

Beni Iyaka H00181266



School of Mathematical & Computer Sciences

Department of Computer Science

Course F20SC: Industrial Programming

“Coursework 2: Data Analysis of a Document Tracker”

Beni Iyaka - H00181266

Bi34@hw.ac.uk

Professor: Smitha Kumar

December 3, 2015

Dubai

Table of Contents

1. Introduction.....	2
2. Requirement's Checklist.....	3
3. Design Considerations.....	4
4. User Guide.....	5
4.1 Views by country/continent - Option 2a/2b.....	6
4.2 Views by browser – Option 3a/3b.....	9
4.3 Reader profiles – Option 4.....	11
4.4 “Also likes” functionality – Option 5a/5b/5d/5e.....	11
5. Developer Guide.....	14
5.1 Formatting the JSON file.....	14
5.2 Implementing Functions	15
5.2.1 Views by country/continent - ViewByCountry(self, doc, user).....	15
5.2.2 Views by browser - ViewsByBrowser(self, task).....	17
5.2.3 Reader profiles - TopReaders().....	18
5.2.4 GUI	23
5.3 Main	25
6. Testing.....	28
7. Conclusions.....	29

1. Introduction

The purpose of this report is to describe step by step the functionality and the development process of an application that analyses data derived from a document tracker. It offers a guide to the users who want to use the application in order to extract useful information from publications hosted at the popular website <http://www.issuu.com>, as well as to developers who want to implement a similar project or understand the implementation of this one.

This project was developed using Python 3.4 which is the latest version, using PyCharm IDE that offers a very friendly developing environment under Windows 8. The development and testing of this application offered a great introduction to the Python language and its functionalities.

Python is considered a powerful scripting language and can be a great tool for any modern programmer since it offers flexibility and reusability under an object oriented prism, along with loose syntactic rules that make it suitable for rapid prototyping when execution speed is not critical.

This report explains in detail the developing process and the different user scenarios. It was developed as a coursework for F20SC: Industrial Programming at Heriot-Watt University, for the Bachelors of Science in Computer Systems Honors program supervised by Professor Smitha Kumar.

2. Requirement's Checklist

Requirements	Result	Details
1. Python	Fully Completed	Python 3.4 was used for the deployment of the project. The IDE PyCharm was used on Windows 8.1
2. Views by country/continent	Fully Completed	a) Histograms that show the number of countries of the viewers is being displayed b) Histograms that display the continents of the viewers are being returned. In addition to the histograms both options also return dictionaries with Countries/Continents and number of occurrences.
3. Views by browser	Fully Completed	a) Histogram with the user agents of the viewers is being returned along with a printed dictionary. b) Histogram with the browser names of the viewers is being returned along with a dictionary.
4. Reader profiles	Fully Completed	A top 10 list is being printed that returns viewers Ids based on the time they have spent reading documents.
5. "Also likes" functionality	Fully Completed	"Also like" functionality implemented that returns top 10 document Ids based on : a) The most avid readers that have read these documents. b) The number of times a document has been read.
6. GUI Usage	Fully Completed	The application includes a GUI.
7. Command-line usage	Fully Completed	The application provides a command-line interface for executing different tasks.

All of the requirements have been met. The application can run from the command-line and the results are returned on the console windows along with the histogram.

3. Design Considerations

The application was designed to offer solutions on analyzing data from a document tracker.

The intended users are the ones that want to gain access to some interesting statistics that accompany a specific document. The application offers capabilities such as:

- Analyzing the number of countries and/or continents that a documented has been visited from. (Option 2a/2b)
- The readers that have read a specific document (Option 5a)
- The documents that a specific reader has read (Option 5b)
- Documents also liked by other readers based on their time spent reading (Option 5d)
- Documents also liked by other readers based on the times that have been read (Option 5e)
- The application includes a GUI that makes the user interaction friendlier.
- The application offers a very simple command line usage, where a user can specify a reader ID, a document ID and the task that he wants to execute and the results are being print on screen in a friendly, sorted manner.

Furthermore the use of graphical representations (histograms) provides an easy way for depicting data and extracting information from them. The use of these graphs is offered in options 2a, 2b, 3a, 3b.

Overall the user has to choose between the following options mentioned before and is more thoroughly described through the next chapters. It should be mentioned that for a user to make a document or a reader search they have to know the document's or the reader's id, as they are specified in the JSON file and unluckily they do not have a very user friendly format.

4. User Guide

In order to access and run the application a user, using a Command prompt, the user has to navigate to the directory where the application files are located. The files needed for the application to execute are:

- CW2.py (which is the main function of the program)
- CW2.json (which is the JSON files that includes all the data)

In addition to the above the intended user has to have matplotlib library installed on his/her computer along with a Python 3.4 compiler of course. Matplotlib libraries can be downloaded from <http://matplotlib.org/> and Python3 from <https://www.python.org/download/releases/3.0/>.

Once the user has completed all of the above, he is ready to execute the program.

In the example scenario the necessary files have been placed inside the path *C:\Coursework2*. That folder contains the above mentioned files.

```
C:\Users\lenovo pc\Desktop\Coursework2>CW2.py -d 140218233015-c848da298ed6d38b98e18a85731a83f4 -t 2a
```

Illustration 1: Command line execution



CW2.json	11/11/2015 10:45 ...
CW2.py	12/4/2015 02:50 PM
cwrk.json	11/9/2015 02:26 PM

Illustration 2: Windows folder

The application can be executed via the command line just by typing:

"CW2.py -u [a user's uuid] -d [document's uuid] -t [task number]"

For example if a user wants to search statistics for a document with a uuid: **140218233015-c848da298ed6d38b98e18a85731a83f4** and he wants to view the countries from which it has been accessed (Task 2a) he has to type

The application can be executed via the command line just by typing:

"CW2.py -d 140218233015c848da298ed6d38b98e18a85731a83f4 -t 2a"

4.1 Views by country/continent - Task 2a/2b

This option returns a histogram which is a graphical representation of the countries or continents that a specific document has been accessed from. The user has to specify the document uuid along with the task 2a→for countries or 2b→for continents and the histograms along with printed list are being returned.

