pytest.main()
    doctest.testmod()
```

Listing 8: Example of `pytest() and doctest()`.

Extensions

In future, a possible extension could include changing exercise 5 to input numerous emails (a list) for multiple users to display different wind speeds/directions in attempt to make a weather forecast map for multiple cities/towns of users. Furthermore, for plotting, it could be possible to add labels on the grid/map of city or town names, this would make the map appear authentic and professional.

Moreover potentially creating a temperature map or perhaps altering the wind speed & direction map to label temperatures. As you can see in exercise 4 (`plot_temp()`), a small temperature plot has been produced however it can be expanded:- a simple plot could suffice: the icon can depend on temperature: if high (hot) temperature: triangle directed upwards with the colour red; if medium the icon be a simple square and coloured green; and if low (cold) triangle pointing down and colour being blue.

I could combine both exercise 4 plotting functions to take in a time list and data-list; the axis labels will be the tag names from the `wind_forecast()` function.

Possibly another extension could be to include code for exercise 1 function that recognises latitude and longitude figures for UK map - so the code could detect if the value isn't <60 and >50 to swap the integer with the other, as long as the number is <10.