DEVELOPMENT OF A DATA ANALYSIS TOOL IN PYTHON 3

Industrial Programming Coursework 2

Paterson, Stuart
SP96

Introduction

The following report documents the creation of a simple data analysis tool in Python 3.7. The program reads data in from a single JSON file then performs one of several operations on it. The operations are as follows; count the countries a document has been viewed from, count the continents a document has been viewed from, count all the full browser information a document has been viewed from, count the short browser title a document has been viewed from, display a sorted list of relevant 'also likes' documents for a document and optionally a visitor and finally display this 'also likes' relation for a document as a graph. The program should be able to run from both the command line and a graphical user interface(GUI). The majority of the code was written in JetBrains PyCharm Community Edition 2018.2.4 and the program was tested on both Windows and Linux machines.

A user guide describes how to carry out each of the tasks and what the expected results are. Use of both the command line interface and the GUI are described here. The developer guide provides insight into how the tasks are executed behind the scenes and why the code has been designed the way it is. An extensive list of tests is included with their expected and actual results. The effectiveness of Python for developing this program is reflected on, along with what was learned from the feedback on coursework 1.

Requirement's Checklist

Requirement	**Status**
1 – Use Python 3	Entire program is written in python 3.
2a – Take string of document ID and display histogram of countries in viewers	Fully complete.
2b – Take data from 2a and show histogram of continents	Fully complete.
3a – Histogram of all browser identifiers and viewers	Fully complete.
3b – Histogram of short browser identifiers	Fully complete.
4a – Take document UUID and return all visitor UUIDS of readers for that document	Fully complete.
4b – Take a visitor UUID and return all document UUIDs that have been read by this visitor	Fully complete.
4c – Implement also like functionality to take document UUID and optional visitor UUID.	Fully complete.
4d – Also likes using sorting to return top 10 document UUIDs as result	Fully complete.
5 – Also likes graph to display relationship between input document and all also like documents. Shorten document and visitor UUIDs to last 4 hex digits	Fully complete. Will display input document and (optionally) visitor highlighted in green.
6 – Develop a simple GUI based on reading user inputs from other tasks with buttons to process data	Fully complete.
7 – Command line usage	Fully complete.

Design Considerations

As the program will be processing a large amount of data efficiency is important. Ideally the program should only parse the data set once to retrieve all the information it needs to complete a given operation. Initially the program loaded all the data into memory then manipulated it there. This was useful for development using small data sets but is not viable when data sets begin to grow. For large data sets there simply won't be enough space in memory to store them and loading the data and then querying it means the data set is parsed twice instead of once. Ultimately loading into memory was replaced with a generator which allowed querying of the json data line by line. In the final program all tasks retrieved all the required information from the raw data in a single pass.

As the exact format of each data entry was not set in stone it was important to verify an entry included the required data before examining it. This meant checking each json line included all the required fields before querying their values. This way, time is not wasted processing lines that are not relevant.

An object oriented approach was taken for structuring the code to make it maintainable and readable. From the beginning of development separate classes were created and developed to be highly cohesive and loosely coupled. For example the class with a method for drawing the histograms is passed only the required data when the method is called and does not access other classes to get it which gives low coupling. Only functionality related to creating graphical displays is included in this class giving it high cohesion.

Originally the main functionality for starting each task was at the top level of the program in cw2.py. I created a new class called TaskController to manage this functionality instead so that tasks could be created by both the GUI and command line arguments easily. This structure is better encapsulated and will be easier to maintain.

User Guide

Users may receive results of the data analysis through either the command line or the graphical user interface(GUI).

Command Line Usage

The program can be run from the command line by a user provided several arguments are passed to it. The program is run using a command in the following format:

```
python cw2.py -f filename -d doc_uuid -t task_id -u user_uuid
```

The arguments can be passed in any order but only one value can be passed to each argument.

Table 1 shows which arguments are used by each task. If no document or filename is provided then the task will not run. It is worth noting at this point if the program is run with absolutely no arguments then the GUI will launch, which is explained in detail later in this report.