Views by country → task 2a

For example

The document uuid is: 140218233015-c848da298ed6d38b98e18a85731a83f4
Will return the following histogram along with a list:

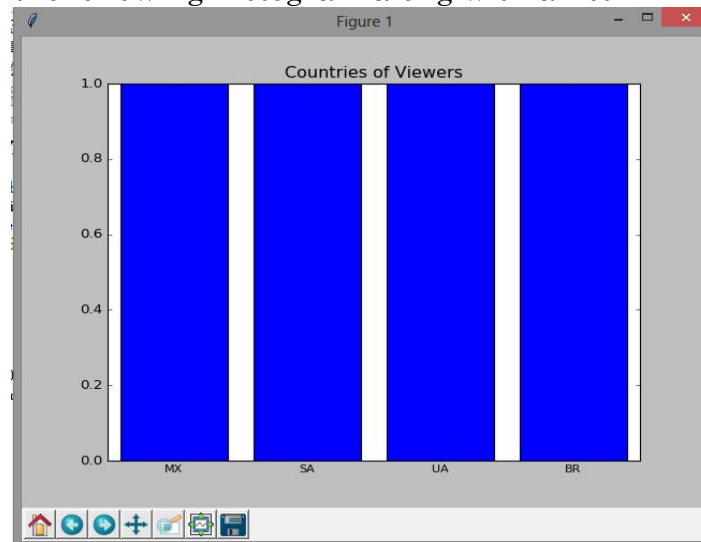


Illustration 4: views by countries

```
C:\Users\lenovo pc\Desktop\Coursework2>CW2.py -d 140218233015-c848da298ed6d38b98e18a85731a83f4 -t 2a  
Counter({'BR': 1, 'UA': 1, 'SA': 1, 'MX': 1})
```

Illustration 5: views by countries dictionary output

It can be seen that this specific document has been visited by 4 countries with the names MX, SA, UA and BR, one team for each.

The x axis shows the country codes and the y axis shows the number of occurrences.

With a different document uuid: 140226164301-db02e4587f79095920154d4cb44cec8a
Returns the following histogram and list:

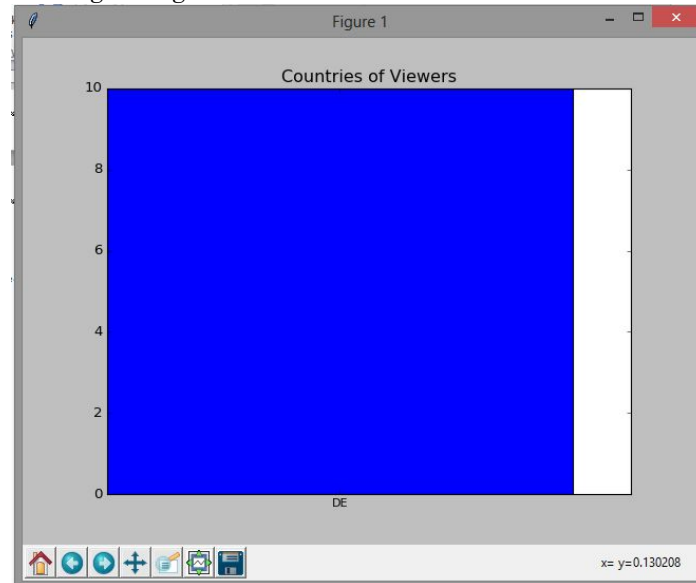


Illustration 6: Histogram of countries

```
Counter<<'DE': 10>>
```

Illustration 7: Views by countries dictionary output

In this example, the specific document has only been viewed from one country, DE but it has been accessed 10 times.

Views by continents → task 2b

In addition to the countries functionality, a specific document can be analyzed in regards to the number of different continents that has been accessed from.

For example in two scenarios above, we would expect the first document to return 4 different continents, since these countries belong to four different ones (North America, Europe, South America and Asia) while in the second scenario, the only continent that should be return should be Europe.

From the interface, the user has to input 2b in the task textbox.

