DocuTrace

F20FC: Industrial Programming
Coursework 2

Sam Fay-Hunt — `sf52@hw.ac.uk`

December 7, 2020

Contents

1 Introduction

The DocuTrace application is a moderate size, data-intensive application, its purpose is to analyse and display document tracking data from the website issuu.com. The website hosts a substantial number of documents, and provides anonymised usage statistics, the data is provided in the form of a sequence of individual JSON entries seperated by new lines.

We make three assumptions based on the information from the dataset specification [1], first that a read is triggered by a 'read' event type, 'pageread' is omitted because it seems there are instances of 'pageread' events that dont correspond to 'read' events, secondly, we assume that any event on a document constitutes a 'view' thus we gather a broader category of metrics when counting views by country/continent/browser than when computing reader profiles or 'Also likes' data, and finally we assume that 'readtime' is the exclusive event indicator for an event containing read time data when building reader profiles.

This application aims to satisfy the task of processing this data according to these parameters while considering optimisations to speed up processing potentially very large files, while also providing a solid interface to visualise the data. It was also a priority to provide easily extendable code practices so that custom data collection strategies can be easily implemented.

It is assumed the users of an application like DocuTrace would be someone with a high enough degree of technical competency to use simple Linux command line applications, the user would likely be a researcher (data science), or a business. The prior assumption leads to the assumption that the hardware running this application would be closer to server class than standard consumer hardware with higher CPU core counts and alot more RAM. To try and compensate for the potential scale of the data a signifantly large amount of RAM is not mandatory to run this application, but a pool of aproximately 8GB of RAM should be installed on the system at a minimum when processing 3 million lines otherwise a performance penalty may be incurred.

DocuTrace was written in Python 3, it is intended to be run on Ubuntu 20.04, and has not been tested on other operating systems.

Comprehensive documentation has been generated, a list of dependencies, installation and run instructions are provided, note the recommended entrypoint of the application is via the "docutrace" shell script, this script will automatically configure an environment variable to specify the output directory of graph files generated.

1.1 Libraries

- numpy - Used for convienience and access to fast C vector operations.
- matplotlib - This library was used to display charts.
- user-agents - Helped translate the 'user-agent' JSON entry into a more readable format.
- pycountry - Converts countries alpha2 code into a full country name.
- pycountry-convert - Get the name of a continent for a given country.
- graphviz - Used to plot the "Also likes" graph.
- regex - This library is prefered to the standard "re" library because it has a `fullmatch()` method, and allows us to easily compile and reuse a regex expression.
- alive-progress - Provides beautiful, easy to implement progress bars when the total load time is unknown.
- python-decouple - Convieniently reads environment variables and .env files with no boilerplate code.

2 Requirements Checklist

The priority for each requirement is encoded as follows:

- **Essential** - This priority indicates that this requirement must be implemented to satisfy basic functionality of the application, these requirements are all explicitly requested in the task brief.
- High - A high priority indicates that this requirement is important for providing a good quality application.
- Medium - A medium priority requirement is nice to have but if it is missing it is acceptable.
- Low - Low priority indicates that this is unlikely to be fulfilled, or it is generally unimportant for the applications functionality as a whole.