Task	Requirements	Optional
2a	Task ID, Doc ID, filename	None
2b	Task ID, Doc ID, filename	None
3a	Task ID, Doc ID, filename	None
3b	Task ID, Doc ID, filename	None
4d	Task ID, Doc ID, filename	Visitor ID
5	Task ID, Doc ID, filename	Visitor ID

Table 1: The required and optional arguments for each task

The data file being read by the program must be in the project folder and the full name including extension type must be passed as the argument. For example "sample_100k_lines.json".

Task 2a will display a histogram showing the countries a document has been viewed from. If the document has no views then the histogram will be blank. Supplying a visitor uuid will not affect this task.

Task 2b is similar to 2a but will show a histogram of continents rather than countries. Again if document has no views then the histogram will be blank. Again supplying a visitor uuid will not affect this task.

Task 3a will display a histogram showing the browsers a document has been viewed from. This task takes into consideration all browser information meaning the labels are very verbose and include specific information about every browser configuration. The length of these names usually means the labels on the histogram will not display correctly on the screen.

Task 3b simplifies the output from 3a to display a histogram showing the browsers a document has been viewed from grouped by a shortened browser name. For example *"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/33.0.1750.117 Safari/537.36"* would simply be displayed as *"Mozilla Safari".* This will result in fewer groups in the data and a more readable output.

Task 4d will retrieve the top 10 relevant documents for the document uuid passed to it. If there are less than 10 documents it will simply return all of the documents. In either case the documents are returned in descending order with the most relevant document first. This also likes functionality is based on how many people who viewed a given document also viewed other documents. The document with the most common visitors with the input document will be first in the list and so on.

If a visitor uuid is provided then that visitor's documents will not be added in to the also likes list as they have already been seen. The result of this task is just a list of document titles in descending order of relevance which will be printed out to the command line.

Task 5 produces the same 'also likes' functionality of task 4d but instead displays it graphically. The user enters the same information as task 4d and the program will calculate the same list of relevant documents. These documents are then displayed in a format like the example in Figure 1.

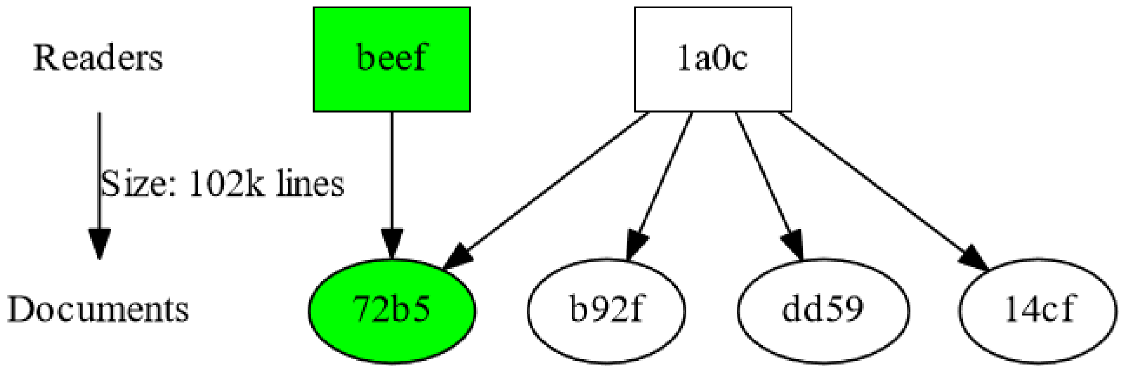

Figure 1: An example graph output from task 5

The visitors are shown along the top in rectangular node and the documents are shown in ellipses. Both visitor and document uuids have been shortened to the last 4 hex digits. The document that was provided will be shown highlighted in green and if a visitor uuid was provided they will also show green. The graph is saved as a pdf into the project directory. If a visitor is provided the graph will not show other documents that visitor has viewed. The graph will also now show any readers of the documents that don't contribute other documents as they do not provide any 'also likes' functionality.

GUI Usage

The GUI can be used to carry out the same tasks as the command line and will yield the same results. To launch the GUI the user simply has to run the program from the command line without passing any arguments. The command to start the GUI is shown below:

```
python cw2.py
```

An image of the GUI is shown in Figure 2. The user simply enters the document uuid of interest, the name of the file where the data is stored and optionally a user uuid. Then they press the button corresponding to the task they wish to carry out. The GUI will close provided they have entered the required information and the task will run. If the GUI is closed without starting a task then the program will exit.

