

Assignment Report

1 LITERATURE REVIEW

A brief description of the problem area and how your problem (the shipping records) fits in. A less brief description of the technical issues involved (with sources).

The first recognition which needs to be made is the type of data that I am working with. "An electronic revolution has already occurred over the last decade in access to records, but the records accessed either remained non-electronic in themselves or were surrogates of non-electronic originals" [1]. From this I deduce that I am working with data (Excel files) which are surrogates of non-electronic originals. That is to say, the Excel files were generated, either automatically or manually (this I don't know), based on the physical port or ship records. This is important to recognise because it allows assumptions to be made about the tidiness and consistency of the data. Physical records which were hand-recorded are likely to have inconsistencies, such as the format in which humans input the different fields, hand-writing legibility, no checks for making sure certain fields are populated, and so on. Records which were 'born-digital' [1] are more likely to have checks which ensure consistency is maintained, including formatting and whether certain fields are required.

Mandemakers and Dillon [1] provide a framework for working with historical data:

1. "To ensure the historical community can trust the results of research based on [the] data".
 - a. This means that the data can be trusted to be accurate without inaccuracies which would skew any attempt to form analyses and conclusions based on the data.
 - b. I have achieved this by ensuring that I don't perform any functions on the data when I import it into the database. If I performed data-tidying functions or data-altering functions at this stage, it would not be guaranteed that the data in the database explicitly and accurately represents the physical or digital historical records (in this case the Excel files). However, I did not have any control over the making of the Excel files, so I cannot be sure that the Excel files are a true match to the physical records.
2. "To benefit from previous experience in creating large databases. Thus, these rules can also be applied to data created by non-academic organizations"
 - a. The database I created (from the digital Excel records) followed a framework (the Python code) which was applied to the data from the presumably non-academic ship/port logs.
3. "To ensure that databases are of sufficient quality for use by secondary researchers (those outside the main database-creation team)"
 - a. Thanks to the implementation of a GUI, it's very easy for any users to interact with the database by navigating the Graphical Interface. Those who were not familiar with the data would still understand what they are being presented with. This does however mean that the user is limited to only the queries and outputs which the GUI is presenting them with, but the foundation is there for more queries and outputs to be performed and presented to the user. The database is also constructed in a way which would make sense to secondary researchers, because it is appropriately labelled and

follows a logical structure (e.g. 'vessel name', 'official number'). The fields are not ambiguous.

2 METHODOLOGY & RESULTS

A technical description of your solution with an explanation of the choices involved.

2.1 IMPORTING THE DATA INTO THE DATABASE

2.1.1 Original structure



Figure 1

Using the original python scripts which were provided by the tutor, the documents in the database had a structure which looked like Figure 1. This was problematic because this meant there was a one to many relationship with ship -> mariners. All of the mariners were held in the field 'mariners' which was an array object. Querying an array would have been much more complex than simply querying each mariner individually, particularly for the first task which concerns individual (unique) mariners.

2.1.2 Modifications

The modified python files for importing the data can be found in "get_records.py", "ship_excel_schema.py", and "transfer_all_ship_data.py".

- Creating a unique key (very difficult process which required a lot of debugging)
 - Had issues with the unique key not being consistent across the python files, so each time the function was called (when the next file was selected), the unique key was being reset

- The solution was to return the unique key from the get_records function, and then assign that in the transfer_all_ship_data where the function is passed

```
# Initialising a unique key
u_key = 1

# Walking through each file in the directory
for root, dirs, files in os.walk(ships_dir):
    for file in files:
        name, ext = os.path.splitext(file)
        if ext == '.xlsx':
            # Adding the data in the csv→json file to the all_ships list
            the_records, u_key = get_records.get_records(os.path.join(root, file), u_key)
            all_records += the_records
```

Figure 2

2.1.3 Testing the modifications

Before I imported the data in with the modified importing python files, I wanted to check that my test data had imported correctly. I first checked the modified structure as below for a visual, face-value check (Figure 3).

```

{
  "_id": 1,
  "vessel name": "Adroit",
  "official number": 8857,
  "port of registry": "Aberystwyth",
  "name": "John Williams",
  "age": 35,
  "place_of_birth": "Cardigan",
  "last_ship_name": "Majestic of Aberystwyth",
  "last_ship_port": "Cardiff",
  "last_ship_leaving_date": {},
  "this_ship_joining_date": "1850-09-21",
  "this_ship_joining_port": "Llanelly",
  "this_ship_capacity": "Mate",
  "this_ship_leaving_cause": "Remains on board",
  "signed_with_mark": "N",
  "additional_notes": "Remains on board"
}

{
  "_id": 2,
  "vessel name": "Adroit",
  "official number": 8857,
  "port of registry": "Aberystwyth",
  "name": "Edward Jones",
  "year_of_birth": 1855,
  "age": "No info",
  "place_of_birth": "Liverpool",
  "home_address": "No info",
  "last_ship_name": "Naval Reserve",
  "last_ship_port": "Liverpool",
  "last_ship_leaving_date": {},
  "this_ship_joining_date": "1850-09",
  "this_ship_joining_port": "Liverpool",
  "this_ship_capacity": "$",
  "this_ship_leaving_date": "1879-03-01",
  "this_ship_leaving_port": "[Hamburg?]",
  "this_ship_leaving_cause": "Discharged",
  "signed_with_mark": "Y",
  "additional_notes": "Discharged - in hospital"
}

```

Figure 3

I realised I could also check by counting the number of mariner records (aka the number of rows in each excel worksheet and workbook, starting from the first mariner row), and compare that to the number of documents in my collection. That is because each record = a new document in the collection, and therefore these numbers should match. **Check_record_count.py** is the code for how I achieved this, which prints out the results of the two counts (records and documents). When I ran this file I saw the pleasing result that the numbers did in fact match (Figure 4):

```

The number of excel mariner records in all worksheets and workbooks is: 5436
The number of documents in collection: 5436

```

Figure 4

Therefore, I felt confident to import all of the data into my database with this new structure.

2.2 INDIVIDUAL STORIES

The first task of the deliverables concerned 'Individual Stories'. The code for completing this task can be found in '[individual_stories.py](#)'. Code is commented as and when any new or particularly complex functions are introduced. Where functions or code is repeated or used in a similar way, I have not commented, as the intended function has already been explained previously. Below I will detail the requirement from the brief, along with the relevant functions and a brief explanation of what these achieve. The actual explanation of each line of code will be commented (though later on in the assignment my comments did become sparser).

2.2.1 "You will need to be able to identify individuals and distinguish them from other similar individuals"

The first challenge was to define logic by which mariners could be considered 'unique'. There is no single unique attribute the mariner's have, therefore a composite unique key would need to be decided. Within the "unique_selected()" function on line 94, you can see I have chosen: name, age, DOB, and place of birth as the group by which a mariner will be deemed as unique. It's not the most ideal, not least because the data is inconsistently populated (i.e., some mariners do not have a DOB), but it allows for a logic to be applied that it's unlikely that there will be many mariners who share all four of those characteristics.

2.2.2 "Your code should expose two Python functions: one to list sailors in order to facilitate selection..."

"group_by_name()" on line 35 is the first function which is used for this purpose. It groups mariners by name, and adds this name to a list, so this list can be used as a selection list for the user. This prevents the user from being bombarded with a GUI which has hundreds of thousands of the same name.

This list is used to populate a graphical list using the Tkinter library (Figure 5). The creation of the list can be seen from line 322 onwards within the view_mariners() function.

