CATALOGUING SYSTEM FOR AMATEUR COLLECTORS

NEA PROGRAMMING PROJECT

SAMI HATNA
CANDIDATE NUMBER: 5072
CENTRE NUMBER: 40531

Contents

Section 1: Analysis

Introduction

Currently, there is no software on the market which enables amateur collectors to quickly and efficiently catalogue their collections. I intend to remedy this issue by devising a cataloguing system specifically tailored towards amateur collectors with the aims of allowing them to log the contents of their collections; present their collections in an aesthetically pleasing manner; and allow them to manage loaning out items by generating notifications when an item is overdue.

I intend to program the front-end of my application in Python - as it is a language which I am fairly experienced in – and I will be using the PyQt library to create my Graphical User Interface. I will use SQL to program the backend of my application as it is an industry standard for creating secure databases. I will also be learning how to interface with an SQL database using Python; how to carry out web-scraping to extract data from websites and how to create stylesheets using CSS.

I felt drawn to the idea of creating a cataloguing system for my NEA because I myself collect books and vinyl records and have never been able to find a simple, well-organised and appropriately priced piece of software to allow me to catalogue my collections. As a result, I want to create a unique application which satisfies my own needs and the needs that I identify from other collectors during my research. The idea was also appealing because it has a large and clearly identifiable target market – approximately a third of people in the UK collect something [1] – making it easy to tailor my software to their requirements.

The Problem

Collecting is a hobby that encompasses all walks of life and all spheres of society. However, there is currently no computational solution to the issue of keeping track of collections, cataloguing items and managing item loaning. Cataloguing one's collection has a wide range of benefits including the ability to manage large collections with relative ease and share collections with others in a digital format.

The collector's software currently on the market generally suffer from at least one of these three issues:

- They have outdated and cluttered user interfaces which overcomplicate the process of cataloguing items and detract from the usability and aesthetic appearance of the software.
- They force users to pay subscription fees or make expensive one-time purchases for software that oftentimes isn't worth paying up to £75 for.
- Users cannot curate several different types of collection using a single piece of software. Instead, they have to download separate applications for each type of collection. For example, if I collected both vinyl records and video games, I would have to download a piece of software specifically for cataloguing games and also download another application for cataloguing records. This is a problematic approach as it does not improve ease of use and takes up more storage than a single multi-purpose app.

Stakeholders

The core market my software is aimed towards is amateur collectors who collect either books, films, video games or records. The demographics of book, film, video game and music collectors are most suited to this software because they are four of the most widespread types of collection, thus giving my software the widest possible appeal within the collecting community.

Because my software would be of most use to collectors with large collections who need the assistance of a computer with handling large datasets, I will have to take into consideration that certain age ranges have more disposable income than others and are consequently more likely to have bigger collections. This is supported by the Journal of International Business and Cultural Studies' statistical report on worldwide collecting consumption behaviours where they observed that 48% of respondents to their survey of serious collectors had annual incomes of $50K – $150K [2]. As a result, I would consider my target age range to be collectors between the ages of 25 and 60, as users in this age range will be in employment and will therefore be able to spare more money to spend on their hobbies. Many collectors also use social media or their personal websites to present their collections and socialise with other members of the community. The software would be of significant use to these particular members of my target demographic because it would allow them to remain on top of where their collection is in terms of size as well as making sure that they can store all the relevant metadata about individual items such as their market value or their release date. Furthermore, these more social collectors will find the loan management system particularly beneficial as they will be able to ensure that items aren't kept too long by borrowers or stolen.

In summary, the stakeholder is a male or female in the age range 25 – 60 with enough surplus income to devote a lot of funds to their collecting hobby. Their collection has expanded past the size of around 50 items and they are now looking for an efficient method of cataloguing said collection. Ideally, the stakeholder will be socially active in collecting communities and on social media and thus will make good use of the loan management system.

