

Data Analysis of a Document Tracker

F21SC - Industrial Programming - Coursework 2

07/12/2021

H00359236

Contents

1.	Introduction	3
2.	Requirements Checklist.....	3
2.1	Python	3
2.2	Views by country/continent	3
2.3	Views by browser	3
2.4	Reader Profiles	3
2.5	“Also likes” favourites.....	3
2.6	“Also likes” graph.....	3
2.7	GUI Usage.....	4
2.8	Command-line usage.....	4
3.	Design Considerations.....	4
3.1	Program Structure	4
3.2	Data Handling	4
3.3	Use of Advanced Language Features	5
4.	User Guide	5
4.1	Command-line Interface	5
5.	Developer Guide	6
5.1	Task 2 & 3.....	6
5.2	Task 5 Sorting Function	7
6.	Testing	8
7.	Reflections.....	10
8.	Conclusions.....	10

Demonstration Video Link: <https://youtu.be/FV--AzPPI80>

1. Introduction

The purpose of this report is to highlight the functionality of the program created for data analysis of a document tracker from both a user and developer perspective, provide motivations for particular design decisions, an overview of the testing done and to reflect on the programming language used and its effect on implementation.

The brief was to develop a data analysis application using Python 3 that would be able to undertake a number of data analysis tasks in an efficient way such that it can be scaled up to large JSON datasets of up to 5 million records. In order to implement this, I used an Ubuntu 20.04.3 Linux distribution running on Oracle VirtualBox 6.1.30. I chose this configuration because I had had issues in the past running x2go and the standalone MACS Linux VM and so chose this as a more stable option while also meeting the Linux requirements of the coursework.

To implement the project, I used Visual Studio Code (version ccbaa2d2) running on my Ubuntu Virtual Machine (VM) as I have used it before so was familiar with the environment and features, in particular working with Python.

Unfortunately, due to being ill for a significant portion of the time allocated to complete this coursework, I was unable to implement all requirements and so I had to make some prioritisation decisions about what to implement. This meant I did not manage to implement the GUI or requirement 2b.

2. Requirements Checklist

2.1 Python

The core logic of this program has been implemented in Python 3,

2.2 Views by country/continent

- a). This application meets this specification fully.
- b). This functionality has not been implemented.

2.3 Views by browser

- a). This application meets this specification fully.
- b). This application meets this specification fully.

2.4 Reader Profiles

This application meets this specification fully.

2.5 "Also likes" favourites

- a,b,c,d). This application meets this specification fully.

2.6 "Also likes" graph

This application meets this specification fully.

2.7 GUI Usage

This functionality has not been implemented.

2.8 Command-line usage

This application meets this specification fully.

3. Design Considerations

3.1 Program Structure

My program is split into 4 main files, "cw2" controls the flow of execution as this is the file that is called when run from the command line. It takes in as possible arguments 'UUID', 'doc_UUID', 'task_id' and 'file_name'. Firstly, the command-line arguments are parsed and a file object is created based on the file name parameter. The task flag is then checked and a function is called that parses the data specific to that task before running the main function for that task. Tasks are a collection of functions split between 3 files; one file contains the functions corresponding to tasks 2, 3 and 4 due to there being similarities in their execution that can leverage shared functions. Tasks 5 and 6 are then separated out into their own files and imported to be called by "cw2".

3.2 Data Handling

When initially designing the application and testing out possible implementations, I made the mistake of using the 10k dataset as my test data. My approach was to iterate through all JSON objects in the data file parsing each one to a Python dictionary and adding it to a list. This worked well for the requirements I had implemented with files up to size 400k however, when I tried to run the 3 million objects file, I received a MemoryError and the program would stop execution. It was immediately clear what the issue was; I was simply trying to parse in too much data at one time and I was running out of memory. I began trying different methods to parse in the 3 million objects without triggering a memory error with the only real success being when I used the pandas read_json function and specifying a chunksize (500000 seemed to work well) to create an iterator of sequential pandas dataframes containing the 3m records.

While this worked in practice it was not a practical solution as I still could not combine the dataframes due to memory constraints and would end up iterating through the pandas json_reader object each time I needed to retrieve data which felt unnecessarily computationally expensive. Therefore, I decided to change my approach. Instead of trying to read all the data in at the start of the program execution and then manipulating it based on the task specified by the user, I selectively read in the data necessary for the task specified by the user. For example, for task 5d, rather than read in all of the data, I only parsed in the visitor_uuid, event_type and subject_doc_id of each JSON object. This significantly increased performance and my program was now able to parse in all 3 million records in just over 20 seconds and completing task 5d in just under 30 seconds.