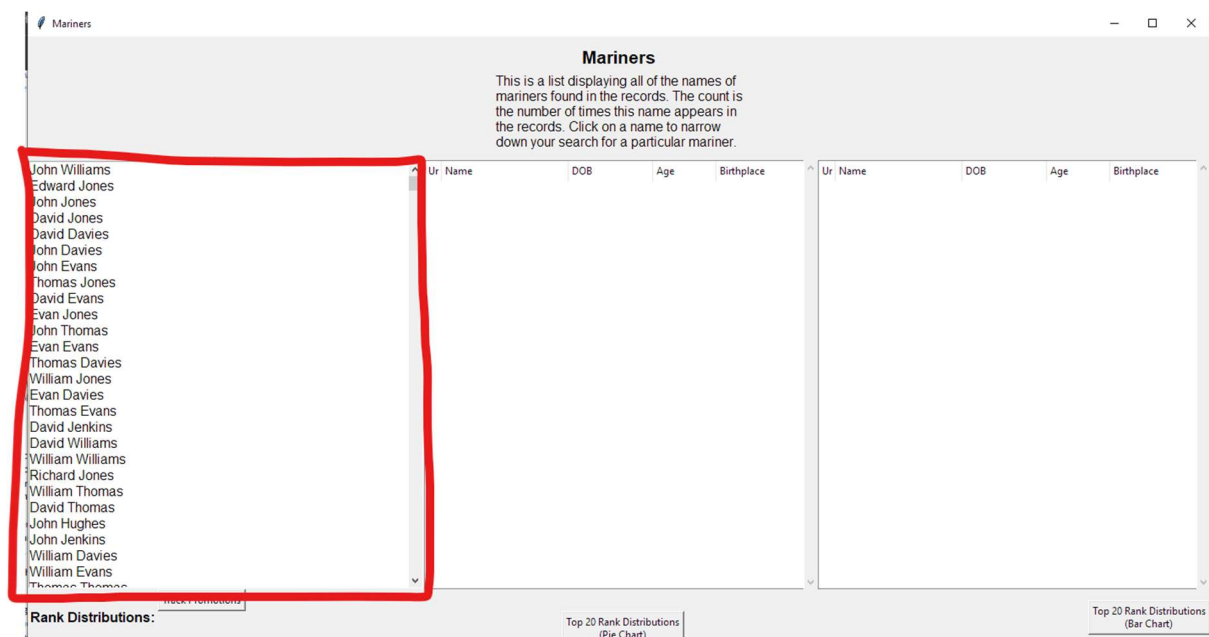


Figure 5

Getting the name the user has selected can be seen in “list_select()” on line 79. This allows a name to be selected, which then passes a query into the database which matches all records with that name and populates another list (or technically a Tree View widget) with all *unique* mariners matching that name (as defined by the unique mariner logic above), as well as those mariners attributes which make the mariner unique (Figure 6). The user can select up to two names.

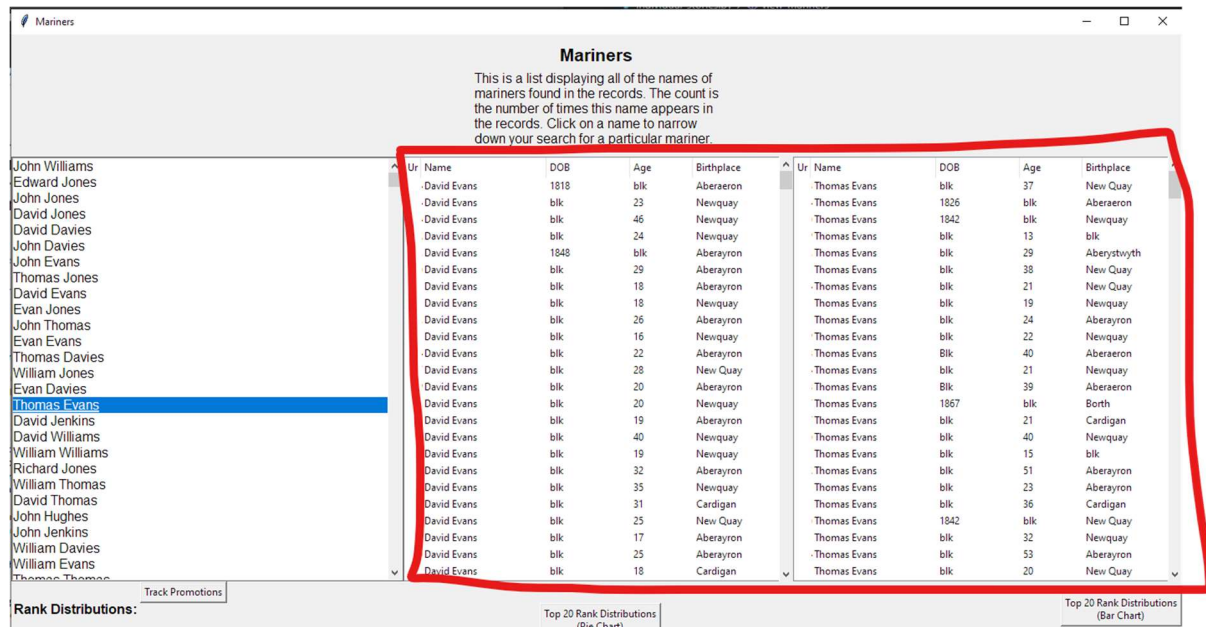


Figure 6

2.2.3 “...and one to get records on the selected sailor.”