Requirement	**Description**	**Priority**	**Status**
Runs on Linux	The application needs to run on an up-to-date Linux platform (Ubuntu 20.04)	**Essential**	Complete
Use Python 3	The core logic of the application should be implemented in Python 3	**Essential**	Complete
Views by country	Given a document UUID find from which countries the document has been viewed, display as histogram	**Essential**	Complete
Views by continent	Given a document UUID find from which continent the document has been viewed, display as histogram	**Essential**	Complete
Views by browser	For an input file count the number of occurrences for each browser used to access each document in the file, display full and truncated strings as histogram	**Essential**	Complete
Reader profiles	For each user find the total time spent reading documents, display the top 10	**Essential**	Complete
Readers of document	For a given document identify all visitor UUIDs of readers of that document	**Essential**	Complete
Documents from reader	For a given reader identify all documents that reader has read	**Essential**	Complete
"Also likes" functionality	Using the readers of document and document from reader functionality return a list of documents sorted by a given sorting function as a parameter	**Essential**	Complete
"Also likes" graph	generate a graph that visualises the "Also likes" functionality	**Essential**	Complete
Simple GUI	Develop a simple GUI based on tkinter to display the statistical data and recieve input parameters	**Essential**	Complete
Command-line usage	Provide a command line usage to test the application functionality in an automated way	**Essential**	Complete
Display query history in GUI	Extend the gui to display a history of queries that allow navigation to past queies	Medium	Incomplete
Provide documentation	Generate and host documentation for the application	High	Complete
Log exceptions and warnings	Log any exceptions encountered to be handled by journald	Low	Complete

3 Design Considerations

Extensive documentation has been provided for other developers wishing to extend this codebase, the documenation has been styled useing the ”readthedocs” theme [2]. Additionally the `PEP8` sytle guide [3] has been utilised to aid readability and help ensure consistent coding style.

As well as the required parameters specified in the CLI task specification some secondary parameters have been included, this includes log level verbosity, an argument to limit the volume of data displayed in the GUI and command line, and a parameter to exit the application early (used when only a single task is needed).

The potentially long loading times when procesing large datafiles motivated the inclusion of an animated loading bar to provide some feedback to the user.

A logger is used to handle debug, info, warning and exception messages. The logger sends messages to stdout so they can be managed by journald, this includes exceptions and traceback messages this is a common practice and lets system admins handle the log output as they see fit. The verbosity of the log message can be set with the -v parameter, by default all messages of logging level WARNING and above are displayed.

4 User Guide

This application is started from the command line, all arguments are optional, but supplying at least a filepath and task id as parameters will speed up operation. For help a `-h` flag can be passed and the program will display all the parameters it can take with a description, see Fig. 1 for a listing of CLI options.

```
docutrace [-h] [-u USER_UUID] [-d DOC_UUID] [-t TASK_ID] [-f FILEPATH] [-n
↪  [LIMIT_DATA]] [-v [VERBOSE]] [-e [EXIT_EARLY]]
```

Figure 1: CLI options

5 Developer Guide

Describe your application design and main areas of code in order to help another developer understand your work and how they might develop it. You may find it useful to supplementthe text with code fragments.

The provided documentation includes a definition of expected parameter types for all functions and methods. Additionally primative type stubs have been added to the function and method signatures to improve readability. More comprehensive type information could be added in the future by definig custom type stubs for all the classes, and union types.

5.1 Summary of application flow

This application can be broken down into 4 parts: Analysis, Gui, Utils packages and the main.py script; main.py is the primary entrypoint of the application, Analysis module handles all the data processing needs of the application, Util provides convienience functions like logging, exceptions, validation and also a task selection script, and finally the Gui module handles all functionality related to the graphical user interface.

The main entrypoint (DocuTrace/main.py) is a script to define the arguments, checks the validity of the filepath (a thread is started to begin processing the data right away) and task identifier from the command line, finally it starts the logic that parses the task and runs the given task once the data has finished processing.

Once processing is finished the task is launched, this will open the GUI. An instance of the ComputeData class is instantiated using the data obtained by the DataCollector class, it is then passed to the gui, this class handles all iteractions between the gui and the data.

Once the gui opens the selected task should be visible on the screen, with the data visible, at this point new parameters for the various tasks can be provided to view other datapoints related to this task, or the number of instances displayed can be modified, depending on the task.