Existing Solutions

Microsoft BookCloud

Microsoft BookCloud is described on the Microsoft store as *"a quick and simple way to catalogue and inventory the books you own, or you want to read later"*

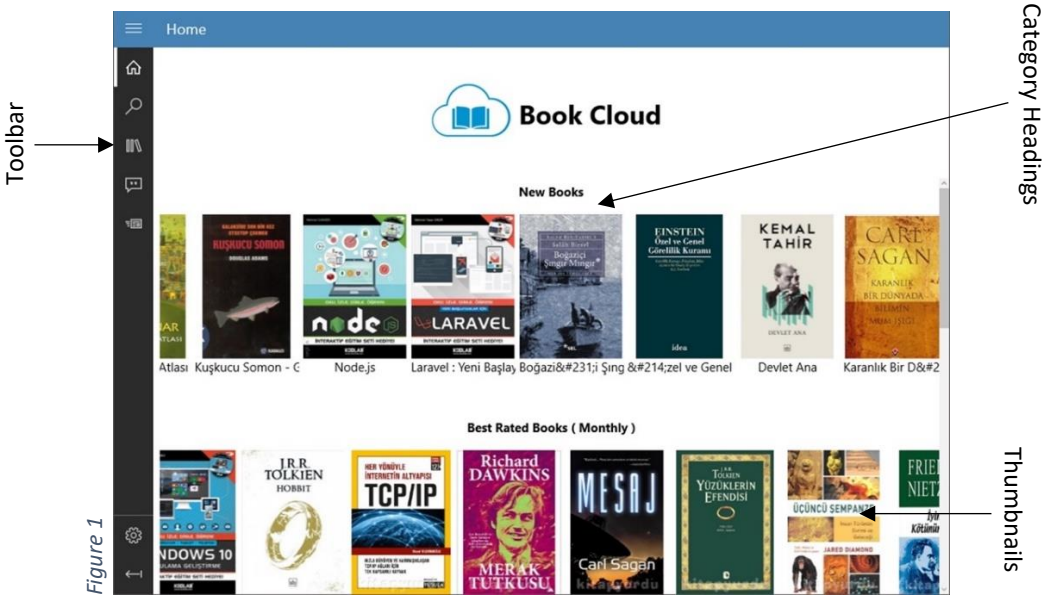


Figure 1

The user interface is comprised of a central widget containing the thumbnails of books in the user's collection and a sidebar allowing the user to navigate between different panes. Microsoft BookCloud costs £0.99 and is only available for Windows computers and Xbox consoles.

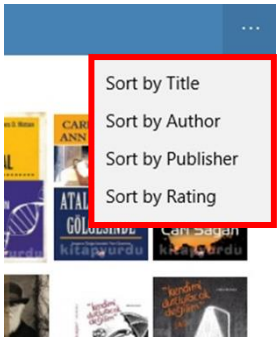


Figure 2: Sorting Menu

The use of thumbnails to present items in a collection is beneficial as it is quite aesthetically pleasing. However, from a functionality perspective it may be better to present items in a list, as a list can be split into columns which can contain descriptive data for each item.

The program also allows users to sort their collection based on various criteria via a drop-down menu. Allowing users to sort through their collection based on certain conditions is a function that I will include in my software to improve ease-of-use and allow users to quickly find the items they are looking for.

An important feature of this software is the barcode searching functionality. This would be a useful feature to implement in my own solution through web-scraping as it helps simplify the process of adding an item to your collection.

However, the Microsoft BookCloud app suffers from irregularities in its barcode searcher, with one reviewer on the Microsoft Store describing it as "functional but irritating" because "out of 32 books it only let me search the barcode for 3" and it only accepts 13-digit ISBN codes. These issues are a massive detriment to the application's functionality because users paying £0.99 will expect one of the application's principal features to work properly. If I am to implement a barcode searcher into my software, I must ensure that it is not error-prone and works with both 10-digit and 13-digit ISBN codes.