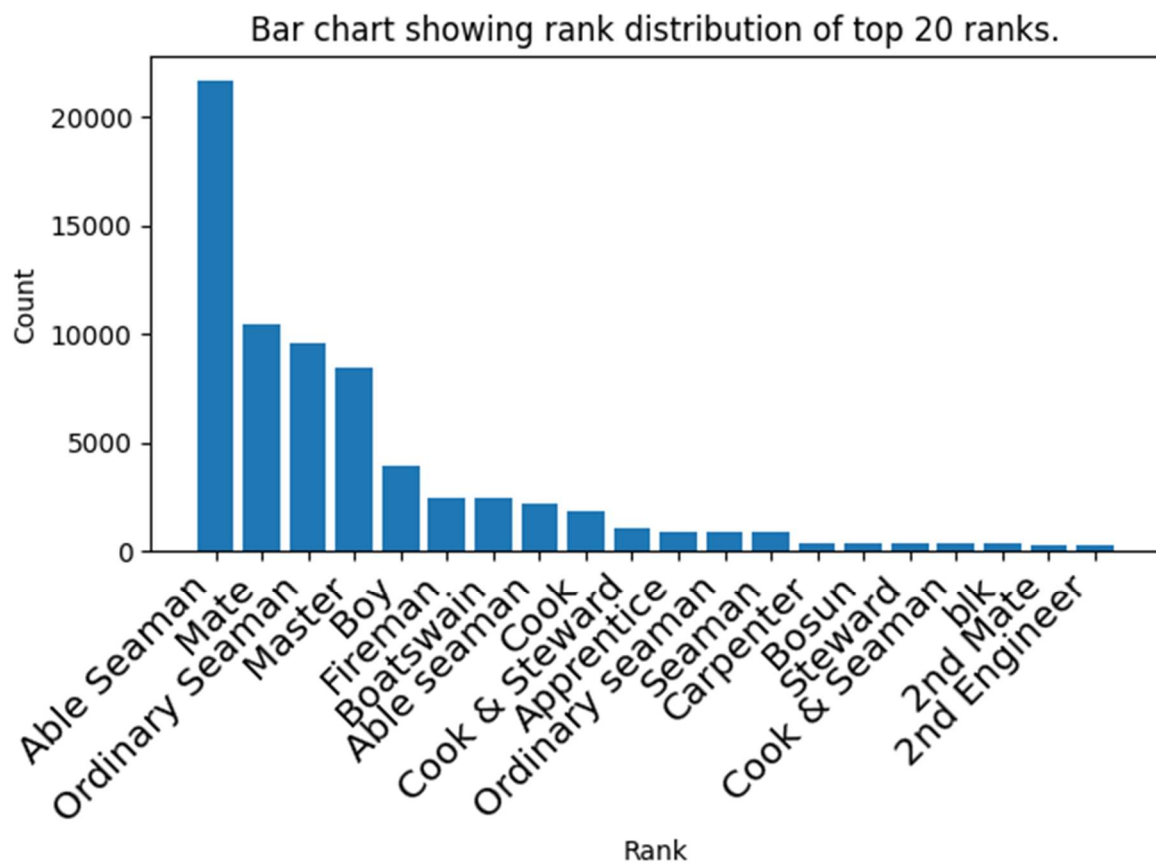
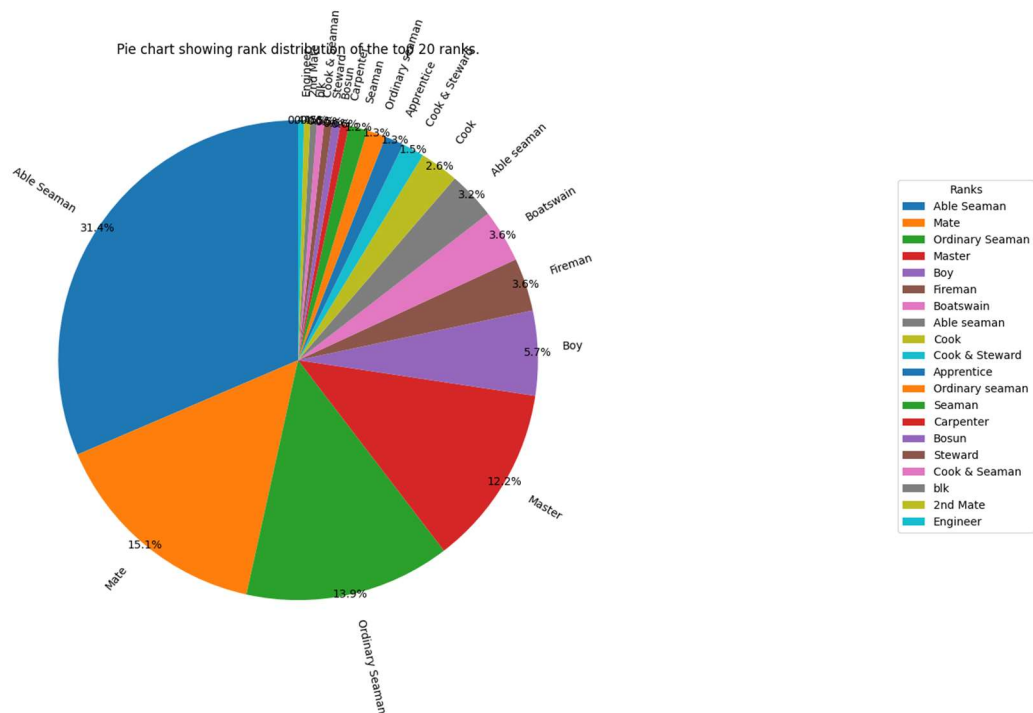
These Tree View widgets can then open up one mariner’s information by double clicking on any mariner within either Tree View (Figure 7). The code for these selections and GUI implementations can be found in lines 188 to 317 in ‘tree_select()’ and ‘tree_select_2()’. All of the widgets have their own scrollbar to allow the user to see lots of information within each widget.

| Id | Vessel Name | Official Number | Port of Registry | Name | Age | Place of Birth | Home Address | Last Ship N |
|--------|------------------|-----------------|------------------|-------------|-----|----------------|-------------------|-------------|
| 3667 | Catherine & Mary | 27303 | Aberystwyth | David Evans | 20 | Newquay | First Ship | |
| 3675 | Catherine & Mary | 27303 | Aberystwyth | David Evans | 20 | Newquay | First Ship | |
| 3689 | Catherine & Mary | 27303 | Aberystwyth | David Evans | 20 | Newquay | Catherine & Mary | |
| 29332 | Alberta | 29301 | Aberystwyth | David Evans | 20 | Newquay | Clifton | |
| 71491 | Mary Sarah | 29306 | Aberystwyth | David Evans | 20 | Newquay | Mary Sarah | |
| 101079 | U. Lansing | 49651 | Aberystwyth | David Evans | 20 | Newquay | Ann & Cath(e)rine | |

Figure 7

2.2.4 “Visualise the proportion of individuals at each rank”

The query for getting the ranks (per unique mariner) can be found in ‘proportion_ranks()’ on line 459. The ranks and counts of ranks (for each unique mariner) is returned from this function. I then use bar_chart() on line 507 and pie_chart() on line 521 to create the appropriate graphs to display the rank distributions. The user can click the respective buttons on the interface to view these graphs.



The graphs are not perfect, but they are functional and meet the requirement. I will discuss issues with data representation later in this report.

2.2.5 “Visualise the promotion track of two individuals”

The first task is getting any two user-selected mariners for whose promotion tracks the user wishes to view. This can be seen in `two_selected()` from line 543 onwards. This takes the two selected items using Tkinter’s functions, and then further narrows that down to just the unique IDs of each person. This allows a query to be passed on line 557 which matches any documents with said IDs, and then uses further code to get the x and y axes for the timeline. The user can click a button in the GUI, after selecting two mariners, which will show the timeline of the promotion track for those two mariners with a legend for each mariner (Figure 8).

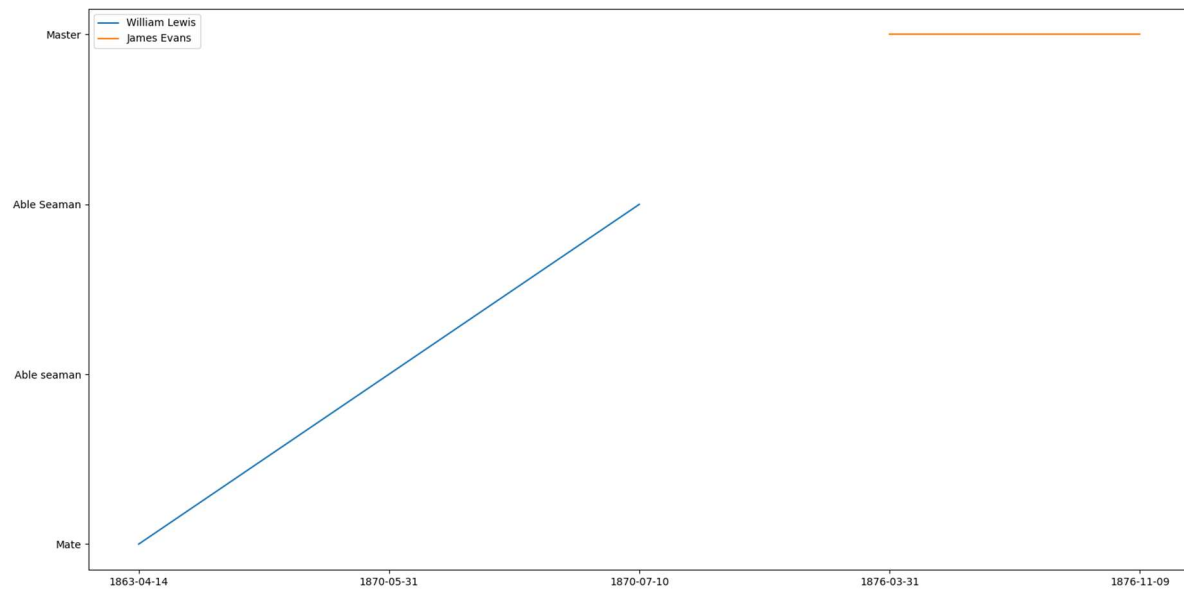


Figure 8

The timeline is not perfect, which again I will discuss later. However, the user is able to see the dates along the x axis which gives an indication of progression, and also allows a natural ordering of the ranks because it is likely that a rank change at a later date is a promotion. Therefore the rank order doesn’t need to be manually coded.