5.2 Analysis package

5.2.1 FileRead

The FileRead module handles all interaction with the filesystem with reguards to opening and closing the file. The core function used to read from the file is the `stream_file_chunks()` function, this function creates an iterator and laziliy reads the file on demand. There are 3 classes within this module, ParseFile, JsonProcessContextManager, and JsonParseProcess. A file is actually read by the `parse_file()` method of the ParseFile class, when concurrency is enabled it uses the JsonProcessContextManager to add chunks of the JSON data file to a `JoinableQueue` [4] managed by the context manager, see Fig. 2. The context manager starts all the processes, and maintains 2 queues: the aformentioned `JoinableQueue` that recieves JSON chunks, and a second `JoinableQueue` to recieve processed data back from each process named the `feedback_queue`.

```python
        with JsonProcessContextManager(data_collector, max_workers) as jtcm:
            for chunk in self.file_iter:
                jtcm.enqueue(chunk)
            logger.debug('Finished queueing chunks')
```

Figure 2: Context manager to start Processes and enqueue chunks from the JSON file, inside the ParseFile class.

The JsonParseProcess subclasses Pythons `Process` [4] class to get around the global interpreter lock, it takes chunks from the JSON queue, processes the chunk and then adds a `DataCollector` class to the feedback queue until all items have been processed. There is substantial room for optimisaion in this class, deepcopy is utilised liberally as a workaround to prevent race conditions, however this adds substantal overhead to the processing time (almost double) for collection of data.

When the context manager main thread is finished enqueueing JSON chunks it starts a thread to deque and merge the feedback queue back into a single instance of the `DataCollector` class.

5.2.2 DataCollector

The DataCollector module features 3 convienience classes (ReadingData, BrowserData, and DocLocation) to help with internal data representation in the DataCollector class. The convienience classes all overload the `__add__()` method which is used when the merge_dict() function is used, additionally ReadingData and BrowserData overload `__eq__()` and `__lt__()` methods which when combined with the `@total_ordering` decorator from functools [5] allows all comparison operations to be performed.

The constructor of the DataCollector class itself has parameters to disable collection of a specific metric if desired, this can be useful to speed up testing, the path paramter is important; it is passed to an instace of ParseFile (see Section 5.2.1). During instantiation the constructor builds a list of methods that will be used by the ParseFile class to process the JSON data once the `gather_data()` method is called. This module also contains 2 decorators; `@CheckEventRead`, and `@CheckEventReadtime` that can be modified, or extended to easy apply checks for certain properties in the json dict, these can be handy to reduce code repetition.

5.2.3 ComputeData

The ComputeData module contains several helper functions indluding some sorting functions and funcstions to get country and continent names from an Alpha2 country code.

The `ComputeData` class is instantiated using an instance of the `DataCollector` class, it has a number of methods to sort and print the processed data, it also has a `histogram()` method, which relies on one of the construct figure methods having been run, this histogram method can be used to display any plot produced by the `Charts` class within the Plots module.

5.2.4 Plots

The Plots module supplies 2 classes, `Graphs` and `Charts`.

The `Charts` constructor allows you to sprecify the size, and dimension of the plot, this class also has a method to produce a matplotlib `ax` object from a dictionary; the `ax_bar_from_dict()` method does this. This implementation means we just need to supply a dictionary of data (no dictionary nesting is supported) and the labels of each axis and titles to plot a histogram convieniently.

The `Graphs` class is used exclusively to plot the "Also likes" digraph, its constructor takes an instance of the `ComputeData` (see Section 5.2.3) class and a file name used to save the graph.

The full logic to plot the "Also likes" graph uses a number of methods from the `ComputeData` class, including the `find_relevant_docs()` and `also_likes()` methods, to prevent circular import the `top_n_sorted()` function is imported inside this method. Once the `Digraph` [6] is instantiated, we first crate a dictionary of readers as keys and lists of douments as values, see figure 3.

```
        self.graph = Digraph(name='Also likes', filename='Also likes', format='png')
        self.graph.attr('graph', ranksep='0.75')
        node_dict = {k: self.compute_data.visitor_documents.get(k, []) for k in
        ↪   reader_list}
```

Figure 3: Instantiate Digraph and build dictionary of reader -¿ document connections

Next the numpy library is used to remove duplicated documents and the `get_edges()` function is used to find all edges encoded in `node_dict`, followed by some cleanup logic to remove unwanted edges. Now that the data is prepared context managers are used to build 3 subgraphs, one to act as a graph legend, showing the readers and documents, `readers` places all the reader nodes in a horizontal row, and the `documents` context manager places all the document UUIDs horizonally below the readers. Finally we loop over all the edges found earlier and add them each to the graph.