Calibre

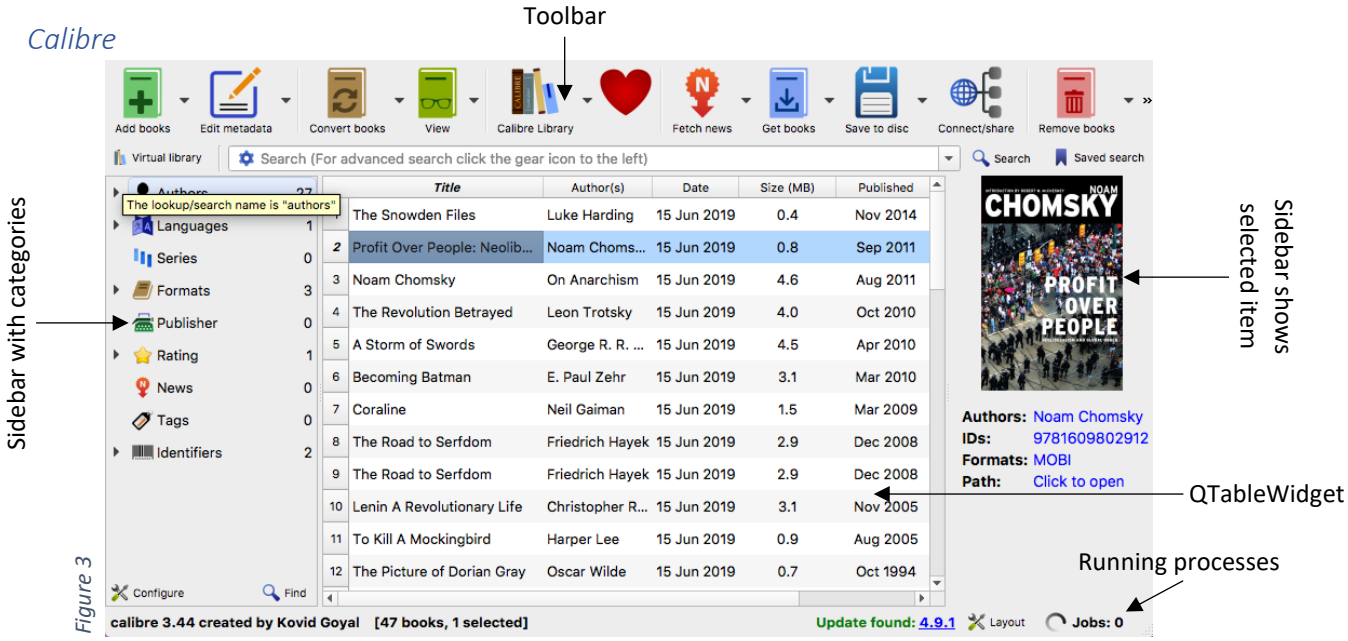


Figure 3

Calibre is a free and open-source e-book management tool. Calibre is used in over 200 countries and is updated and maintained by dozens of volunteers. Its success means that, despite it differing from my software in that it is not solely a cataloguing software, it is a good reference point for what to get right when creating software aimed at collectors.

Calibre is written in Python, JavaScript, C++ and C. The user-interface was created using the PyQt library. The user-interface is centred around a QTableWidget which displays a virtual library

containing the user's e-books. The central list is surrounded by a toolbar at the top, a navigation pane on the left and a sidebar displaying the selected book on the right. The application makes heavy use of icons and other visuals to maximise ease of use. Calibre has many functions on top of its cataloguing purposes, including the ability to convert and edit e-books as well as allowing users to sync books across several e-readers.

Positive aspects of the software that I would consider applying to my solution include the ability to easily navigate collections using the sidebar and the search bar. In addition, when using Calibre, one of its standout features for me was its customisability and the amount of freedom it offers the user. I think a core requirement of my solution would be maximising user privileges so as to allow them to use the software to its full potential.

Calibre is limited by the fact that it only encompasses e-book collections and thus is not an entirely relevant solution to the issue I have identified. From a technical standpoint, Calibre is incredibly complex and consequently suffers from numerous security vulnerabilities. For example, version 2.24.0 of Calibre was found to contain an embedded copy of Mozilla's WOFF decoder [3]. The decoder is susceptible to an integer overflow error which, when exploited, can allow a remote attacker to execute code on an affected system. This is evidence that overcomplicating software with excessive features can result in oversights when debugging. As a result, it is worth bearing in mind that my software should be as thoroughly tested as possible to ensure that it does not contain any bugs and runs well.