2.3 WHO AND WHERE

The second task of deliverables concerned ‘Who and where is visiting’. The code for completing this task can be found in `who_and_where.py`. Again, code is commented as necessary, and the report will follow the same structure as above.

2.3.1 “Create a histogram of number of crew on each ship”

The first part for creating the histogram can be found in `counting_vessels()` on line 23. This is where a query is passed into the database which returns a list of vessels with the number of mariners they have had throughout all time (or, the time period of the data in the database). The same assumption for a unique mariner as before applies here, where vessel name and official number is grouped with the unique mariner to avoid counting duplicates.

The second part can be found in `get_histogram_data()` on line 69. This takes the data returned from `counting_vessels()` and organises it into data which can be used to plot a histogram (aka sorting, setting upper ranges, defining the bin size).

The third part is where the histogram is actually plotted, in “plot_histogram()” on line 119, which uses the defined values returned from the get_histogram_data() function.

Due to a lack of time towards the end, I have not added a user button in the GUI to be able to allow the user to view this histogram. Rather, the user can run the Python file which will automatically display the histogram. This was not a necessary requirement, and so I did not feel an urgent need to offer this functionality.

2.3.2 “List the possible destinations of ships that belong to the Aberystwyth port [using a tree structure].”

2.3.2.1 “Select top 10 and 5 most visited ports for branch #1 (/joining_port) and branch #2 (/leaving_port), respectively.”

The ‘joining_port’ field in the documents in the database shows the port at which a mariner joined the ship, as each document in the database is per mariner record, per ship log (in the excel files). Therefore, it can be surmised that in order for a mariner to have joined at said joining port, the ship must have visited said joining port. Therefore, the port where the most mariners joined is the port which has been most visited (and likewise for the other top 9 ports). I do not know if this is *actually* true to reality, as perhaps in real life a ship would have visited ports without any mariners joining, but it is true to the data we have been given and allows a workable assumption. The same can be applied to the ‘leaving port’, as the ports where mariners left the ship must have been visited, and the top 5 ports where mariners left would be the top 5 leaving destinations (per top joining port).

To achieve this, within “get_port_maps()” from line 140 onwards, the group operator has been used to get all ships that are unique with the composite of: vessel name, vessel number, joining port, joining date, and leaving port. The date is necessary because it could be that the destination was visited again by the same person but on a different date, and this would need to be counted as a new visit to this destination. However, this does not account for all possible scenarios, for example: What if someone joins today and someone joins tomorrow? Did the ship visit twice, or did it just stay overnight?

Another group operator is then used in the aggregation, which gets unique combinations of joining port and leaving port and adds to a count the number of times this unique combination appears (having already factored for date and other issues which would cause repeats). This allows the top 10 to be found by taking the sum of the counts for every time the joining port is mentioned in a combination (using a variation of dictionaries and lists in Python). The top 5 is then calculated by sorting the combinations, taking only 5 for each joining port, and adding these to an array with the key of the joining port.

2.3.3 “Your code must visualise and explore the journey pattern (route) of most Aberystwyth ships, i.e., the most repeated routes.”

This is achieved in the function “visualise_tree()” from line 245 onwards. This takes the data returned from the get_port_maps() function and converts it into a tree structure using the Python libraries ‘anytree’ and ‘graphviz’.



Figure 9

The resulting visual tree is very simple and basic, and quite flawed, though it fulfils the intended function.

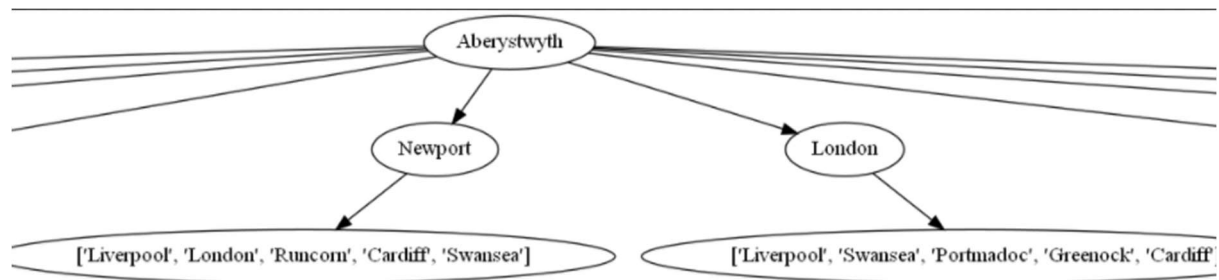


Figure 10

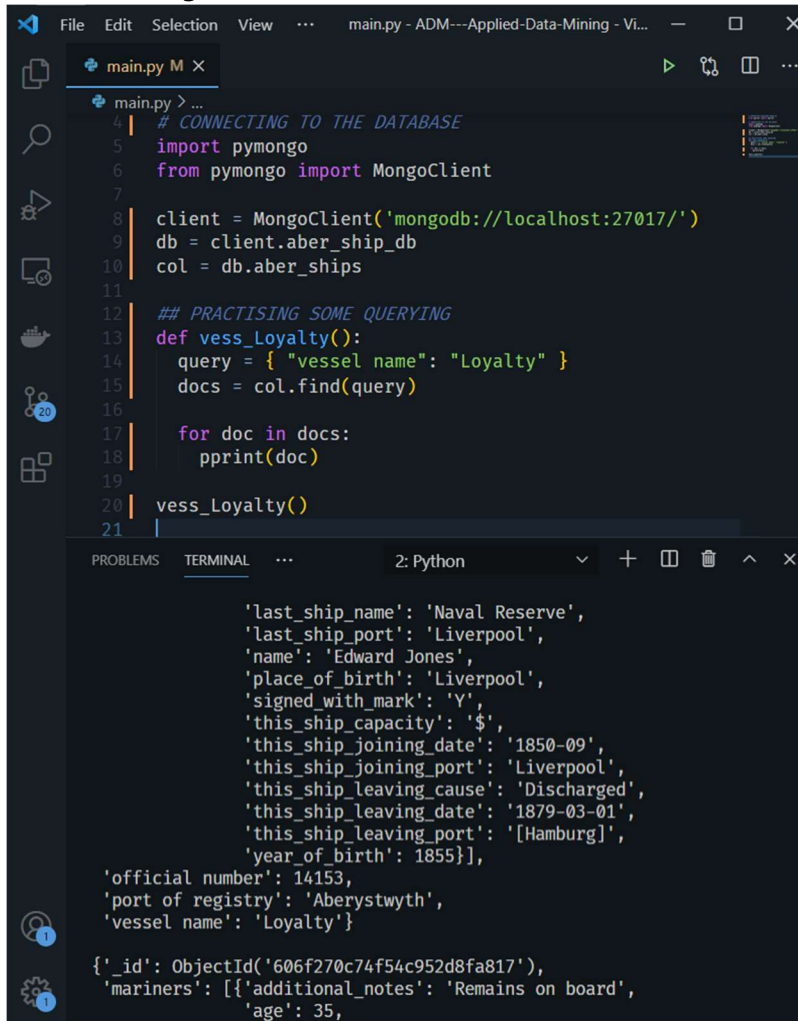
3 DISCUSSION

An overall analysis of your project, this should summarise evidence from your results section, it can also be more subjective (but still evidence based).