Where possible, with the exception of when I needed to use a set to create a collection of unique values, I used dictionaries and lists to store and process the data so that I could use dictionary and list comprehension to increase performance speed. One exception to this was when creating a dictionary of visitor_uuid, event_type and subject_doc_id to append to the list when parsing in the data for task 5d. I initially used dictionary and list comprehension to achieve this in fewer lines of code but I found that it was on average 5 seconds faster to create an empty dictionary and add each of the 3 datapoints individually and then appending to the list.

3.3 Use of Advanced Language Features

While I feel I could have used more advanced language features, I was happy with the level of functional programming I managed to achieve using list and dictionary comprehension. I think this drastically reduced the size of my program and helped achieve significant performance gains.

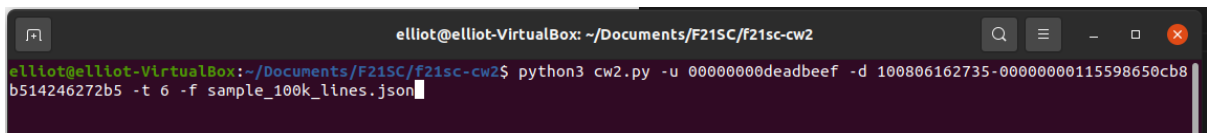
4. User Guide

4.1 Command-line Interface

The command for running the program from the command-line takes the following form:

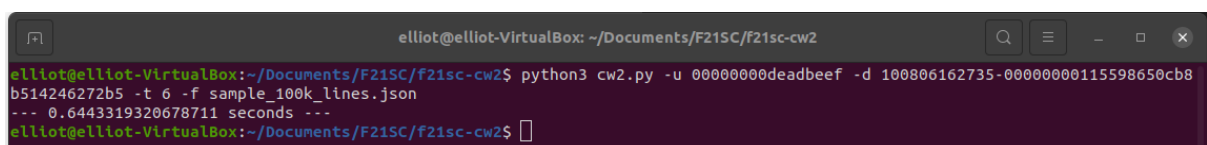
```
python3 cw2.py -u 'user_uuid' -d 'doc_uuid' -t 'task_id' -f 'file_name'
```

For example, in the following screenshot running task 6 on the 100k dataset:



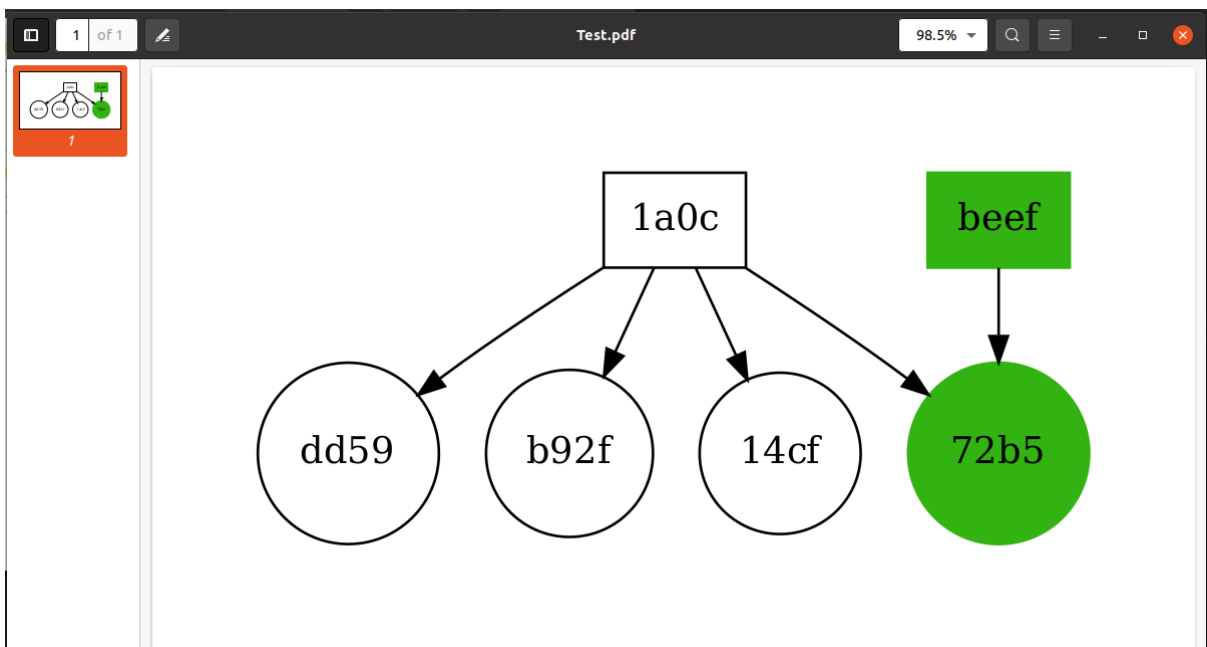
```
elliott@elliott-VirtualBox: ~/Documents/F215C/f215c-cw2
elliott@elliott-VirtualBox:~/Documents/F215C/f215c-cw2$ python3 cw2.py -u 00000000deadbeef -d 100806162735-00000000115598650cb8b514246272b5 -t 6 -f sample_100k_lines.json
```