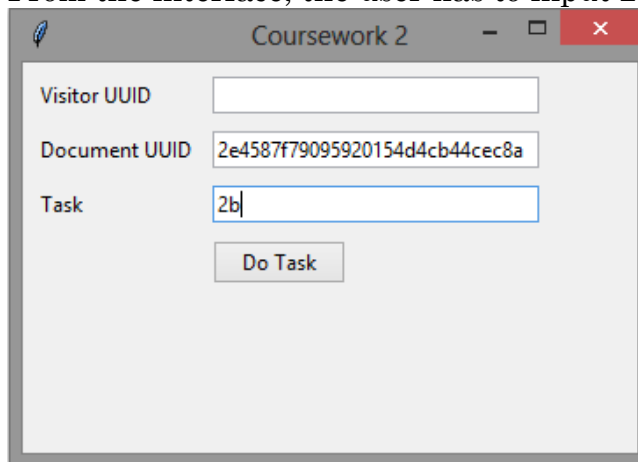


Illustration 8: Views by continents

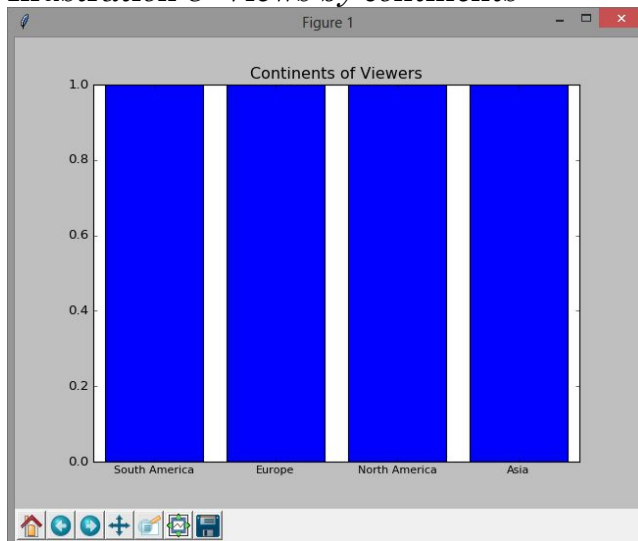


Illustration 9: Views by continents histogram

```
C:\Users\lenovo pc\Desktop\Coursework2>CW2.py  
Counter(<{'Asia': 1, 'South America': 1, 'North America': 1, 'Europe': 1}>)
```

Illustration 10: Views by continents dictionary

4.2 Views by browser – Task 3a/3b

In this task, the user can get information on the browser agents and browser names of all the visitors.

The result will again be displayed both in the form of a histogram and in a list. The difference between task 3a and task 3b is that:

- Task 3a returns the full agent names that can be hard to read but gives a view of the great variety between different browsers distributions and versions.
- Task 3b simplifies the previous results and returns information based on the browser name. More specifically the browsers that are being searched for are: Mozilla Firefox, Google Chrome, Safari, Opera mini and Internet Explorer. All other browsers go into the category named “Others”

Views by browser identifiers → task 3a

Using the document's uuid: 140218233015-c848da298ed6d38b98e18a85731a83f4

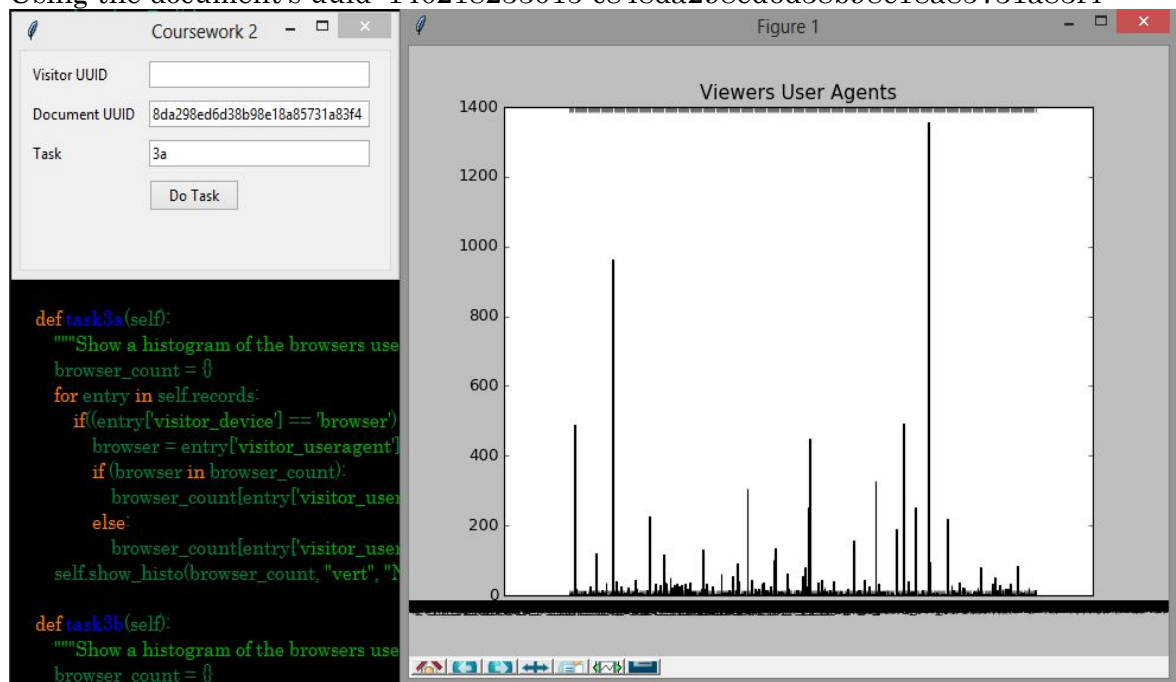


Illustration 11: Browser Identifiers Histogram

Views by browser names → task 3b

In the similar way like before, when a user specifies task 3b, he will get a histogram and a list of the browser names that have accessed all the documents.