The brief did not cover conducting any form of research nor analysis, and so where you would typically expect a discussion in a report to cover said analysis and a discussion of it, this report will not feature that. I will, however, use this space to discuss some other relevant items which are yet to be addressed.

3.1 TESTING AND QUALITY ASSURANCE

3.1.1 Testing Queries



The screenshot shows a VS Code editor window with a file named `main.py`. The code is a Python script that connects to a MongoDB database and queries for a specific vessel. The terminal output shows the result of the query, which is a dictionary containing various details about the vessel 'Loyalty'.

```
4 | # CONNECTING TO THE DATABASE
5 | import pymongo
6 | from pymongo import MongoClient
7 |
8 | client = MongoClient('mongodb://localhost:27017/')
9 | db = client.aber_ship_db
10 | col = db.aber_ships
11 |
12 | ## PRACTISING SOME QUERYING
13 | def vess_Loyalty():
14 |     query = { "vessel name": "Loyalty" }
15 |     docs = col.find(query)
16 |
17 |     for doc in docs:
18 |         pprint(doc)
19 |
20 | vess_Loyalty()
21 |
```

```
'last_ship_name': 'Naval Reserve',
'last_ship_port': 'Liverpool',
'name': 'Edward Jones',
'place_of_birth': 'Liverpool',
'signed with mark': 'Y',
'this_ship_capacity': '$',
'this_ship_joining_date': '1850-09',
'this_ship_joining_port': 'Liverpool',
'this_ship_leaving_cause': 'Discharged',
'this_ship_leaving_date': '1879-03-01',
'this_ship_leaving_port': '[Hamburg]',
'year_of_birth': 1855}],
'official number': 14153,
'port of registry': 'Aberystwyth',
'vessel name': 'Loyalty'}

{'_id': ObjectId('606f270c74f54c952d8fa817'),
'mariners': [{'additional_notes': 'Remains on board',
'age': 35,
```

Figure 11

I tested the output of my queries by pretty printing (`pprint()`) the output to the console (Figure 11). This allowed me to view the structure of the output, which Python recognised as objects (dictionaries) with key value pairs. I have sometimes left these debugging processes in my code as commented functions, but other times I removed them. However I did perform this check for all of the queries, sometimes using the VSCode debugging process to view multiple variables side by side as they changed throughout my code (such as in for loops and if statements).

3.1.2 Maintaining Semantics

Throughout the code I have tried to keep my variable assignments and data 'keys' or 'object names' consistent, meaning it is clear what each variable actually represents. This maintains both consistency of the data within the Python code itself, but also consistency of the data in transforming this to a GUI, and also consistency of the data in relation to it's digital record. For example, I don't reassign the document field names when I use them in queries in the Python code unless they need to be clarified further.

```

query = [
  {
    "$group": {
      "_id": {
        "vessel_name":
          "$vessel name",
        "vessel_number":
          "$official number",
        "name": "$name",
        "age": "$age",
        "DOB":
          "$year_of_birth",
        "place_of_birth":
          "$place_of_birth",
      }, # gets documents
    }
  }
]

```

Figure 12

In Figure 12, matching 'variable' and 'field' names are used when assigned in the \$group operator, to maintain consistency throughout the data.

3.1.3 Testing Data Inconsistencies

I would check the code routinely to evaluate whether the output was producing ideal results (aka not really messy data, and not any blank or 'None' or 'Remains' data, particularly for the leaving ports).

```

> 'Llanelly': [None, 'Swansea', 'Runcorn', 'Liverpool', 'Dublin']
> 'Liverpool': ['[Hamburg]', '[Hamburg?]', 'London', 'Swansea', 'Cardiff']
> 'Cardiff': ['Liverpool', 'London', 'Swansea', 'Bristol', 'Runcorn']
> 'Swansea': ['Liverpool', 'London', 'Runcorn', 'Blk', 'Bristol']
> 'Newport': ['Liverpool', 'London', 'Runcorn', 'Cardiff', 'Swansea']
> 'London': ['Liverpool', 'Swansea', 'Portmadoc', 'Greenock', 'Cardiff']
> 'Portmadoc': ['London', 'Aberdovey', 'Hamburg', 'Swansea', 'Liverpool']
> 'Runcorn': ['Liverpool', 'Swansea', 'Cardiff', 'Portmadoc', 'Blk']
> 'Porthmadog': ['Blk', 'London', 'Runcorn', 'Remains', 'Hamburg']
> 'Aberdovey': ['Portmadoc', 'Liverpool', 'Aberystwyth', 'Swansea', 'Runcorn']
len(): 10

```

Figure 13

I could then use this information to further refine my query to make sure it did \$ne 'blk', or 'Blk', or 'None', or 'Remains', and so on (Figure 14). \$ne means not equals which means the query only returns documents where the specified fields do not equal the specified value.

```

{"$match": {"this_ship_joining_port": {"$ne": "blk"}}},
{"$match": {"this_ship_joining_port": {"$ne": "Aberystwyth"}}},
{"$match": {"this_ship_leaving_port": {"$ne": "blk"}}},
{"$match": {"this_ship_leaving_port": {"$ne": ""}}}

```

Figure 14

3.2 OTHER PROBLEMS

3.2.1 Visualising The Tree (Deliverable 2, Part 2)

It would be better if the formatting was slightly better so the tree wasn't as stretched and didn't need to be zoomed into to view the actual data (Figure 9). Also, the items on the third branch were still in their list form (Figure 10), where I think it would look better if they appeared as separate nodes themselves as children of the 'joining_port' node. However, with the limited time and tools I was able to find online, the tree serves its purpose.

3.2.2 Data Inconsistencies / Dealing with Difficult Data

3.2.2.1 Dates on the Timeline

```

raise TypeError(
TypeError: 'value' must be an instance of str or bytes, not a datetime.datetime

