Temperature Data Analysis EE24B141-A2

Files included : assignment2.py, plotting_lib.py
Aim: Implement 5 analytical functions

Documentation

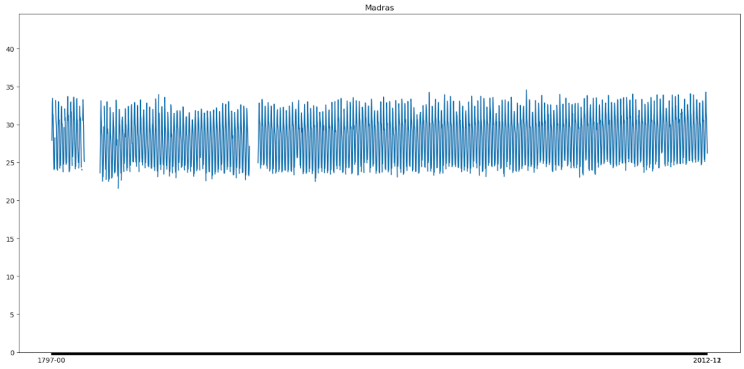
Introduction

The main principle behind the codebase is to emphasize orthogonality and modularity. Functions are divided into 4 types - Extraction, Modifier, Analysis and API calls.

- API calls are wrappers linking the first 3 types in meaningful contexts.
- Extraction functions interact with external files. *Ex. get_city_temperatures()* has been split as an I/O wrapper and *filter()*; which is now considered a modifier function and does the data comprehension
- Modifier functions can be chained together to perform operations on the data as required. The universal language between these functions is the **records** datatype (dictionary of date:temp pairs).
- Analysis functions provide insights on records and are the final part of the pipeline

Find_temperature_extremes()

The function finds the hottest and coldest months and temperatures for a city. Function *get_city_temperatures()* returns records pertaining to the particular city. If no records are found the function returns appropriate null values.
Hottest and coldest temperatures are found using the stdlib functions *max()* and *min()* and the dates are looped through to find which date corresponds to each extreme. Results are output in the appropriate dictionary format. Given graph is temperature data for Madras



Get_seasonal_averages()

Seasons are defined with a hashmap from the season name to a tuple containing the months it spans. The function goes through the record for a city and logs no.of data points and sum of temperatures. If no data points exist for that season, average temperature is given as NaN.

Compare_decades()

The extraction function *get_city_temperatures()* is given a start_date and end_date and extracts only that particular decade for each. The analysis function *get_period_average()* takes in these as inputs and performs an unweighted average. This returns average and no.of data points considered. The average for the two decades is then considered and a trend produced.

Find_similar_cities()

Functionality is similar to that of the above functions - extract and perform an unweighted average. Initially implemented using *get_city_temperatures()* with every available city, it stalled code execution. After splitting the function into a file wrapper and *filter()* it improved execution time significantly.

For further performance gains though, a new extraction function *get_city_averages()* was created to do the same in one go and return the required data for calculating average and the city's country – It loops through the csv file and logs every record as needed. If the average temperature is within the tolerance window around our target city's average temperature then the city is added to *similar_cities.*

Get_temperature_trends()

This function necessitated the creation of 2 modifier functions *get_annual_records()* and *get_window_average().* The former averages temperatures over a year by comparing strings, while the latter performs a centered moving window average. The output of the moving window average is fed to the analysis function *build_trends().* The function uses a class to keep track of the previous state of the system, and based on the current data read it either continues a trend or logs a pushes a trend into cooling_periods / warming_periods. The overall slope is calculated using the normalised correlation coefficient formula and verified by plotting on the respective graph.

(Graph above is temperature_trends in Madras and the rate of change of temperature for the period)

Appendix

- For complicated output structures classes are used to simplify error handling. Since they are of the same scope as the function, variables defined within the function can be directly referenced.
- Each helper function has different configurations and parameters with the intention of increasing the scope of the codebase