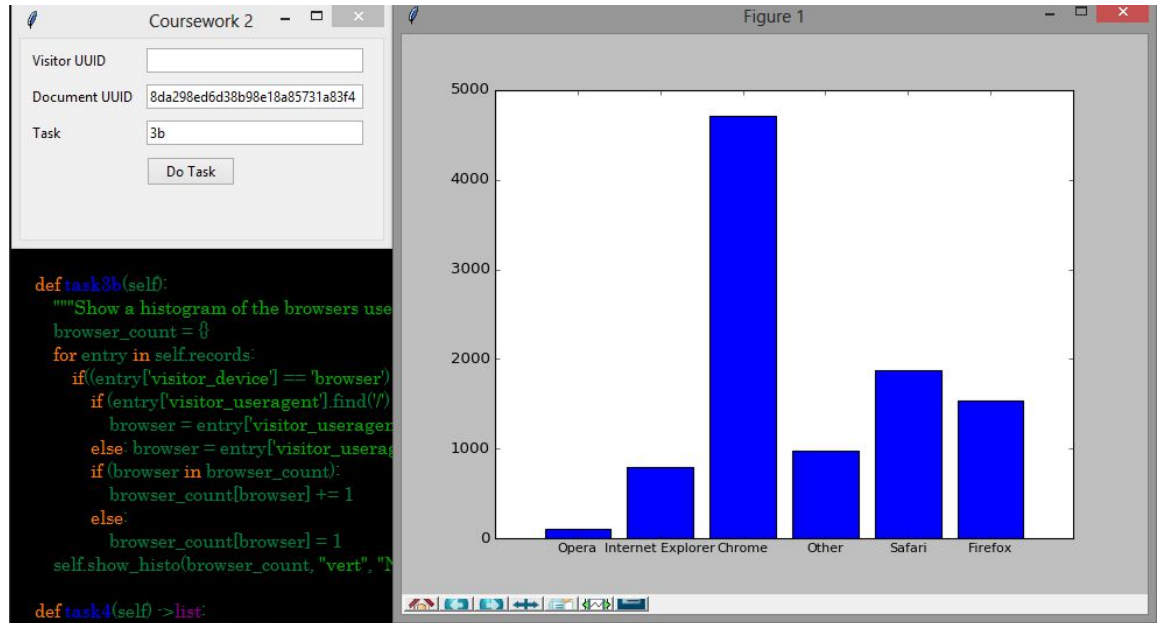


Illustration 12: Histogram of browser names

```
C:\Users\lenovo pc\Desktop\Coursework2>CW2.py
Counter({'Chrome': 4720, 'Safari': 1869, 'Firefox': 1534, 'Other': 977, 'Internet Explorer': 796, 'Opera': 107})
```

Illustration 13: browser names dictionary

The above information is very useful since we can see the popularity among browsers and which one is being used more to read documents.

4.3 Reader profiles – Task 4

When starting task 4, the user get the top 10 list of the most avid readers. That is, in order words, the readers that have spent the most time reading documents. This function does not require a specific document or a user uuid and can be executed just by specifying the task number.

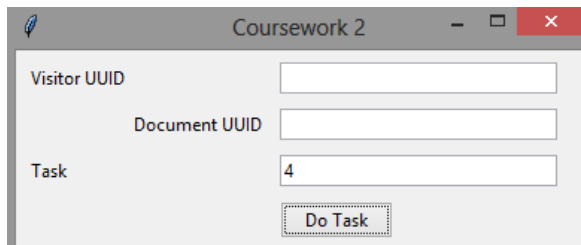


Illustration 14: Task selection

The result as expected is a list with the top 10 user_uuid

```
<'e529f034d3430af2', 5356278>  
<'dd326898d5605e63', 648318>  
<'458999cbf4307f34', 576177>  
<'849bb060cb110347', 486053>  
<'df70cddb46fd5da', 239350>  
<'14e1e343078d3d75', 161381>  
<'2105dd9bc68afb9d', 109458>  
<'da0df8a63107e139', 98291>  
<'b9caded38e707eca', 98183>  
<'e1178362fc11d6ba', 98172>
```

Illustration 15: Task 4 output

4.4 “Also likes” functionality – task 5a/5b/5d/5e

The also likes functionality provides the user with the capability of finding documents similar to his interests based on what common interests between him and other users. More specifically the user can specify a document uuid and then get a list of readers who have read that exact document.

Furthermore, given a user_uuid he can see what other documents that specific reader has read. Then he can get some recommendations which can be based on:

- User's credibility, that is the time that each user has spent reading documents
- The number of times a document has been read.

Readers of a specific document → task 5a

- The user has to identify a document uuid and he will get a list of user uuid that have read that document.

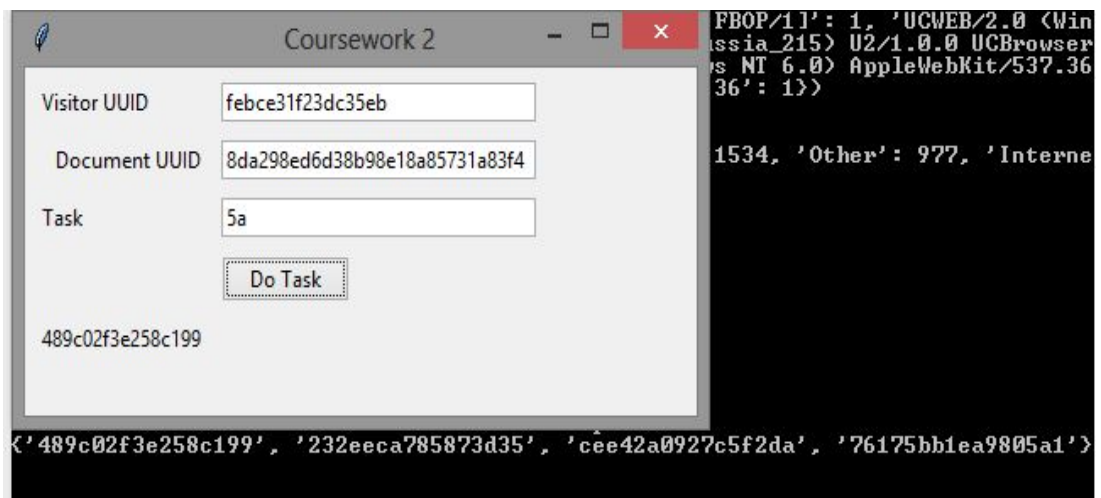


Illustration 16: Task 5a input and dictionary output

Documents of a specific user → task 5b

Given a user's uuid, the user can have a list of the document that has been read by that reader.