```

Figure 15

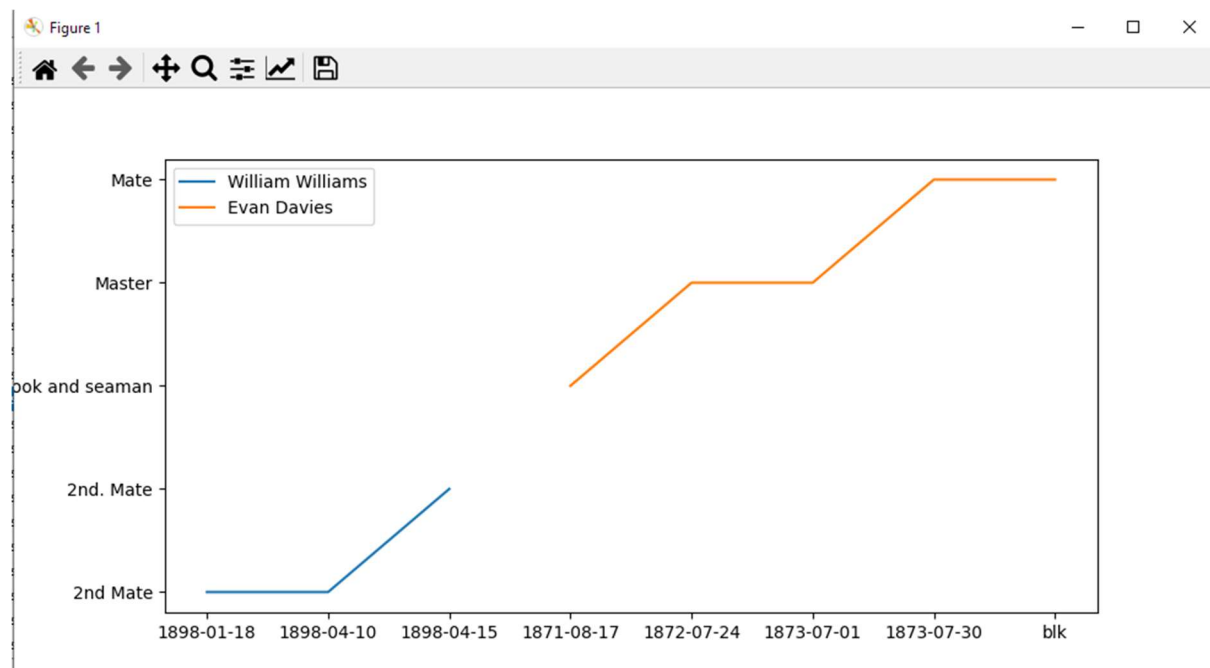


Figure 16

DateTime objects could not be used because of the inconsistencies. It was easier to convert dates to strings than to convert strings to dates, so strings were chosen. However, that meant that the strings could not be ordered by date unless, again, they were parsed as dates. Due to the inconsistencies of what is entered into the dates field, it would be very difficult to parse the strings as dates. Therefore, as with the timeline above, though the ranks are mapped to the dates at which they were updated, they are not in order on the x axis. So the dates for each mariner, on the blue

and orange line respectively, are not co-ordinated. However, the visualisation is still effective, and the promotion track of each individual can be seen, therefore the requirement is met. This is something which (with time and more experience with Python and parsing dates) could be improved upon in future versions.

3.2.2.2 Machine Learning to Improve Data

Despite including explicit parameters in my queries to filter out messy data, some still made it through (Figure 17, Figure 18, Figure 19). It is not time efficient to manually filter out any queries by 1) randomly testing a bunch of queries to see what data is output and 2) using any messy outputs as filters in the queries. Machine Learning would automate this process, tidy up the data, and reduce the need for manual interference. It would make working with the data much more efficient and would produce more consistently accurate results from queries.

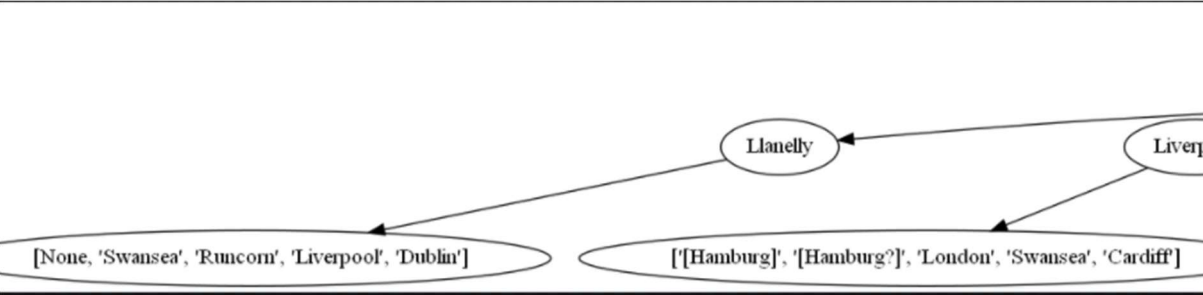


Figure 17

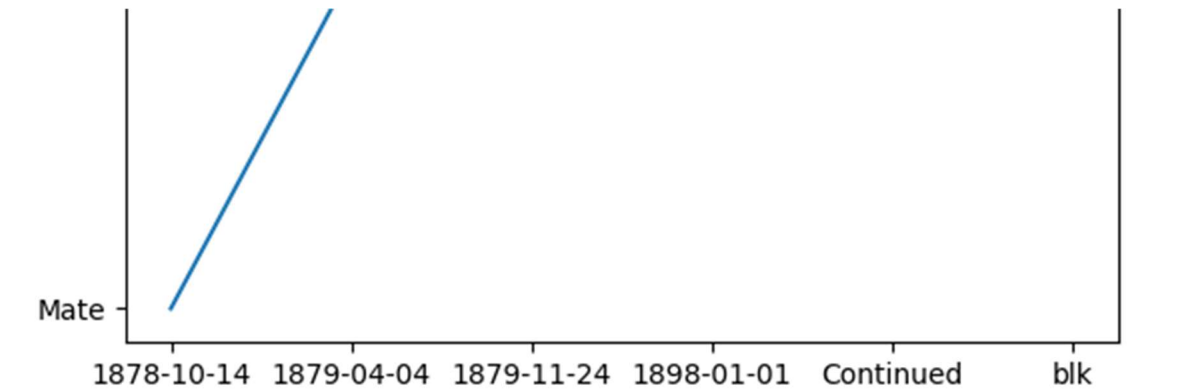


Figure 18

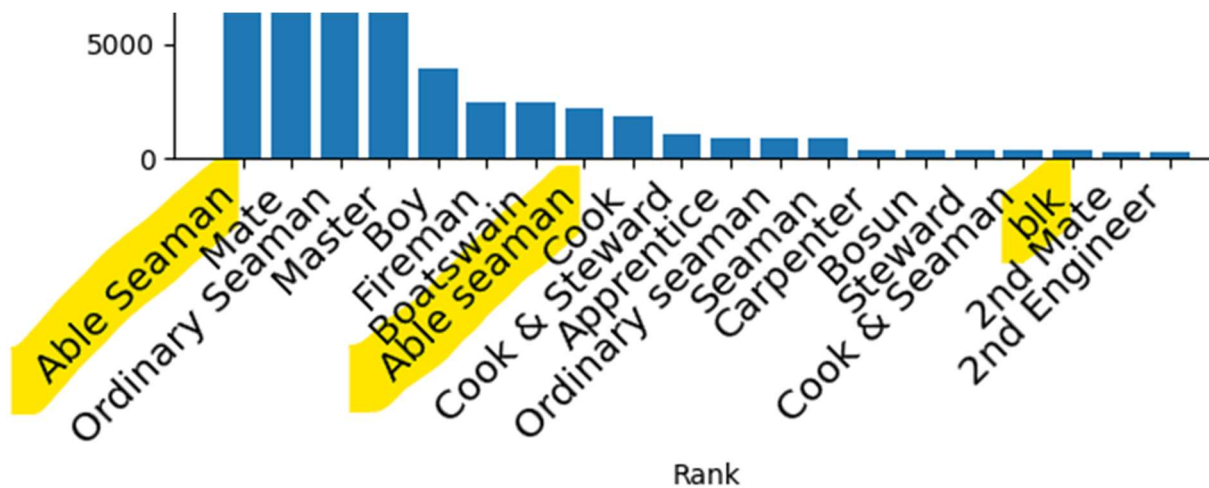


Figure 19

Additional work could be done to help match similar values such as names and places (like removing brackets or differences due to upper/lower case), help create consistent dates from date strings and date objects, and so on.

4 SELF-REFLECTION

Although it is not normally part of a peer-reviewed paper, it is university policy that you include a reflection section here. Reflection on your work has been shown to improve your learning in the long term. You should consult the marking schema and position your own work on it. You should also reflect on how you would do it differently if you were to repeat the exercise.

4.1 RELATING TO THE MARKING SCHEMA

4.1.1 Functionality

I believe my code is fully functional in all requirements, as I have demonstrated above how I meet each requirement. I also feel the GUI helps tie all of the requirements together, so I think perhaps I have *some* optional advanced functionality (buttons, multiple GUI windows, multiple charts for different requirements).