When we wish to display the graph in the Gui we use the `save_view_graph()` method to render and save the graph on the filesystem to be loaded and used as required.

5.3 Gui

5.3.1 main

The main module for the GUI, this module handles window cleanup on close, and configures an on tab changed event, it also programatically instantiates all the tabs to display and keeps a dictionary reference of them, see Fig. 4. Its primary purpose is to act as the root node of the tkinter gui, all other gui elements are encapsulated by this element.

5.3.2 Tab

The Tab module handles the majority of the gui logic. The `Tab` class handles all the logic of constructing a tab in the gui. Upon instantiation it creates 3 fields that are used to store user input, see Fig. 5, the reference to these is passed as a parameter to the `Controls` class. The `widget_fn` parameter is important, it dictates how the tab will render the `Controls` and `Content` frames, this

```
        self.tab_ref = {}
        for k, v in tab_dict.items():
            self.tab_ref[k] = Tab(compute_data, v, master=self.tabs, doc=doc_uuid,
            ↪  user=user_uuid, n=n)
            self.tabs.add(self.tab_ref[k], text=k)
        if start_tab:
            self.tabs.select(self.tab_ref[start_tab])
```

Figure 4: GuiRoot class instantiation of Tab class, using a dictionary of task building functions and selection of initial tab.

function is called at the end of the constructor. A dictionary is used to associate a task identifying string with a function, this function is passed into the constructor when a Tab is instatiated in the main module, see Fig. 4.

The `Tab` class also manages interaction with the `ComputeData` class, by calling methods based on the tab context and passes the necessary parameters.

```
        def __init__(self, compute_data, widget_fn=pass_fn, master=None, doc:
        ↪   str=None, user: str=None, n: int=None):
            super().__init__(master)
            self.master = master

            self.doc_uuid = tk.StringVar(self.master, value=doc)
            self.user_uuid = tk.StringVar(self.master, value=user)
            self.n = tk.IntVar(self.master, value=n)

            self.controls = Controls(self, self.doc_uuid, self.user_uuid, self.n)
            self.controls.grid(row=0, rowspan=2, columnspan=10, padx=10, pady=5,
            ↪  sticky=(tk.N, tk.W))
            self.content = Content(self, self.doc_uuid, self.user_uuid, self.n)
            self.content.grid(row=1, rowspan=10, columnspan=10, pady=90,
            ↪  sticky=(tk.N, tk.E, tk.S, tk.W))

            self.compute_data = compute_data
            self.widget_fn = widget_fn
```

Figure 5: Part of the Tab constuctor, StringVars and IntVar used to manage the user input, instantiation of Controls and Content classes

The `Controls` class extends the frame class and is used to isolate the control section of the GUI from the content section, this class supplies methods to draw the various input boxes on the window, none of these are fixed because the same methods are reused to draw every tab. A similar pattern has been used for the `Content` class, it has 3 methods to handle displaying various types of content: matplotlib figures, graphviz graphs, and text information.

5.3.3 Tasks

The functions in this module are the definitions of the functions used in the `task_dict` from the Tab module.

5.4 Utils

5.4.1 Exceptions

This module defines all the custom exceptions used in the application.

5.4.2 Logging

The Logging module supplies the basic config for the logger used throughout this application. It also supplies a convienience wrapper function to set the log level to DEBUG.