Example:

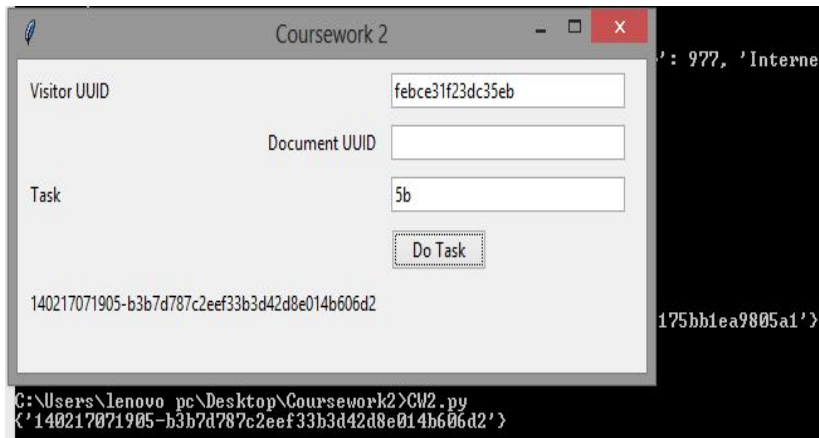


Illustration 17: input visitor uuid and output Document uuid

“Also like” documents based on readership profile – task 5d

This task allows the user to get a list of the top 10 documents that are also liked by other visitors.

This means that given a document 1, the result will be some other documents that have been read by readers who have also read document 1. The criteria for sorting these documents will be each user's reading profile. So for example if a reader has spent 200 minutes reading some documents and another has spent 100 minutes, then the documents of the first reader will come on top of the list.

Example:

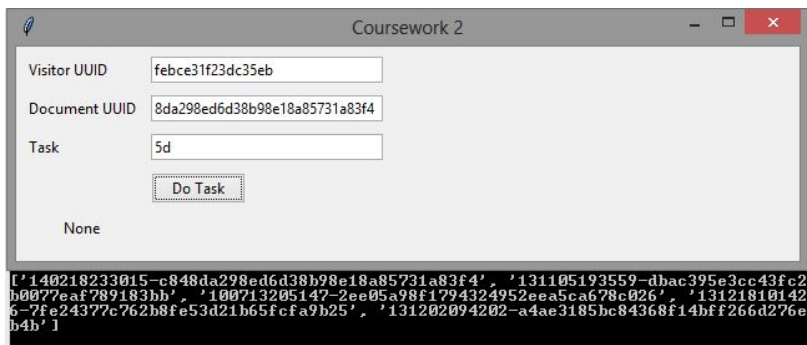


Illustration 18: Suggested documents based on reader profiles

“Also like” documents based readers count – Option 5e

This option returns again a top 10 list of suggested documents that have been read by readers that have originally read the specified document, but in this case the sorting of the top documents is taking place, taking into consideration the number of times that each document has been read. So for example if document 1 has been read by 20 readers and another document 2 has been read by 5, then document 1 will be on top of that returned list.

For example:

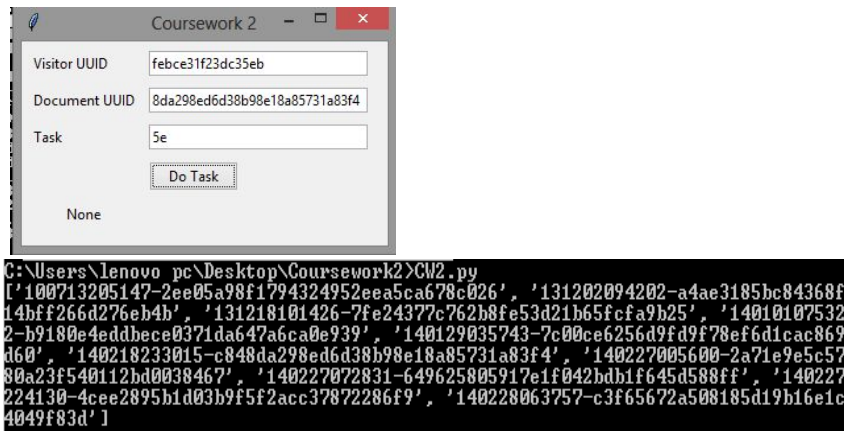


Illustration 18: Suggested sorted documents based on reader profiles

5. Developer Guide

The application is built in 1 python file. **CW2.py** that contains the functions for each of the required tasks along with the libraries and the GUI which receives user input and calls the functions to be executed.

During the whole implementation, specific coding standards tried to be maintained such as declaring all functions with an uppercase first letter, while variables were declared with a lower case one for accessibility.

Furthermore lists and dictionaries created had the name of the function that was operating along with the word Dict or List at the end.

5.1 Formatting the JSON file

The purpose of this application is to take data from a given JSON file, then analyze them and do the required tasks, for different user cases.

The JSON file was provided and was originally titled CW2.json and it is provided along with the application code.

5.2 Implementing Functions

All the program's functions are located inside **CW2.py** and are being called by the **Do Task button** in the GUI along with user specified arguments such as document uuid, reader uuid and task number.

The libraries that were used are: matplotlib, collections, tkinter, json and getopt.

5.2.1 Views by country/continent – *Task 2a(self, doc)*

For this and every other function iteration though the fixed JSON file is required and was required. After finding the document with the specified uuid that is being passed on by the user, which is the value of the element *visitor_country* is being passed on a list that will contain all the country names under the name *docCountryList*.

```
def task2a(self, doc):
    docCountryList=[]
    for x in self.records:
        if x.get('subject_doc_id') == doc:
            docCountryList.append(x['visitor_country'])
```

Code 1 task2a viewByCountry(self, doc)

Views by country

Taken the list that was created above, we make use of the function *Counter*, which is included inside the library *collections*. Counter function, counts the occurrences of each element inside a list and maps them to a dictionary. So the dictionary will have the form of:

```
C:\Users\lenovo_pc\Desktop\Coursework2\CW2.py
Counter(<<'MX': 1, 'SA': 1, 'UA': 1, 'BR': 1>>)
```

These values are being passed on to two lists that will be the x and y axis of the histogram that will be formed.

```
#Run histogram and return countries list
x = []
y = []
for k,v in Counter(docCountryList).items():
    x.append(k)
    y.append(v)
print (Counter(docCountryList))
plt.title('Countries of Viewers')
plt.bar(range(len(y)), y, align='center')
plt.xticks(range(len(y)), x, size='small')
plt.show()
return docCountryList
```

Code 2: Countries histogram and list creation

Views by continents

Using again the `docCountryList` that was created, along with the list of continents and countries that I created at the start of the script, we are mapping the countries inside the list to specific continents.