4.1.2 Robustness

I would say my code performs well to produce normal looking outputs. I did code checks for some abnormal scenarios, though some still managed to slip through. I would therefore place myself between the upper two grade categories in this regard.

4.1.3 Visualisation

I believe my visualisations are clear, fully functional, and I explained what the GUI is for so it is logical and complete and understandable.

4.1.4 Documentation

I would say I could have made more use of comments in my code, and I could have made it easier on myself in writing this report by documenting my experiences and thoughts throughout the process.

I attempted to do this, but largely documentation fell by the wayside and I had to reflect on my code after-the-fact, as opposed to reflecting in-the-moment.

4.1.5 Implementation

I would say I have used a range of resources from Tkinter, Matplotlib, Anytree, pprint, pymongo, Graphviz, and perhaps more which I'm missing in this list. I've nailed the basics of each library and I think I have used them all to good effect. I do not know whether my code itself is 'elegant', I think I have some work to do in that regard, but the output of the code is elegant and I'm happy with my performance in this regard.

4.2 UI LAYOUT

I dislike how the buttons are not aligned in the UI. This is very minor, and I prioritised spending my time on the main functions to meet the requirements, however it is something which I would've liked to improve.

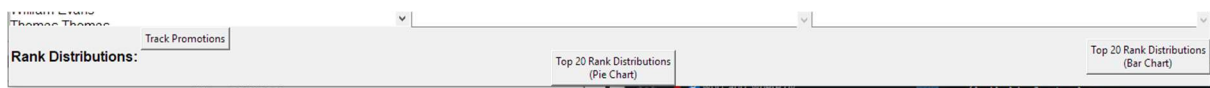


Figure 20

4.3 BETTER PYTHONIC CODE, FORMATTING AND LINTING

I have not made use of Classes in my code, which I think is one major downfall. It would have been much easier to relate the functions to one another, and to the user interface particularly, if I had made use of Classes and used functions within those classes. As I did not do that, I had to use some global variables (not ideal), and also used a specific order to nesting functions to get them to achieve the desired output.

I only realised auto-formatting was something I could do with VSCode towards the final tasks of this brief. It would have saved a lot of time earlier in the process if I did not have to spend time manually formatting bracket alignments and so on. It did however mean my code looked more consistent and clearer when I applied the auto-formatting feature.

I also did not realise I didn't have linting enabled in VSCode until I'd finished the first task 'Individual Stories'. This meant that when I turned linting on and ran checks against my 'individual_stories.py', a seemingly endless list of problems was presented to me (such as unused variables, unused imports, messy indentation). They were not code breaking so I could ultimately ignore them, but it will definitely encourage best practice if I follow the guidance provided by linting to make my code efficient and organised and non-problem inducing.

4.4 RUNNING A CONTAINER WITH MONGO

```
PS C:\Users\caraj> docker run -d --name mongo
>> -v C:\Users\caraj\OneDrive\Documents\GitHub\ADM---Applied-Data-Mining:\data\
b
>> -p 27017:27017
>> mongo:latest
c75ffa0ae4e067155643047f754b9f2e5603414f0ba9c62988d7837242e9644a
```

Figure 21

For this assignment I decided to run mongo locally within a Docker container. This is the first time I have used Docker but I saw it as a good opportunity to become familiar with a tool which I know to

be widely used in the industry. It allowed me to create a virtual environment with my local mongo server, without messing with my actual system's installs and paths etc. It took a while to become familiar with but now I know the basic functionality I can see how useful it is, and I will definitely use it more in the future.

4.4.1 ENTERING THE BASH TERMINAL WITHIN THE CONTAINER AND THEN EXECUTING THE MONGO COMMAND TO BE ABLE TO PERFORM MONGO OPERATIONS

```
PS C:\Users\caraj> docker exec -it mongo bash
root@c75ffa0ae4e0:/# mongo
MongoDB shell version v4.4.4
connecting to: mongod://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
```

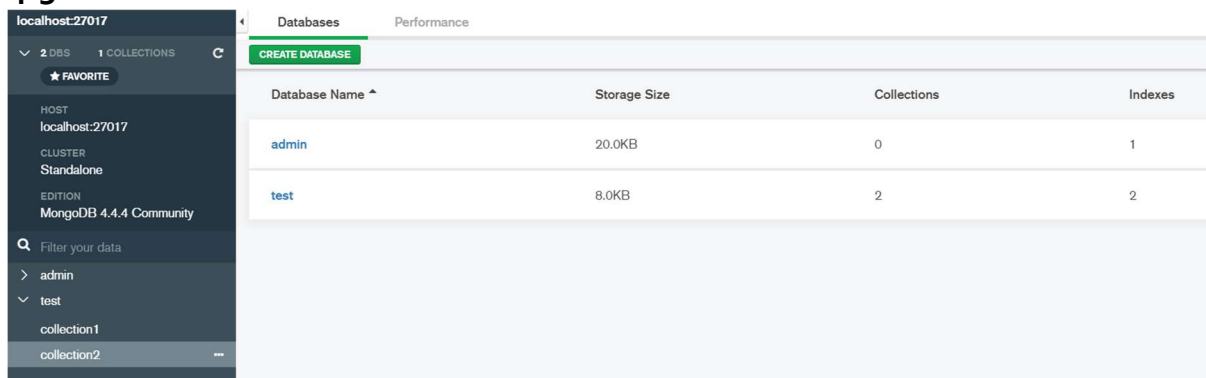
Figure 22

4.4.2 Executing mongo operations

```
> show collections
ships
> show dbs
admin 0.000GB
config 0.000GB
local 0.000GB
> _
```

Figure 23

4.5 GUI INTERFACE TO QUICKLY VISUALISE MONGO OPERATIONS



The screenshot shows the MongoDB GUI interface. On the left, there is a sidebar with navigation options like 'HOST', 'CLUSTER', 'EDITION', and a search bar. The main area displays a table of databases and collections. The table has columns for 'Database Name', 'Storage Size', 'Collections', and 'Indexes'. Two databases are listed: 'admin' and 'test'.

| Database Name | Storage Size | Collections | Indexes |
|---------------|--------------|-------------|---------|
| admin | 20.0KB | 0 | 1 |
| test | 8.0KB | 2 | 2 |

Figure 24

This GUI interface was incredibly useful to me as a visual learner, as it allowed me to quickly visualise the exact structure of my data without having to perform mongo queries and operations. I didn't use it to create any code or formulate any queries, so it did not perform any 'write' functions. I did use it a lot to 'read' my data.