Excel or LibreOffice Base

Figure 4

Headings

Title	Director	Studio	Release Date	Genre	Condition	Notes
The Shining	Stanley Kubrick	Warner Bros.	1968	Horror	Mint	Loaned to Bob Smith
Barry Lyndon	Stanley Kubrick	Hawk Films	1975	Historical	Poor	
A Clockwork Orange	Stanley Kubrick	Hawk Films	1972	Dystopia	Average	Scratched Disk
Dr. Strangelove	Stanley Kubrick	Columbia Pictures	1964	Satire	Mint	Contains Director's commentary
Eyes Wide Shut	Stanley Kubrick	Hobby Films	1999	Drama	Poor	
Full Metal Jacket	Stanley Kubrick	Pinewood Studios	1987	War	Mint	
2001: A Space Odyssey	Stanley Kubrick	MGM	1968	Sci-Fi	Average	Damaged case

Buttons open sorting menus

Many collectors choose to use Microsoft Excel, LibreOffice Base or any other equivalent spreadsheet software to create custom virtual catalogues. Users can either create their own cataloguing spreadsheets from scratch or use pre-made online templates like the one pictured above.

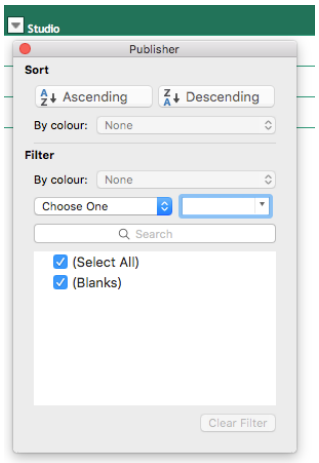


Figure 5: Multitiered sorting

Using spreadsheets is the least restrained way to catalogue a collection I came across during my research, as users can add as many columns of descriptive data they desire and can create spreadsheets for any type of collection they have. In addition, the template I experimented with made use of Excel's multitiered sorting functionalities to help users better visualize their collections.

My solution would take into account the merits of Excel's customisability. However, Excel is not a particularly visually stimulating way to present a collection so I must ensure that my solution is aesthetically pleasing enough to reflect the pride that most collectors take in their collections.