The execution speed is then printed to the command-line:



```
elliott@elliott-VirtualBox:~/Documents/F215C/f215c-cw2$ python3 cw2.py -u 00000000deadbeef -d 100806162735-00000000115598650cb8b514246272b5 -t 6 -f sample_100k_lines.json
--- 0.6443319320678711 seconds ---
elliott@elliott-VirtualBox:~/Documents/F215C/f215c-cw2$
```

And the graphviz graph is displayed showing the “also likes”:



5. Developer Guide

My application has fairly simple structure consisting of a main “cw2” that controls the flow of execution and 3 additional files; one for tasks 2,3 and 4 as they share some of the same functions and a file for tasks 5 and 6 respectively. For each task the flow of control follows this sequence:

1. Create a file object using the file variable passed in by the user.
2. Identify the task from -t flag input.
3. Call the data parsing function that corresponds to that task and pass in the file as an argument.
4. Perform the task.
5. Output the result.

5.1 Task 2 & 3

For my implementation of tasks 2 and 3 I have used a similar process. During data processing the necessary data is condensed into a list of values. For task 2 this is a list of countries and for task 3 this is a list of visitor_useragent strings. Where they differ is that there is now an extra process step for task 3b in order to extract the browser name from the verbose string. I do this using a module called [httpagentparser](https://pypi.org/project/httpagentparser/) (<https://pypi.org/project/httpagentparser/>) that parses browser user agent string into a dictionary of its components. From this I am able to extract the browser name.

```
if flag == '3b':  
    browser_list = [http.detect(x) for x in browsers]  
    browsers = [i['browser']['name'] if i.get('browser') else i.get('version') for i in browser_list]  
    browser_arr = np.array(browsers)  
    browser_arr = browser_arr[browser_arr != np.array(None)]
```

Using list comprehension to parse the user agent strings and then again to extract the browser name I found to be the quickest way to undertake this process. Parsing the user agent strings is by far the most computationally expensive operation in the program as it can take up to 110 seconds to process the 3m records file. After extracting the browser names, I convert the list into a numpy array in preparation to be input into the get_unique_counts function, and then remove any NoneTypes from the numpy array as there are some instances where the user agent string had invalid browser names that were not parsed by the httpagentparser.

The data for task 2 and 3 is then passed through the get_unique_counts function:

```
values, counts = np.unique(arr, return_counts=True)  
return dict(zip(values, counts))
```

The numpy.unique function with return_counts=True returns an array of unique values and an array of the number of times each unique value appears. I then return a dictionary of the zipped values and counts which will be used to produce the histogram for the respective task using the show_histogram function.

5.2 Task 5 Sorting Function

```
result = defaultdict(int)

for i in dict.keys():
    for n in dict[i]:
        result[n] += 1

sort_result = sorted(result.items(), key=lambda x:x[1], reverse=True)

return sort_result
```

To create the sorting used by the “also likes” functionality I first create a defaultdict from the python collections module and set the default value to int. The purpose of this is so the dictionary does not throw a KeyError when indexing a key that doesn’t exist in the dictionary; instead, it creates an entry for the key and sets the value to the default value set when defaultdict was initialised. This led to more concise and cleaner code as I didn’t have to put in if conditions to avoid any KeyErrors and incrementing the value of a certain key only requires one short line of code i.e. `result[n] += 1`.

To sort the result dictionary which is now a dictionary of key value pairs where the key is a document uuid and the value is the number of people who have read it, I call the sorted function and use lambda expression `x[1]` to specify the sort value. This sorts the dictionary based on the values and with `reverse=True` it is in descending order. This sorting function is then passed in as a parameter to the `also_likes` function as not to fix the sorting implementation of the “also likes” functionality.