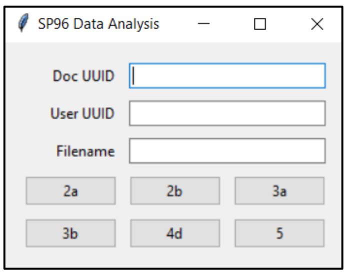

Figure 2: The display of the unpopulated GUI on a Windows machine

Developer Guide

The argparse module was used to read the arguments the user passes when running from the command line. This allows arguments to be easily created and described with statements like the following:

```
parser.add_argument("-f", "--filename", type=str, help="Name of file storing json")
parser.add_argument("-u", "--user_uuid", type=str, help="User id of visitor to search for")
parser.add_argument("-d", "--document_uuid", type=str, help="Document id of the document to search for")
parser.add_argument("-t", "--task_id", type=str, help="ID identifying which task the program should carry
out.")
```

Help statements were created to allow new users to call the help command and receive information about how to run the program. If the user doesn't pass the required arguments (filename, doc_uuid and task_id) then the program will print an error message directing the user to the help statement.

Before checking the arguments the user has passed the program first checks if no arguments were passed. If this is true then the GUI is run instead of the command line execution. The class that actually executes the tasks is called TaskController. If the GUI is not run then a TaskController is created with the arguments from the command line. If the GUI is run then when the user inputs data and a task the GUI closes itself and creates a TaskController with the arguments input by the user into the GUI.

Before trying to analyse the data the first thing the program does is try to locate the file specified by the user. If it cannot be found or opened the program will exit as there is no data to analyse. If the file is opened then it can be parsed for the analysis. The data in the file is examined line by line using a generator and yield expression as seen below.

```
def genData(self):
    try:
        with io.open(self.filename, 'r', -1, 'utf-8') as f:
            # Code based on https://stackoverflow.com/questions/2835559/parsing-values-from-a-json-file
            for line in f:
                self.lineCount += 1
                yield json.loads(line)
    except FileNotFoundError:
        print('The file \'%s\' does not exist' % str(self.filename))
        sys.exit()  # Exit the program if there is not data to process
```

This generator is then simply iterated over in a for loop to process every line in the data. The object containing this generator is passed to the other methods which then use it to access this method. The data should not be loaded into memory as large data files may not fit. The json library is used to load each line and the result of this is a dictionary containing the key-value pairs in the json file. This is ideal as it allows other methods querying the data to access it directly.

All of the methods involve looping through the data set to extract the required information. The first thing to do when querying a line is to establish it contains the required data. This is done by checking if the dictionary for that line contains the keys for the data required. Next it must be established if the line describes a 'read' event as these are the only event of interest for this analysis package. Finally the data can now be queried to check things like whether the document id matches. The below code is an example of parsing the data in the method that counts country visits. It first checks the required information is present, then that the event type is read and finally it checks the data to see if the document is the one of interest. If the document is a match it will perform the country counting functionality.

```
for entry in data.genData():
    if 'subject_doc_id' in entry and 'visitor_country' in entry and 'event_type' in entry:
        if entry['event_type'] == 'read':
            if str(entry['subject_doc_id']) == docID:  # cast to string needed as input is string
```

The methods for counting country and counting full browser names work in a similar manner. They query the data line by line as described above to collect all relevant data. When they find relevant entries that match the document id being searched they add a new entry to the results if it is a new result or increment it, if it exists already. The code snippet below shows an example of this behavior in the country counting method. The result of both functions is a dictionary containing all the unique results and how many times each occurred. A dictionary was used as it is easy to lookup entries by key to edit their value. It is also easy to add new elements due to the lack of order which is important as the dictionary is created empty then populated on the fly. The data was not required in any order so this was not a problem. A list of tuples could've been used for the data but it would've been slow to lookup entries so that they could be incremented and it would allow duplicate keys which is not desirable.

```
if entry['visitor_country'] in self.countryCounts:
    self.countryCounts[entry['visitor_country']] += 1
else:
    self.countryCounts[entry['visitor_country']] = 1
```

The methods for counting continent and short browser name take the output from counting country and full browser name respectively. They then essentially group the data in the output of these results into fewer unique results. Count continents groups all the data from the countries by which continent that country is in. The pycountry-convert module is used to convert the country codes to continent codes.