4.5.1 Example of GUI interface visualising mongo operations

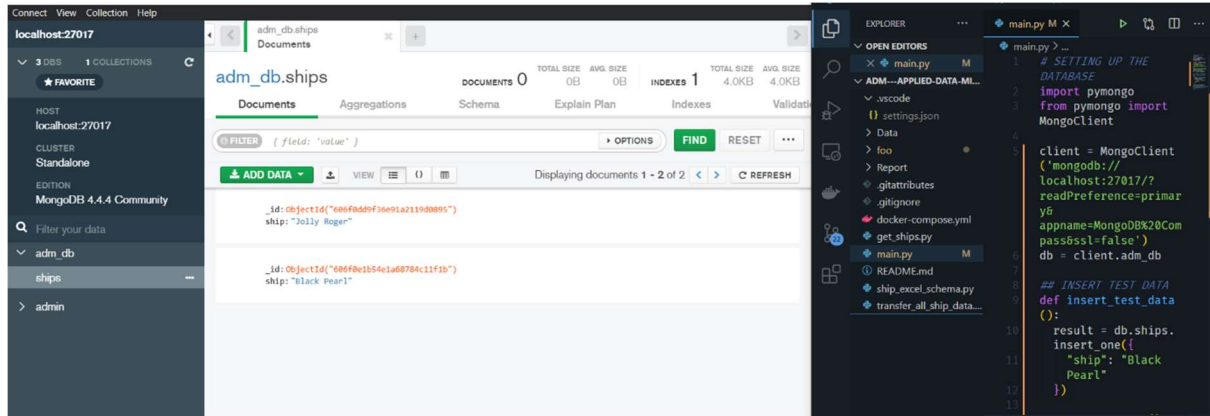


Figure 25

4.6 CONVERTING THE BRIEF INTO REQUIREMENTS

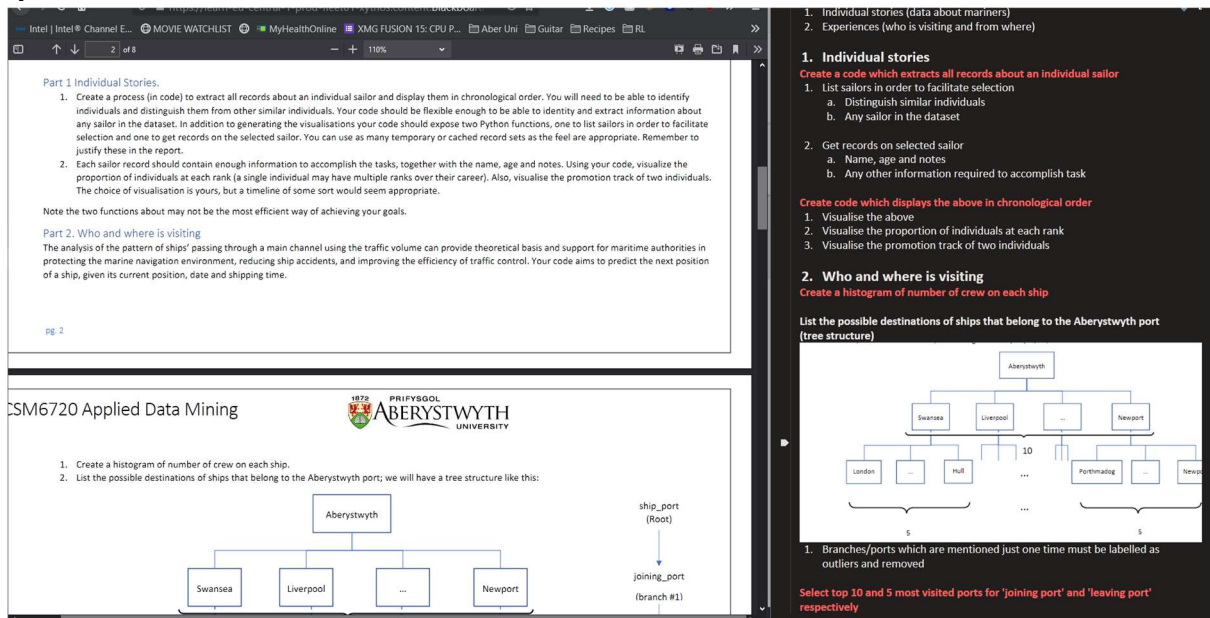


Figure 26

At the beginning I read the product brief and converted this into a list of requirements which made sense for me. This allowed me to then apply logic to know which functions I would need to write, how they would link together, and how they would meet the requirements.

4.7 DEBUGGING WAS VERY USEFUL

I use VSCode's debugging tool a lot to work out what code was doing (by 'Watch'ing variables) and why it was potentially not doing what I wanted it to do.

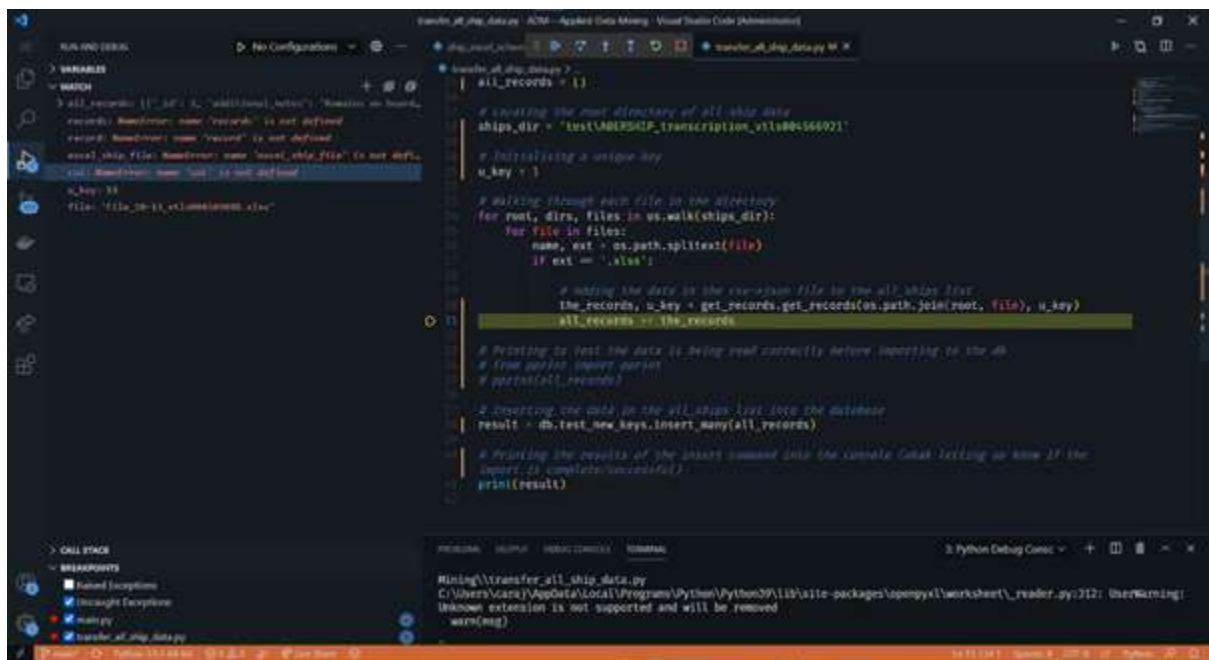


Figure 27

5 REFERENCES

- [1] E. Hampshire and V. Johnson, "The Digital World and the Future of Historical Research," *Twentieth Century British History*, vol. 20, no. 3, pp. 396-414, 2009.
- [2] K. D. L. Mandemakers, "Best Practices with Large Databases on Historical Populations," *Historical Methods: A Journal of Quantitative and Interdisciplinary History*, vol. 37, no. 1, pp. 34-38, 2004.

6 TABLE OF FIGURES

| | |
|-----------|----|
| Figure 1 | 2 |
| Figure 2 | 3 |
| Figure 3 | 4 |
| Figure 4 | 4 |
| Figure 5 | 5 |
| Figure 6 | 6 |
| Figure 7 | 6 |
| Figure 8 | 8 |
| Figure 9 | 9 |
| Figure 10 | 10 |
| Figure 11 | 11 |

| | |
|-----------|----|
| Figure 12 | 12 |
| Figure 13 | 12 |
| Figure 14 | 13 |
| Figure 15 | 13 |
| Figure 16 | 13 |
| Figure 17 | 14 |
| Figure 18 | 14 |
| Figure 19 | 15 |
| Figure 20 | 16 |
| Figure 21 | 16 |
| Figure 22 | 17 |
| Figure 23 | 17 |
| Figure 24 | 17 |
| Figure 25 | 18 |
| Figure 26 | 18 |
| Figure 27 | 19 |