Collectorz

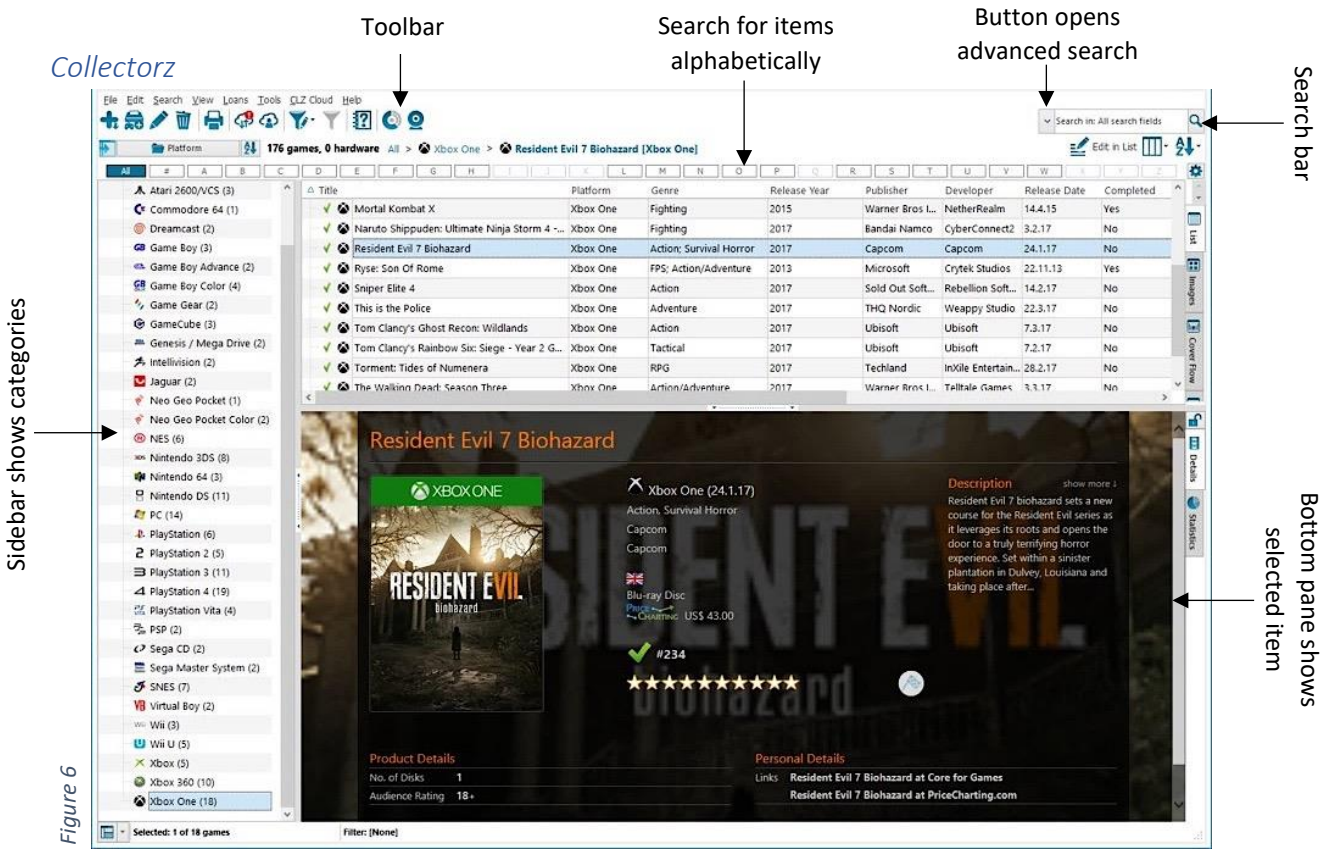


Figure 6

Collectorz is a collection database software available on Windows, MacOS, Android and IOS. They offer applications for movie, book, music, comic book and game collectors as well as a cloud storage service for syncing collections which can be accessed through a web application. Similar to Calibre, the user interface comprises of a central list of items surrounded by a navigation sidebar and a toolbar for performing functions.

Collectorz produces graphs based on a user's database. This is a feature that I believe would be useful to implement in my software by using the Matplotlib python library. It also allows users to export their databases to text or XML files. The main benefit of Collectorz was that their software felt well-optimised for the machine I was running it on and was extremely lightweight and responsive; attributes I would like to carry forward to my own software.

The main drawback of Collectorz is that it is a subscription-based service. As a result, if a user would like to use Collectorz's desktop and mobile applications as well as the CLZ cloud connect service to catalogue their collections, they would have to pay a fee of £32.45 per month. The choice to move to a subscription model appears to be unpopular with users. For example, one reviewer on Trustpilot wrote that the software is "plagued by an out of age annual subscription system" and another wrote that "I like the program, but I do not like the subscription service" [4]. It is thus safe to assume that the demographic of amateur collectors is probably more inclined to use free or one-time payment software.

Another issue with Collectorz that I need to be aware of when creating my software is the viability of providing users with a pre-existing master database. Collectorz provides users with an online database from which they can search for items and automatically add them to their collections without having to manually input details themselves for each item. However, in my own experience of using their software, I have found that the online database contains very little unique and niche items such as comics with variant covers or rare pressings of records. As a result, in my solution I

think it will be better to allow users to input data manually rather than using a premade database so as to allow them to easily add rare and variant items.

Why the Problem is Suited to a Computational Approach

Datamining

Datamining is the process of looking for general trends in large datasets. Because the backend of my software will be programmed using SQL, I will need to use datamining to identify useful patterns and relationships between large volumes of data stored within my relational database. After identifying these patterns/relationships, I can streamline and optimise my database design to improve performance. Data mining can also be used to implement a search function and to allow users to apply filters to their collections.

Backtracking

Backtracking will allow me to implement failsafe processes in case a process fails. This will be most useful when interfacing with the SQL database because, if the program is unable to connect with the database, it could backtrack and display an error message or use a backup save of the database.