To extract the short browser name from the full browser name the string is split. Each section of the browser name is separated by a '\' so first it is split on the backslash character. The first and second last sections of the string contain the important information so these are selected by index. Then the contents of the second last section is split again, on whitespace this time, to extract only the name which is the last entry. The code below performs this operation in a single line. This implementation is very dependent on the input format of the browser data and will give wrong results if the data is not in the right format.

```
browserName = browser.split('/')[0] + ' ' + browser.split('/')[-2].split(' ')[-1]
```

Both counting continents and short browser names use the technique described above for counting unique entries and returning a dictionary of unique results and their counts.

Matplotlib was used for generating the histograms for tasks 2a, 2b, 3a and 3b. A simple method was created for creating a histogram from a dictionary (Wolfgang-Loidl, 2018). Matplotlib was used as it allows displaying of the results quickly and easily. If it was desired to control the display of the data in detail then defining your own module to do this may be more appropriate but for this program the results were the focus, not the appearance. The keys of the dictionary become the bars on the chart and then the values are the count of how many times that key's value occurred.

Simple methods for collecting all documents for a visitor and collecting all visitors for a document were written for part 4a and 4b. Each one loops through the data set and checks for the visitor or document uuid they are searching for and adding their document or visitors to a list which is returned at the end. The first implementation of also likes used these methods but was quickly replaced due to it having very poor performance. Because these methods were written in isolation originally they both looped through the data set once every call collecting the important information.

When combined into the 'also likes' method below this resulted in parsing the entire data set multiple times (specifically equal to the number of visitors for the document being searched for plus one).

```
def alsoLikes(self, data, docID, visitorID = None, sortFunc = sortByValues):
    """Returns a sorted list of documents relevant to a specific document.
    Optionally takes a visitor who's documents should be omitted."""
    result = {}
    for visitor in self.commonVisitor(data, docID, visitorID):
        if visitor == visitorID:
            continue
        for doc in self.visitorDocs(data, visitor, docID):
            if doc in result:
                result[doc] += 1
            else:
                result[doc] = 1
    return sortFunc(result)
```

The improved version of also likes only traversed the data set a single time. When parsing the data it created a dictionary with the document id as key and a list of visitor ids who had visited that document as the value. At the same time it also created a dictionary with the visitor id as a key and a list of all documents they had visited as a value. These two dictionaries contain all the information required to calculate the also likes results and can be created with a single pass of the data set. It is important to capture the data in two separate dictionaries for speed. It is possible to capture all the information in a single dictionary, for example store all documents as keys with lists of visitors, and calculate the documents for each visitor from this but it would be very slow and inefficient. This method uses an 'in' check to make sure it does not add duplicates to the lists. In hindsight a set would have been more appropriate than a list as it would enforce this constraint without an explicit check.

There are in fact two 'also likes' methods as different data is required to be produced for text output and graph output. Part 4d only requires a list of documents sorted by most views so it returns a list of document names. To produce the graph a dictionary is needed to describe the relationships between documents and visitors. Both methods require the same two dictionaries to produce the also likes relationships as described above so this was encapsulated into its own method called alsoLikesData. It returns a tuple which are the two dictionaries.

The method for part 4d just counts how many times each document that has been viewed by a viewer of the selected document has been seen by common visitors. It ignores the selected document and if a visitor has been specified it ignores their documents. The method takes a sorting function as a parameter. This higher order function is used to sort the result before returning the list of documents. In this program it has been set to a default that sorts the documents in descending order by number of views. This method does not actually receive the raw data, just the two dictionaries described above.

Dictionaries are not ordered so sorting the result before returning required a slight workaround. In order to create a sorted result the dictionary was changed into a list of tuples. Itemgetter is used to specify sorting by the values from the dictionary rather than the keys (Fiocco, 2017). The reverse parameter has to be set to true so that the list is in descending order of matches. As only the document uuids were wanted without the number of readers a generator expression is used to make a list from only the ids (Pietzker, 2015).