To do so we iterate each every country inside the list, and then find a match in `cntry_to_cont` dictionary which maps countries to continents and finally we map each continent with each full name (SA → South America).

Once the mapping is complete the continent is appended inside the *continentsList*.

```
continents = {
    'AF' : 'Africa',
    'AS' : 'Asia',
    'EU' : 'Europe',
    'NA' : 'North America',
    'SA' : 'South America',
    'OC' : 'Oceania',
    'AN' : 'Antarctica'
}
```

```
cntry_to_cont = {
    'AF' : 'AS',
    'AX' : 'EU',
    'AL' : 'EU',
    'DZ' : 'AF',

```

Code 3: Dictionaries for mapping countries and continents

```
continentsList = []
for entry in docCountryList:
    for country, continent in cntry_to_cont.items():
        if entry == country:
            for cntnt, cntntName in continents.items():
                if cntnt == continent:
                    continentsList.append(cntntName)

x = []
y = []
#Insert countries and number of occurrences in two separate lists
for k,v in Counter(continentsList).items():
    x.append(k)
    y.append(v)

print(Counter(continentsList))
plt.title('Continents of Viewers')
plt.bar(range(len(y)), y, align='center')
plt.xticks(range(len(y)), x, size='small')
plt.show()
return (Counter(continentsList))
```

Code 4: Inserting continent in the list

On a similar way like we did with countries and with the use of *Counter* function, we iterate through the list and create a dictionary of the continents and occurrences. After that, two lists are created that will include continents' names and continents' occurrences and will form the x and y axis for the histogram.

5.2.2 Views by browser – Task 3(self)

This function was implemented on a similar way as the previous one. Iteration through each object of the JSON file is required, and the value of the field *visitor_useragent* was extracted and added to the *userAgentList*. Then using the *Counter* function again, a dictionary was created and the two lists extracted from its keys and values were used as axis for the histograms.

```
#3a.Return browser names
def task3a(self):
    """Show a histogram of the browsers used by viewers"""
    userAgentList = []
    for entry in self.records:
        userAgentList.append(entry['visitor_useragent'])

    x = []
    y = []
    #Insert countries and number of occurrences in two separate lists
    for k,v in Counter(userAgentList).items():
        x.append(k)
        y.append(v)
    print(Counter(userAgentList))
    plt.cla()
    plt.title('Viewers User Agents')
    plt.bar(range(len(y)), y, align='center')
    plt.xticks(range(len(y)), x, size='small')
    plt.show()
    return userAgentList
```

Code 5: Adding browsers name to the list, counting them and making a histogram.

For the second part, where the browser names had to be displayed and not the whole browser identifiers, we were searching for keywords inside each *visitor_useragent* element.

These keywords were the names of the popular browsers (Firefox, Chrome etc.). If a match was found then the name of the browser was being added to the *browsersList*.

```

def task3b(self):
    """Show a histogram of the browsers used by viewers
    browser_count = {}
    browsersList = []
    for entry in self.records:
        if "Firefox" in entry['visitor_useragent']:
            browsersList.append("Firefox")
        elif "Chrome" in entry['visitor_useragent']:
            browsersList.append("Chrome")
        elif "Safari" in entry['visitor_useragent']:
            browsersList.append("Safari")
        elif "Opera" in entry['visitor_useragent']:
            browsersList.append("Opera")
        elif "MSIE" in entry['visitor_useragent']:
            browsersList.append("Internet Explorer")
        else:
            browsersList.append("Other")

```

Code 6: Adding names to the list

After the creation of the list, the histogram was created using the same logic as before.

```

#Run histogram and return browsersList
x = []
y = []
#Insert countries and number of occurrences in two separate lists
for k,v in Counter(browsersList).items():
    x.append(k)
    y.append(v)
print(Counter(browsersList))
plt.cla()
plt.bar(range(len(y)), y, align='center')
plt.xticks(range(len(y)), x, size='small')
plt.show()
return browsersList

```

Code 7: Browsers Histogram

5.2.3 Reader profiles – Task40

For this function, the top readers had to be calculated, based on their total reading times. To calculate the reading times what was done was to iterate through the json file and look at specific objects, where the value of the element *event_type* is *pagereadtime*.

```
#Look for pagereadtime occurrence
if(entry['event_type'] == 'pagereadtime'):
```

Code 8: Conditional check

Once that check has been made, the *visitors_uuid* and his *pagereadtime* are being added inside *readTimeDict*, which is a dictionary in the form of *readTimeDict[visitor_uuid] = pagereadtime*.

Since the total amount of readtime for each visitor has to be calculated and some users appear more than once inside the JSON file, there has to be iteration through the json file in order to check if a user has already been added and add the new reading time to the previous one. This is being accomplished in the following lines of code.

```
#Iterate through the json file
for entry in self.records:
    #Look for pagereadtime occurrence
    if(entry['event_type'] == 'pagereadtime'):
        if (entry['visitor_uuid'] in user_readTimes):
            user_readTimes[entry['visitor_uuid']] += entry['event_readtime']
        else:
            user_readTimes[entry['visitor_uuid']] = entry['event_readtime']
```

An extra check is being done to secure that the *['event_readtime']* element is not empty. Once the dictionary is done it is being returned, for sorting and printing.