6. Testing

Test	Command	Outcome	Execution Time (seconds)	Comments
Task 2a 100k	python3 cw2.py -t 2a -f sample_100k_lines.json	Pass	3.2	Histogram produced correctly with correct configuration and axis labels.
Task 2a 3m	python3 cw2.py -t 2a -f sample_3m_lines.json	Pass	24	Histogram produced correctly with correct configuration and axis labels.
Task 3a 100k	python3 cw2.py -t 3a -f sample_100k_lines.json	Pass	5	Histogram produced correctly with correct configuration and axis labels.
Task 3a 3m	python3 cw2.py -t 3a -f sample_3m_lines.json	Pass	12	Histogram produced correctly with correct configuration and axis labels.
Task 3b 100k	python3 cw2.py -t 3b -f sample_100k_lines.json	Pass	2.4	Histogram produced correctly with correct configuration and axis labels.
Task 3b 3m	python3 cw2.py -t 3b -f sample_3m_lines.json	Pass	22.4	Histogram produced correctly with correct configuration and axis labels.
Task 4 100k	python3 cw2.py -t 4 -f sample_100k_lines.json	Pass	0.79	Top 10 readers in terms of reading time printed correctly and in order.
Task 4 3m	python3 cw2.py -t 4 -f sample_3m_lines.json	Pass	23.23	Top 10 readers in terms of reading time printed correctly and in order.
Task 5d 100k	python3 cw2.py -u 00000000deadbeef -d 100806162735-00000000115598650cb8b514246272b5 -t 5d -f sample_100k_lines.json	Pass	2.92	Documents also liked by readers of document input as parameter printed correctly in order of number of readers in descending order.
Task 5d 3m	python3 cw2.py -d 140213232558-bdd53a3a2ae91f2c5f951187668edd50 -t 5d -f sample_3m_lines.json	Pass	31.99	Documents also liked by readers of document input as parameter printed correctly in order of number of readers in descending order.
Task 6 100k	python3 cw2.py -u 00000000deadbeef -d 100806162735-00000000115598650cb8b514246272b5 -t 6 -f sample_100k_lines.json	Pass	0.99	Document graph correctly shown with input user and doc uuid highlighted.

Task 6 3m	python3 cw2.py -d 140213232558-bdd53a3a2ae91f2c5f951187668edd50 -t 6 -f sample_3m_lines.json	Pass	42.45	Document graph correctly shown with input doc uuid highlighted.
Task 5d 100k with invalid user uuid	python3 cw2.py -u invalid -d 100806162735-00000000115598650cb8b514246272b5 -t 5d -f sample_100k_lines.json	Pass	0.85	Program still executed and output list of most popular docs read by reader of input doc. Invalid uuid caught in also_likes function and set to None.
Task 5d 100k with invalid doc uuid	python3 cw2.py -d invalid -t 5d -f sample_100k_lines.json	Pass	0.8	Program execution did not exit, invalid input did not raise any exceptions. There are simply no matches.
Task 5d 100k with missing user and doc uuid inputs	python3 cw2.py -t 5d -f sample_100k_lines.json	Pass	0.8	Program execution did not exit, missing input did not raise any exceptions. There are simply no matches.
Task 5d 100k with invalid file name	python3 cw2.py -u 00000000deadbeef -d 100806162735-00000000115598650cb8b514246272b5 -t 5d -f invalid_file	Pass	N/A	File not found exception successfully caught, message printed out and program execution successfully terminated.
Task 5d 100k missing file parameter	python3 cw2.py -u 00000000deadbeef -d 100806162735-00000000115598650cb8b514246272b5 -t 5d	Pass	N/A	Argument of None caught in if condition, message printed out and program execution successfully terminated.

7. Reflections

While developing this application in Python 3 I found the list and dictionary comprehension and higher order function capabilities particularly useful. I found they helped me speed up my code writing with the added benefit of often being a faster method of implementation than traditional loops. I used list comprehension frequently throughout this application and it allowed me to significantly reduce the size of the code base while maintaining good performance.

In addition to the above, the range and depth of Python 3 libraries was also a great help. For example I used numpy to count the unique instances of items in an array in order to plot a histogram. To do this I used the `numpy.unique` function with `return_counts=True` and then created a dictionary of zipped values returned from that function. This eliminated the necessity to write multiple iterative loops and comparator statements making my code much cleaner and undoubtedly faster.

One limitation of my application is that I didn't leverage Object Oriented Programming as much as I should have, favouring more a functional approach. While the application still fulfils the majority of requirements, utilising classes and objects more could have helped me create more separation between logically similar elements of the application and potentially reduce the code further while maintaining speed and robustness.

In terms of the usability of Python 3 for this application I think it is incredibly well suited. It is fast, lightweight and is fast to build in comparison to a system language such as C#. A scripting language such as Python 3 is ideal for the kind of data processing required in this application where the same operation is required on the data many millions of times. This makes it easy to see why it is popular for back-end data processing and to provide glueware between larger more systems-based applications.

8. Conclusions

Overall, I enjoyed this project as I find Python 3 an agreeable language to work with and the availability of such a wide range of packages means you can find creative way to solve problems while also trying to increase performance.

I am most proud of the efficiency I managed to achieve in this application, with it being able to process and file of 3 million records for the most computationally intensive task in under a minute. Stemming from this I am also proud of the concise nature of the code for this application; by using list and dictionary comprehension where possible, I managed to keep the overall codebase relatively small for this application.

I am disappointed that I was not able to dedicate more time to this project due to being ill as I would have liked to improve my overall program structure and include the use of more advanced language features. However, this is learning I will take forward and use to build on in my next project.