The method for part 5 uses the dictionary of documents with lists of visitors to retrieve the list of all visitors for the specified document. Then it loops through the dictionary containing visitors and their lists of documents and extracts only the ones for visitors who visited the specified document. The result is a final dictionary with visitor id as the key and lists of documents they visited as the value. Visitors who have only visited the document and no other documents are not included as the purpose of also likes is to show other relevant documents and they contribute none. If a visitor uuid is specified then all other documents this user has visited other than the specified one are omitted. This dictionary is then passed to the method which creates the graph. It creates every visitor and document as a node then uses the key value relation of the dictionary to create edges between visitors and their documents. The render method from the graphviz package is then used create a pdf of the graph and save it to the system (Graphviz, 2018). This package saves having to manually create any dot code from the data and then render it making the method very simple.

The tkinter module was used to create a simple GUI. The simple feet to meter example GUI was used as the basis for the code and was expanded to include all the required functionality (Loidl, 2018). Three text fields allow the user to input document uuid, user uuid and the filename where the data is stored. 6 buttons are mapped to the 6 tasks and selecting one will close the GUI and begin the task. If the user has not entered any data the GUI will not close and no task will begin. The user must at least enter a document uuid and a filename. Each button triggers the same method but passes a different task id to this method. This method then creates a TaskController which will execute the task and display the results. New tasks could be added and easily triggered by creating a new button but if there were too many tasks the GUI will become messy and covered with buttons. In this case it should be adapted to include something like a drop down list for selecting the task which will make the appearance much more concise. Ideally the GUI should include a label with some detail of how to use it, i.e which fields are required and what task each button executes. As currently any users will know what task number corresponds to which task this wasn't an issue but it would not be appropriate for software that is to be distributed. The GUI is closed when the user enters their information and selects a task. Destroy is used to close the window instead of quit so that it exits the mainloop that is running. (Vascellaro, 2013).

One of the main limitations of this program is its lack of optimisation. The maximum size of data set it was tested with was 3 million lines but if even larger data sets were used processing time would increase greatly. A technique that may have been useful is parallel programming which would allow multiple areas of the data to be parsed simultaneously. A module like 'threading' or package like 'multiprocessing' could be used to achieve this and it would cut down the time the program spends looping through the data set to extract the results. For a simple example you could have two methods parsing the data set, one doing the first half and one starting halfway through and doing the second half. Best case scenario is that it takes half as long to traverse the data however in real life this won't be the case as starting each process will have some overhead. The issue with this parallel approach is merging the data after it is collected. One process may collect 3 documents a visitor has read and the other may collect 2 documents where one of them is common. After the processes are finished the two data sets for this visitor must be merged together so that there are no duplicates but no data is lost. This will add some additional overhead but for large data sizes will still be much faster.

Testing

Test	Expected Result	Actual Result
Command line		
No task argument passed to command line.	Error reports that a task is always required.	Error displays saying ‘Invalid arguments passed. For instruction on usage use the -h argument or to run GUI pass no arguments’.
No document uuid argument passed to command line.	Error reports that a document uuid is always required.	Error displays saying ‘Invalid arguments passed. For instruction…’
No filename argument passed to command line.	Error reports that a filename is required.	Error displays saying ‘Invalid arguments passed. For instruction…’
Invalid task id passed as task argument.	Error reports that task is invalid.	Error displays saying ‘Task ‘*argumen*t’ is not a valid task. Please type -h for help on program usage.’.
Filename that does not exist passed as filename argument.	Error reports the file does not exist.	Error displays saying ‘The file ‘*argumen*t’ does not exist.’.
Task 2a called with a doc uuid that is not in the data set.	Error reports that the document is not found in the data.	Empty histogram displays.
Task 2b called with a doc uuid that is not in the data set.	Error reports that the document is not found in the data.	Error prints ‘No data to count’. Empty histogram displays.
Task 3a called with a doc uuid that is not in the data set.	Error reports that the document is not found in the data.	Empty histogram displays.
Task 3b called with a doc uuid that is not in the data set.	Error reports that the document is not found in the data.	Error prints ‘No data to count’. Empty histogram displays.
Task 4d called with a doc uuid that is not in the data set.	Error reports that the document is not found in the data.	Error displays ‘Document ‘document_uuid’ not found in data set.
Task 5 called with a doc uuid that is not in the data set.	Error reports that the document is not found in the data.	Error displays ‘Document ‘document_uuid’ not found in data set.
Duplicate arguments passed in command line when running program. For example (-t 5a -t 2a	Error reports arguments are wrong.	Program runs as normal using only the last argument of each type.
More than one value passed to arguments in command line. For example (-t 5 2a)	Error reports too many values	Error display ‘unrecognised arguments: *extra argument*’
Try to create graph for part 5 while target file is open on machine.	Error reports that file can’t be saved.	Error prints ‘Error: Could not open "*dot_graph.pdf*" for writing : Invalid argument Cannot save graph to file with filename ' *dot_graph.pdf* '.