The reason why the sorting is not being executed inside this function is because *user_readTimes* is being used by another function (5d) in later stages.

The operations of sorting and printing are being executed in another function which takes the dictionary returned from *AvidReaders()*, where the dictionary is being sorted based on the values, in reverse order and the top 10 readers are added inside *readTimes*.

```
readTimes = list(sorted(user_readTimes.items(), key=operator.itemgetter(1), reverse = True))[0:10]
for times in readTimes:
    print(times)
return readTimes
```

5.2.4 “Also likes” functionality – *AlsoLike(self, doc, task)*

This function has to return “also liked” documents for a given document uuid based on two different criteria.

The first one is the readership profile and the second is the documents' read count, meaning by how many different readers each document has been read.

Given a document uuid iteration through the file is being made to check all the readers that have also read this document, and the results are being inserted into *readersList*.

```
def DocToReaders(self, doc):
    readersList = []
    for entry in self.records:
        #Search for doc in JSON,if the element doc_id exists
        if entry.get('subject_doc_id') == doc:
            #Insert visitor's ID inside list
            readersList.append(entry['visitor_uuid'])
        #return distinct readers IDs
    return set(readersList)
```

Code 9: Readers List based on document uuid

On a similar way and given a reader's uuid, all the documents that have been read by that specific reader are being added into *docList*.

```
#5b. Returns DocumentIDs based on a readerID
def ReadersToDoc(self, user):
    docList = []
    for entry in self.records:
        #Find user ID
        if entry['visitor_uuid'] == user:
            #Check that doc ID exists
            if entry.get('subject_doc_id') != None:
                #Add document ID to the list
                docList.append(entry['subject_doc_id'])
    #return distinct values for document IDs
    return set(docList)
```

Code 10: Documents list based on reader uuid

After these two lists have been filled with readers and documents respectively, a dictionary is created, named *readersDocDict*.

This dictionary will map the readersIDs with DocumentIDS. So for each reader key there will be all the documents that he has read as values. Taking the form bellow:

Reader1	[Doc2,Doc3,Doc15,Doc6]
Reader2	[Doc1,Doc20,Doc15,Doc7,Doc60,Doc12]
Reader3	[Doc120]

Table 1: Readers to Documents Dictionary Example

```
def AlsoLike(self, doc, task):
    readersDocDict = dict()
    #Iterate over readers IDs
    for k in self.DocToReaders(doc):
        #Iterate over Document IDs that each reader has read
        for v in self.ReadersToDoc(k):
            #Insert readerID,docID into Dictionary
            readersDocDict.setdefault(k, []).append(v)
```

Code 11: Readers to Documents dictionary

“Also like” - based on readership profile - 5d

Once this dictionary is completed, we are making use of the *readTimes* that was created at 5.2.3 for reader profiles, in order to map Reading Times of the users to the documents that they have read.

Iteration through each of those dictionaries is performed and these two dictionaries are being merged into a new one which will include reading times of the users as keys and the documents that they have read as values. In order to make this procedure more clear to understand we are using the diagram bellow to depict these stages.

<u>ReaderID</u>	<u>ReadTime</u>
232eeca785873d35	1100
c08fc48b49f0e1be	520
0826ad759b5d254e	2500

Table 2: readTimes

<u>ReaderID</u>	<u>DocumentsID</u>
c08fc48b49f0e1be	[110322220408-aadc46d780e4a7605], [140206010823-b14c9d93a04234af7], [1107-000009cca70787e5fba1fda005c85]
232eeca785873d35	[140206010823-b14c9d966be95031], [000e976612072abfdd0e95]
0826ad759b5d254e	[130701025930-558b150c485fc89], [180e4eddbece0371da647a6ca0e],[3306- ba4f086f3bc24fdd93],[31120234743- 616166d17],[81ae8f2b9acdf0324d892]

Table 3: readersDocDict

ReadTime	Documents uuid
1100	[140206010823-b14c9d966be95031], [000e976612072abfdd0e95]
520	[110322220408-aadc46d780e4a7605], [140206010823-b14c9d93a04234af7],[1107- 000009cca70787e5fba1fda005c85]
2500	[130701025930-558b150c485fc89], [180e4eddbece0371da647a6ca0e],[3306- ba4f086f3bc24fdd93],[31120234743- 616166d17],[81ae8f2b9acdf0324d892]

Table 4: DocReadTimeDict

The new dictionary will contain Reading times mapped to Document uuid. Each user who is represented by his reading time can have multiple documents.

```
#Option 5d for returning a list of documents based on readership profile
if task == "5d":
    docReadTimeDict = dict()
    #Iterating through Doc -->
    for k,v in self.TopReaders().items():
        for key,value in readersDocDict.items():
            if k == key :
                #Iterate through values
                for i in value:
                    #Insert DocID -- > readTime into dictionary
                    docReadTimeDict[i] = v
```

Code 12: Creating docReadTimeDict

Once the dictionary is created we can then sort it based on reading times and then get the documents for each of the top readers and add them on a list.

The list used for this purpose is called *alsoLikeList* and it includes the top 10 documents based on reading times of the readers.

If the top reader has read 6 documents, then these 6 documents will be at the top of the list. The second reader's documents will be added etc.

```
alsoLikeList = []
for k,v in sorted(docReadTimeDict.items(), key = lambda x:-x[1])[:10]:
    #Insert top 10 values on list
    alsoLikeList.append(k)
print(alsoLikeList)
return (alsoLikeList)
```

Code 13: Also Like list → Based on readership profile

“Also like” - based on number of readers of the same document - 5e

This function has to return suggested documents, for a given document uuid which will be based on the number of readers that have read them. The implementation is quite simple and similar to the above.

Iteration through *readersDocDict* is being made, which includes reader uuid and all of the documents that each one has read, and inserts each document inside *docList*.