5.4.3 Tasks

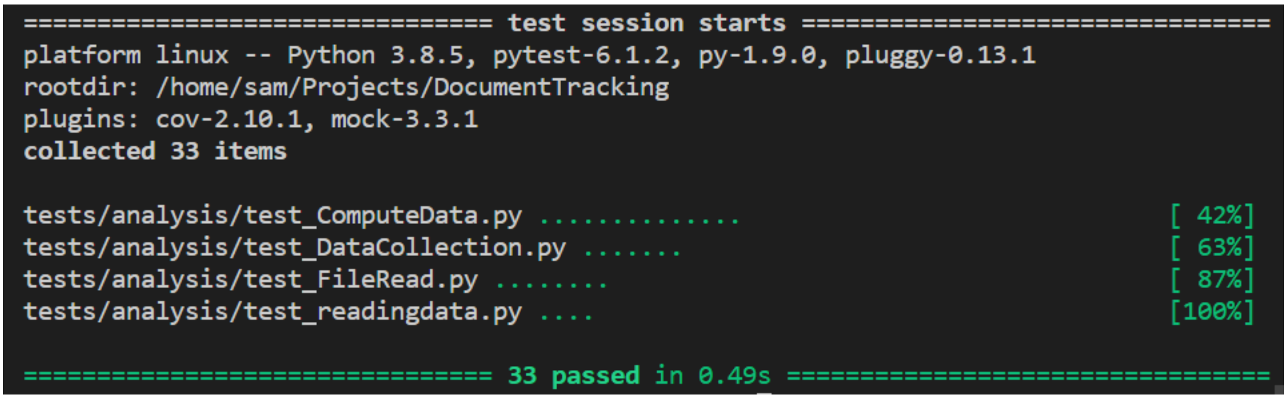
The Tasks module is the link between the Gui and the CLI, with functions to handle task selection, application exit conditions, and acquireing CLI arguments. It uses a similar pattern as the Gui for task selection, however this time it uses and `OrderedDict` class to enable functionality to move to the next task in order without restarting the program, when the `-e` parameter is not supplied.

5.4.4 Validation

The Validation module has functions to handle input validation and path checking. Credit for the `is_pathname_valid()` function goes to Cecil Curry from this entertaining stack overflow post[7].

6 Testing

Unit tests are included in the tests directory, this application was initially written in a test driven development manner, however only the core Analysis package code developed in this style due to time constraints. Pytest was used as the unit testing library, mainly because it requires less builerplate code than the Python built in Unittest library.



```
================================ test session starts =================================
platform linux -- Python 3.8.5, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /home/sam/Projects/DocumentTracking
plugins: cov-2.10.1, mock-3.3.1
collected 33 items

tests/analysis/test_ComputeData.py ..............                              [ 42%]
tests/analysis/test_DataCollection.py .......                                  [ 63%]
tests/analysis/test_FileRead.py ........                                       [ 87%]
tests/analysis/test_readingdata.py ....                                        [100%]

================================ 33 passed in 0.49s =================================
```

Figure 6: Tests running and passing

The unit tests proved invaluable when working on the FileRead module with concurrency. There are some issues with input validation handling logic, many possible CLI parameters have not been extensively tested so it may be possible to open the GUI without the requisite parameters.

Sometimes Numpy raises the "FutureWarning: elementwise comparison failed" warning, this is raised from line 157, 158 of the ComputeData module, the bug is inconsistent and not always easy to replicate, so fixing it has been challenging, it appears to be caused by using numpy boolean indexing to filter unwanted data from the `relevant_docs` numpy array.