Decomposition

The process of adding an item to a collection can be decomposed into a series of sub-steps:

1. Make a connection with the SQL database
2. Allow the user to input details
3. Insert the data into the SQL database
4. Save changes made to the SQL database
5. Refresh the front-end table

Abstraction

An item within a collection is a complex entity for which a large amount of data can be collected. I will therefore have to abstract collection items into a set of attributes and store each attribute as a field within a table. For example, if a book was our entity, it could be abstracted into its *title, author, publisher* and *rating*. I will also be using an object-oriented approach when programming my solution which will mean that I will have to use abstract classes and abstract methods to make my code versatile.

Visualisation

Visualisation will allow me to present data in a visual format. This technique will be implemented in my solution through the creation of graphical breakdowns of a user's collection, which will be programmed using the matplotlib python library and data from an SQL database. I will also use visualisation during the development process of the software by creating UML (Unified Modelling Language) diagrams to visually represent classes and ER (Entity-Relationship) diagrams to help plan database structures.

Pipelining

Pipelining is the process of taking a task and splitting it up into smaller tasks and then overlapping the processing of each sub task to speed up the overall process. I can use this computational method when querying data from my database, as large queries can be split up into smaller sub queries which will be overlapped to speed up the process of retrieving data.

End-User Interviews

My interviews with potential end-users will be carried out using a digital survey created using Google Forms. I intend to send the survey to several collectors I know personally, and I will also make the survey available to some online collecting communities such as Facebook groups and Reddit communities. This should allow me to get a wide range of responses from collectors of varying levels of commitment to their hobby.

Questions

1. What do you collect?
2. Approximately how many items are in your collection?
3. How serious of a collector do you consider yourself to be
   (1 = casual, 5 = extremely serious)
4. Which statement best applies to you?
   - I do not catalogue my collection and do not want to catalogue it in the future
   - I do not catalogue my collection, but I am looking for a method of cataloguing my collection
   - I catalogue my collection but am dissatisfied with the method I use
   - I catalogue my collection and am satisfied with the method I use
5. Do you currently use software to catalogue your collection?
   - 5.a. If they answered yes to question 5:
     What were the positives and negatives of using the software?
   - 5.b. If they answered no to question 5:
     Would you ever consider using software to catalogue your collection?
6. Does your collection contain niche or rare items?
7. Do you prefer data to be presented to you visually or through text?
8. Which is the most important to you when using software?
   - Ease of use
   - User freedom / customisability
   - Lots of features
9. Do you believe it would be necessary to password protect collections?
10. How experienced are you with using computers?
    (1 = no experience, 5 = advanced)
11. How experienced are you with using data management software (MySQL, Excel)?
    (1 = no experience, 5 = advanced)
12. Do you share your collection on social media?
13. Do you loan items from your collection to others?
    - 13.a. If they answered yes to question 13:
      Have you ever had incidents where items haven't been returned?
14. Would you benefit from being able to back up your collections to the cloud?
15. What are the basic specifications of your computer?
16. What key features would you expect a cataloguing software to have?

Questions 1, 2 and 3 are designed to gauge what kind of collectors are answering the questionnaire.

Questions 4 and 5 establish if they have already used collecting software or any other cataloguing method. Question 4 will allow me to determine if respondents catalogue their collection or not. Question 5 will then narrow the focus to specifically asking if they use computer software to catalogue their collection.

The follow up question 5.a. will be asked if the user responds “yes” to question 5. It shifts the focus to the user’s requirements by asking respondents to evaluate the software they use.

The follow up question 5.b. will be asked if the user responds “no” to question 5. This question assesses the likelihood of people using software to catalogue their collection.

From question 6 onwards, the focus of the questions is on the actual software and potential features. Question 6 is based on the complaints I read about Collectorz not having enough provision for niche and rare items in its online database, so I need to gauge how many people collect rare items and add features accordingly.

Questions 7 and 8 investigate what qualities users prioritise when choosing software and how they like data to be presented. These questions are designed to maximise the ergonomics of my software.