		Check if the file is open or locked
Invalid user uuid entered for any task that requires a user uuid.	Error reports that user is not in data set.	Programs completes as if there were no user uuid as it finds no matches.
User uuid entered for any task that does not require a user uuid.	Error reports that a user uuid is not required for this task.	Task runs as normal unaffected by user uuid.
Program tested on different files sizes from 100k lines to 3m.	Program takes longer to process larger data sets but completes without any problems.	As expected.
Count continents called on a data set containing an entry with an invalid country code that does not match to a continent.	Error reports this anomaly to user.	Error prints 'No continent for country with code *code*'. Entry not added to continent counts.
Also likes test on a document that has readers who have only read that document.	Readers who have only read the input document should not be displayed as they contribute no related documents	As expected
Empty arguments passed to the command line. (e.g *python cw2.py -t -f -d*)	Error reports that arguments are required.	Usage guide displays along with error of the form "*error: argument -t/--task_id: expected one argument*"
Data set with entry that has browser name in very different format to current data.	Browser name parsed from string.	Browser name will be parsed incorrectly or not at all. First part of short browser name more likely than second to be captured.
Trailing whitespace after arguments in command line	Program runs as normal	As expected
GUI		
Trailing whitespace after arguments in GUI.	Program runs as normal	Whitespace is read in as part of name and results will be incorrect. (Should use strip() on input)
Push button for task without entering a document uuid, a filename or both.	Program will not run. Error displayed to user stating required arguments.	Program does not run. GUI stays open and 'Please enter at least both a document uuid and a filename' is printed to the command line.
GUI closed without starting a task.	Program exits.	As expected.
Invalid document uuid not in data set entered into GUI and program run	Same as command line.	Same as command line.

Reflections on programming language and implementation

The most commonly used data structure in this program was the dictionary. The dictionary is perfect for storing data whilst capturing a relationship, for example all the documents a visitor has read. It also perfectly matches the structure of JSON because it consists of key value pairs which made it ideal for storing the raw data. Dictionaries cannot include duplicate keys which is usually advantageous as we do not want to store duplicate entries for one document or one visitor.

Python's dynamic typing made it easy to quickly write a flexible program for reading and managing the data. Being able to create the data structures on the fly and have them detect the types meant time didn't have to be wasted ensuring data structures with the exact types were initialized and everything was cast to these types where required. The downside of this dynamic typing was the lack of control and security for some operations. For example a data list may accept inputs of any type but later in the program when that list is iterated over and each value compared an exception may be raised if the type is incorrect. This is quite common because Python is a strongly typed language so something like comparing an int to a string without casting the int to a string would raise an exception. The code snippet below shows an example that would raise a type error in Python.

```python
try:
    if 'test' > 1:
        print('this won\'t work')
except TypeError:
    print('Python doesn\'t allow this comparison between str and int')
```

This combination of dynamic typing and strong typing means that although working code was created very quickly, to ensure security a lot of time and lines of code are required to carry out type checking and ensure values are cast where required. In the case that very heavy type checking and casting are required the code will quickly become difficult to read due to the mass of expressions that serve no purpose other than to check types. In this situation Python is probably the incorrect language for the program and a statically typed language should be used. In this program type checking was only carried out where deemed sensibly necessary. Type checking was used to avoid foreseeable exceptions but not to prevent every possible type mismatch. The decision of where to use type checking was based on common sense and the likely possible data inputs.