After that each element inside the list is being counted and finally the elements top 10 elements (documents) with the most occurrences are being returned.

```
#Option 5e for returning a list of documents based on number of readers
if task == '5e':
    docList = []
    for k,v in readersDocDict.items():
        #Store each document read, inside a list
        for i in v:
            docList.append(i)
    countDocDict = dict()
    #Create dictionary with DocumentsRead --> Number of occurrences
    for x in docList:
        if x in countDocDict:
            countDocDict[x] += 1
        else:
            countDocDict[x] = 1
    alsoLikeList = []
    #Insert top 10 documents, based on the times they have been read inside a list
    for k,v in sorted(countDocDict.items(), key = lambda x:-x[1])[:10]:
        (alsoLikeList.append(k))
    print(sorted(alsoLikeList))
    return sorted(alsoLikeList)
```

Code 14: Also like list → Based on amount of readers

5.2.5 GUI

For this project, I chose to go for a very simple looking interface so as to allow the user to understand what he has to do.

I used 3 labels, 3 textbox and one button and to have a proper fit and layout, I used a Frame to display those components.

The code above represents the frame I used to place all the components.

```
#-----
#GUI Class
#-----
class GUI:
    def __init__(self):
        """initialisation of the UI"""
        self.root = Tk()
        self.root.title("Coursework 2")

        mainframe = ttk.Frame(self.root, padding="3 3 12 12")
        mainframe.grid(column=0, row=0, sticky=(N, W, E, S))
        mainframe.columnconfigure(0, weight=1)
        mainframe.rowconfigure(0, weight=1)
```

Code 15: Frame design

After designing the frame, I then started creating textboxes and labels and placing them in appropriate locations.

The above code is the creation of labels and placing them on the frame.

```
ttk.Label(mainframe, text="Visitor UUID").grid(column=1, row=1, sticky=W)
ttk.Label(mainframe, text="Document UUID").grid(column=1, row=2, sticky=E)
ttk.Label(mainframe, text="Task").grid(column=1, row=3, sticky=W)
```

The above code is the creation of textboxes and placing them on the frame next to each label.

```
self.visitor_uuid = StringVar()
self.document_uuid = StringVar()
self.task = StringVar()

visitor_entry = ttk.Entry(mainframe, width=30, textvariable=self.visitor_uuid)
visitor_entry.grid(column=2, row=1, sticky=(W, E))

document_entry = ttk.Entry(mainframe, width=30, textvariable=self.document_uuid)
document_entry.grid(column=2, row=2, sticky=(W, E))

task_entry = ttk.Entry(mainframe, width=20, textvariable=self.task)
task_entry.grid(column=2, row=3, sticky=(W, E))
```

The above code is the creation of the “Do task” button and placing it at the bottom (row=4).

```
ttk.Button(mainframe, text="Do Task", command=self.doTask).grid(column=2, row=4, sticky=W)
```

5.3 Main

In order for the application to run from the command line and receive document uuid, reader's uuid and the task Id as arguments, *getopt* library is being used.

```
import getopt
```

Code 15: Imports and to get inputs

What follows -u will be the reader's uuid, -d will be document's uuid and -t will be task.

After the inputs have been parsed they are assigned to their respective variables and the program continues its execution. Right after that and depending on the user's selection the selected task is being used.

In every case the matching list is being returned, iterated and every element of it is being printed.

```
#-----
#Main Method
#-----
if __name__ == '__main__':
    try:
        opts, args = getopt.getopt(sys.argv[1:], "u:d:t:")
    except getopt.GetoptError as err:
        print(str(err))
        sys.exit(0)
    user = ""
    doc = ""
    task = ""
    for o, a in opts:
        if o == "-u":
            user = a
        elif o == "-d":
            doc = a
        elif o == "-t":
            task = a
        else:
            | assert False, "unhandled option"
    if ((user == "") and (doc == "") and (task == "")):
        gui = GUI()
    else:
        taskEx = MainClass()
        taskEx.executeTask(user, doc, task)
```

Code 16: Command line code

6. Testing

Several different cases were tested, to verify that the application does not crush and follows the expected behavior.

Case Tested	Expected Result	Actual Result
Executing with no arguments	Do nothing – then exit	The expected
Execute task 5b with no user ID	Print empty list	The expected
Execute task with no doc ID	Print empty list/Empty histogram	The expected
Corrupt JSON file	Error / No execution	The expected
Executing with wrong doc ID	Print empty list/Empty histogram	The expected
Executing with wrong user ID	Print empty list/Empty histogram	The expected

Table 5: Tests Conducted

In general the application does not crush or to be more precise there was none of the different scenarios tested, that crushed the application. Corrupting the JSON file will make the application ineffective, since it cannot get data from it. Several conditions could have been added to check whether a list or a histogram is empty and print a relevant message to the user, but were not crucial to the functionality of the application and were omitted.

Furthermore if a user gets an empty list or an empty histogram he should consult this documentation and table 5 specifically, to identify what he might have done wrong. Empty lists and/or histograms are most probable to appear due to faulty user input.

7. Conclusions

The application is based on Python 3.4 and as a 4th year student, who had no background in Python programming; this coursework has taught me more about this language. As stated in the introduction, this coursework offered a great introduction to the Python language, getting to know some of its tools and libraries. Designing and developing an application in a new programming language, in a short time period had been a very challenging experience but gave me a chance to test my limits, improve my learning process and learn how to be effective under pressure.

GUI design implementation was used to enhance the user interaction. Furthermore several scenarios such as wrong document/reader Ids, which result in empty lists and histograms, was well dealt with according to conditional checks and the respective error messages, so a user can instantly know what is wrong.

To conclude with, this project offered a very intense approach to a new programming language, that required fast learning and implementation of what was learned offering overall a great educational experience.