Question 9 will help me in deciding whether to include a security/authentication system in the application. Some people are more concerned with security than others, so the results of this question will help me ascertain if users feel this would be a necessary feature or if it would be a useless gimmick.

Questions 10 and 11 will be used to determine how capable and experienced the average user is when using computers and data management software. The results from this question will help me consider the user’s level of skill when developing my software.

I included question 12 because I was thinking of including the option to share collections to social media, after seeing several social media pages dedicated to showcasing people’s collections. The answers to this collection will inform me if this feature is something the majority of collectors would find useful.

Question 13 and the follow-up question 13.a. are designed to identify how many collectors would find a loan-management system within the software useful.

I included question 14 because I was considering allowing users to back up their collections to their Google Drive. The answers to this question will inform me whether this would be a useful feature or not.

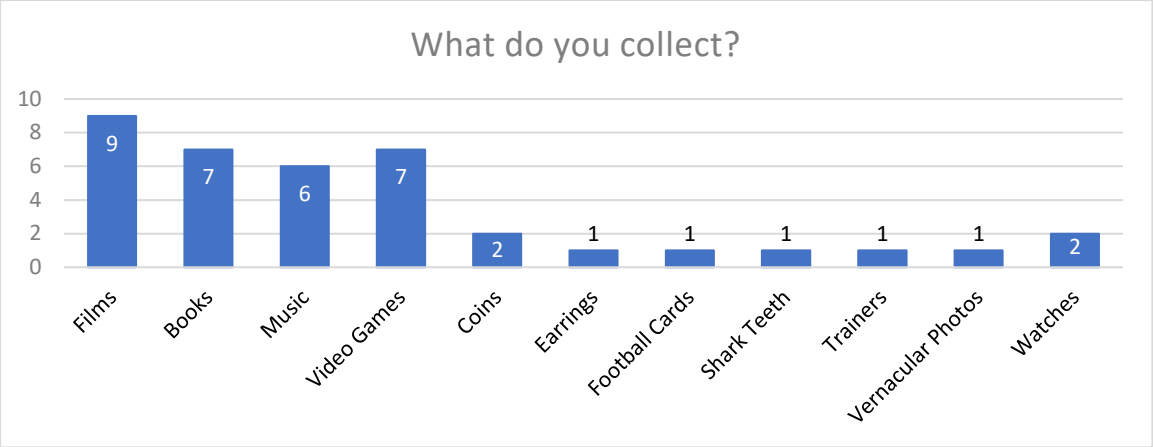
Question 15 will help me when specifying the minimum software and hardware requirements for my project.

Question 16 has been deliberately placed at the end of the questionnaire to allow the user time to think over their answer to it. The answers to this question will allow me to take on board end-user recommendations and potential features that they would like to see in the application.

Results

38 people responded to my survey. The results are presented below.