Many libraries were used to aid the development of this program. Libraries like graphviz and matplotlib made it very fast to create a working program without having to spend time creating all the required tools. By default python includes a large collection of modules such as argparse and tkinter. These default modules provide a lot of powerful functionality without even having to install anything additional on the system. The important factor when using external modules and libraries is to consider what machine's the program will be run on. If the program must be portable and run on different machines care must be taken to ensure that the program can access these libraries where it is run. For this data analysis package the external modules like pycountry-convert were included locally in the project directory and then this directory was added to the python path so that python could find these modules. An alternative method is to include a requirements file which will install the required packages on the target machine before executing the program (PIP, 2018).

The advantage Python for data processing was that it didn't take long to create a program that could query the entire data set and return results. The time to go from raw data to the desired results was short due to making use of existing libraries. However Python also has a major disadvantage for data processing which is speed. In the case that a program is being developed to analyse large volumes of data optimisation will often be a key focus. When data sets get very large processing time for the current program would become unmanageable. There is room for further optimisation within this

Python program, for example the pandas library could be used for processing the data. However Python does not allow the user direct access to memory management as it is all done by the Python interpreter (Python, 2018). Without the ability to directly manage memory the optimisation that can be done on the program is limited and therefore Python would be poorly suited for developing a data analysis program where performance was key. Typically a systems language like C++ has more access to system features like memory and provide more control at the cost of more programming needing to be done.

Python is interpreted rather than compiled which provided some advantages and disadvantages for the development of this program. This means that Python compiles the program at line by line at runtime rather than compiling the entire program then running it. When writing the code and adding new functions it was faster to test as only the entire program did not need to be recompiled after every change. However a compiled language may create a program that executes faster because it does not need to do any compiling while running. This of course comes at the downside of having to wait for it to re-compile after every change. An issue with interpreted languages like python is that errors can go unnoticed if their case is never triggered. For example the code snippet below shows a python program that would compile and run with no problems:

```
x = True
if not x:
    x = "test" + 123 + 2.0 * False
```

Although there is a clear type error in the then branch Python will not raise any sort of error because the interpreter never reaches this line of code. The problem is that if something in the program changed the value of x such that this line was then interpreted it would throw an error at that point. A compiled language would check all the code and an error like this would prevent it from compiling. Even a clear syntactic error like the following code snippet will not cause any error in a python program. However most IDEs will show this as an error.

```
x = True
if not x:
    fhdkspcvuf12
```

Higher order functions are a useful feature that are easy to use in python and can be very powerful. For example the sorting in the ‘also likes’ method was done by a higher order function. Higher order functions allow functions to be treated almost like variables and importantly allows them to be passed to other methods as arguments. Higher order functions allow reuse of code by passing methods to other methods and this can improve maintainability and also provide better encapsulation. The clean encapsulation makes the code more readable and maintainable because a method can be passed to several other methods as a higher order function and only needs to be edited in one place.

Python supports object oriented programming but is not as strict about encapsulation as some other languages. Fields cannot be made private with the same security as a language like C# and can be accessed with a simple workaround. The value can still be looked up in the ‘__dict__’ for that object by inserting an underscore at the start of the field’s name. This lack of private fields means they can be accessed from anywhere which may lead to high coupling as other classes can use any field from another class. This is poor programming practice as these dependencies are difficult to see without searching through the entire codebase and changes to one class may cause problems in the coupled classes.

What did I learn from CW1

If I found myself reusing code I tried to encapsulate it in methods wherever possible to keep my classes short and readable. I took care to ensure that by doing this I wasn't creating suboptimal code. For example when implementing the also likes functionality using the methods I wrote for part 4a and 4b would've resulted in inefficient code as I explained earlier so I wrote a new method instead.

Despite Python not being as strictly object oriented as C# I tried to continue my style from the first coursework of encapsulating functionality and data into classes. This keeps individual classes short and increases the readability of the code making it easier to maintain. Although Python doesn't support private fields the same way C# does I tried to store data in the classes that required access to it.

In some of the code in the previous coursework I mixed functionality inside methods that should not have been together. For example some mixtures of GUI code and program logic. I tried to write more cohesive classes for this coursework to ensure this doesn't happen. For example the method for drawing the dot graph does not apply any changes to the data it receives, it is fully processed beforehand.