7 Personal Development

A short discussion on lessons learnt from the feedback given on CW1and a discussion how you integrated this feedback into CW2. Cover both coding and report writing,possibly more (project manage-

ment, preparing for interview style questions etc). Lessons learnt from the experience of CW1: Started out using test driven development

7.1 Code feedback

- Feedback: limited input validation - Considerably more input validation takes place, use of regex expressions and loops to allow users to re-enter data helps, however a more systematic approach would be better.
- Feedback: Too much global state, not restrictive access modifiers - This has laregely been addressed, but this is due to the language shift, python doesnt use public/private access modifiers.
- Feedback: No custom exceptions - Custom exceptions have been used for input validation, more custom exceptions could have been used but the descision was made to handle most exceptions right away using logic based around the `None` type (for the most part these are `KeyErrors`).
- Feedback: No checks for arguments - Pythons duck typing system can make resolving this more complicated, explicit primative type stubs (although not enforeced) should provide hints to other developers when debugging, also in some cases the `hasattr()` function was used to try and test the correct argument type was provided.
- Feedback: Too much code duplication - Some effort was made to reduce code duplication in the Gui by using methods to build repeated gui elements, there still exists some code duplication particularly in the DataCollector class. This could be improved by creating a better class structure.

7.2 Report feedback

- Introduction
  - Feedback: Should briefly cover the spec - The introduction features a brief description of the spec.
  - Feedback: Should cover the goals -
  - Feedback: Discuss environment - Detail about the environment has been included in the introduction, from expected operating system to user technical competency and expected kind of hardware.
- Developer guide
  - Feedback: Discuss class dependencies - Class interdepenencies have been mentioned, additionally provided documenation supplies extra detail on this topic.
  - Feedback: Discuss method interfaces - Some detail here, could include alot more but it feels somewhat redundant with the provided documentation and primative type stubs.
- Conclusion
  - Feeback: Should discuss advanced language features -

8 Conclusions

Reflect on what you are most proud of in the application and what you'd have likedto have done differently. You should reflect on the produced software, and compare software devel-opment in scripting vs. systems languages.

Most proud of: - Concurrency during file reading - Structure of the code, good decoupling front and back end, and also interclass dependencies

Do differently: - Better time management, too much time spent bug fixing custom concurrency implementation, libraries exist that could handle this in a faster more resource efficient manner such as the fileinput library [8] - Leave more time to work on the gui - subclass the datacollector class to break it into smaller tasks - Should really provide an interface for an abstract datacollector, additionally make a base datacollector class to make extending the application with custom data collection methods easier.

Adv lang features: Decorators, lazy iterators, Processes, higher order functions, dynamic (duck) typing

Comparison between scripting and sys languages: Python as a hybrid language has scripting, oop and functional features, this flexibility is wonderful for rapid development, however it is a double edged sword

A References

References

[1] (). “Issuu anonymous dataset,” [Online]. Available: http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/dataset_spec.html (visited on 12/07/2020).

[2] (). “Home — Read the Docs,” [Online]. Available: https://readthedocs.org/ (visited on 12/06/2020).

[3] (). “PEP 8 – Style Guide for Python Code,” Python.org, [Online]. Available: https://www.python.org/dev/peps/pep-0008/ (visited on 12/07/2020).

[4] (). “Multiprocessing — Process-based parallelism — Python 3.8.6 documentation,” [Online]. Available: https://docs.python.org/3.8/library/multiprocessing.html?highlight=process#module-multiprocessing (visited on 12/05/2020).

[5] (). “Functools — Higher-order functions and operations on callable objects — Python 3.9.1rc1 documentation,” [Online]. Available: https://docs.python.org/3/library/functools.html (visited on 12/05/2020).

[6] (). “API Reference — graphviz 0.15 documentation,” [Online]. Available: https://graphviz.readthedocs.io/en/stable/api.html#graphviz.Digraph (visited on 12/06/2020).

[7] (). “Filesystems - Check whether a path is valid in Python without creating a file at the path’s target,” Stack Overflow, [Online]. Available: https://stackoverflow.com/questions/9532499/check-whether-a-path-is-valid-in-python-without-creating-a-file-at-the-paths-ta (visited on 12/07/2020).

[8] (). “Fileinput — Iterate over lines from multiple input streams — Python 3.8.6 documentation,” [Online]. Available: https://docs.python.org/3.8/library/fileinput.html (visited on 12/07/2020).