I tried to ensure no exceptions would go uncaught in my program by using try-except blocks. When I used these I tried to always catch the specific errors I expected first before using a generic exception. I tried to always report some sort of meaningful information when an exception is caught so that it is easy to diagnose what went wrong and why. Some areas of the program may still throw exceptions if the input data is drastically different type to what is expected but I tried to be pragmatic about handling expected exceptions without going over the top with checks that may make the code difficult to read and understand.

The feedback on coursework 1 said I could've included more corner cases in my testing so I have tried to report a more comprehensive list of tests for this program.

I included a copyright license in my work for this program which I did not do for coursework 1. I simply chose a Creative Commons CC-BY license which allows people to share or adapt my work provided they give appropriate credit to me (Creative Commons, 2018). I realise it is important to control how code I write will be treated and distributed and had not previously considered this before the first coursework. I included comments in the top of every python file which direct readers to the license.

For this report I've tried to implement feedback I received on the first coursework. I've pasted the code into the text in a more readable format than using screenshots. The dark highlights make it easy to see which areas are code snippets and which are text. I've kept a complete list of references to any sources I used throughout the project and included these in this report.

Conclusions

In conclusion a data analysis package was created in Python 3.7 which met the project specification fully. Care was taken to ensure good quality code was written and object-oriented design was used. Python was good for creating this tool quickly by using existing modules and libraries, which allowed a lot of functionality to be implemented in a short time span. However if the program was to be used on much larger data sets Python may limit the optimization of the program due to lack of access to memory management. Despite this, care was taken to ensure the program only processed the data set a single time for each operation. A command line interface and GUI were both created for using the program and extensive testing of these has been reported.

I am proud of the design and structure of this program. I tried to write highly cohesive and loosely coupled classes that contained short readable methods and were well commented. I tried to ensure I accounted for exceptions and handled unexpected behavior where possible and I'm pleased I implemented all of the required functionality.

I tried to embrace Python's 'glue-wear' type nature and use as many libraries and modules to help in the design of the program. For example I elected to use the pycountry-convert module to convert the country code to continent and the graphviz module for generating the dot code for task 5. This allowed fast development of the functionality which gave me more time to work on code quality.

I tried to handle any limitations I discovered when I found them but one thing I would've liked to have done better is optimisation. Although I ensured to only query the data set once there was likely still room for optimization within this program. Although run times are currently manageable if larger data sets were required the program would quickly become difficult to work with. I also should have looked to use inheritance within the code to improve the structure.

The GUI for this program has a lot of room for improvement and could do with some instructions displayed and some error reporting if the user enters incorrect data. The program could also be improved such that multiple tasks could be run in a single execution of the program.

References

Creative Commons, 2018. *Attribution 4.0 International (CC BY 4.0).* [Online]
Available at: https://creativecommons.org/licenses/by/4.0/
[Accessed 11 2018].

Fiocco, D., 2017. *Stack Overflow: Python 3 sort a dict by its values.* [Online]
Available at: https://stackoverflow.com/questions/20944483/python-3-sort-a-dict-by-its-values
[Accessed 11 2018].

Graphviz, 2018. *Graphviz User Guide.* [Online]
Available at: https://graphviz.readthedocs.io/en/stable/manual.html
[Accessed 11 2018].

Loidl, H.-W., 2018. *Python Samples: feet2meter.py.* [Online]
Available at: http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/python_samples.html
[Accessed 11 2018].

Pietzker, T., 2015. *Stack Overflow: python - get list of tuples first index?.* [Online]
Available at: https://stackoverflow.com/questions/10735282/python-get-list-of-tuples-first-index
[Accessed 11 2018].

PIP, 2018. *Pip Documentation: Requirements Files.* [Online]
Available at: https://pip.readthedocs.io/en/1.1/requirements.html
[Accessed 11 2018].

Python, 2018. *Python Documentation: Memory Management.* [Online]
Available at: https://docs.python.org/3/c-api/memory.html
[Accessed 11 2018].

Vascellaro, S., 2013. *Stack Overflow: How do I close a tkinter window?.* [Online]
Available at: https://stackoverflow.com/questions/110923/how-do-i-close-a-tkinter-window
[Accessed 11 2018].

Wolfgang-Loidl, H., 2018. *Python Samples: Simple_histo.py.* [Online]
Available at: http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/python_samples.html
[Accessed 11 2018].