Question 1

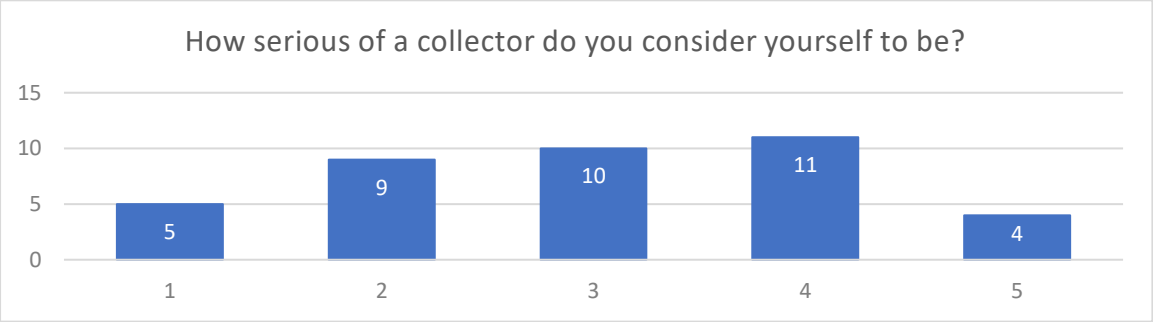


Question 2

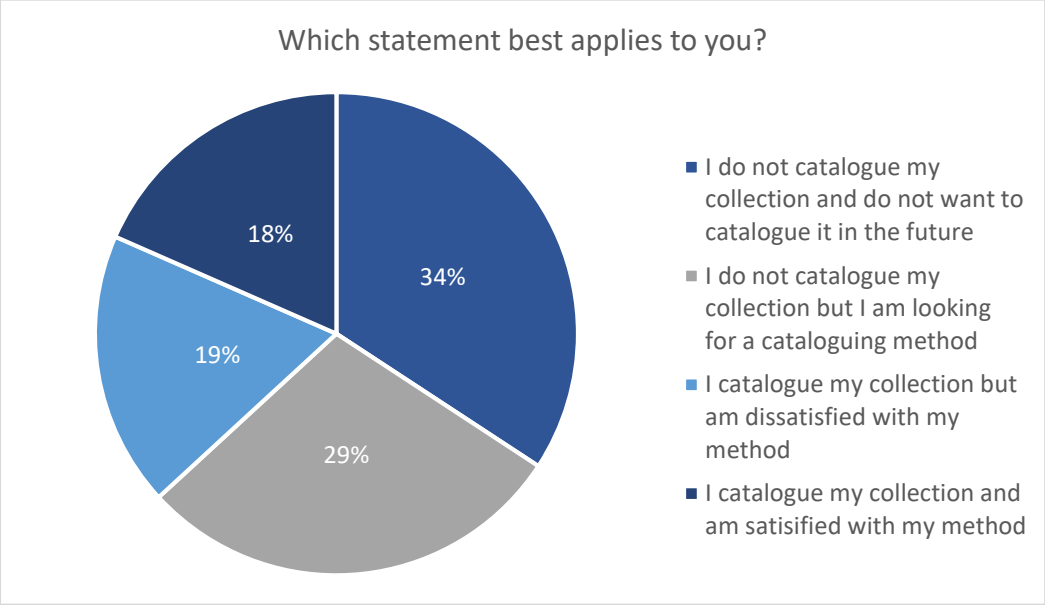
Approximately how many items are in your collection?

5	[1]	80	[3]
8	[1]	89	[1]
9	[1]	90	[1]
14	[1]	100	[2]
17	[1]	110	[1]
20	[1]	120	[1]
25	[2]	150	[1]
29	[1]	250	[1]
35	[1]	320	[1]
40	[1]	350	[1]
43	[1]	500	[2]
50	[5]	573	[1]
53	[1]	600	[1]
70	[2]	3500	[1]

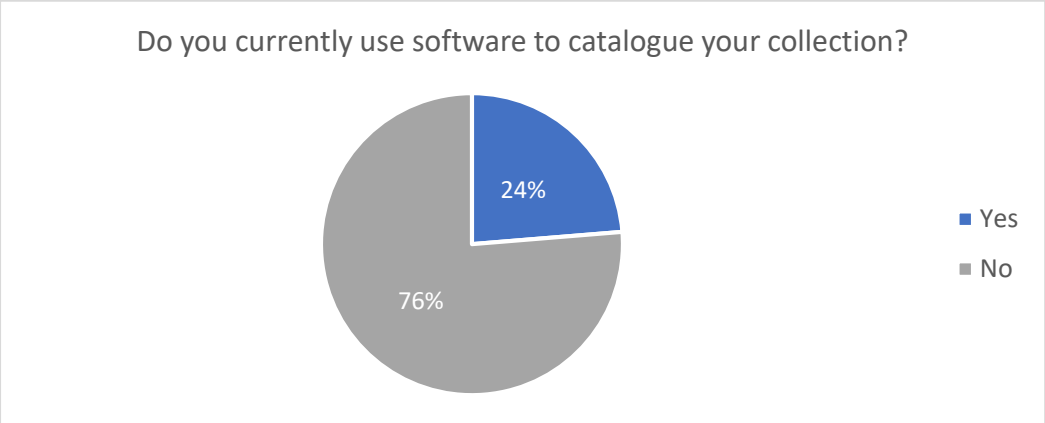
Question 3



Question 4



Question 5

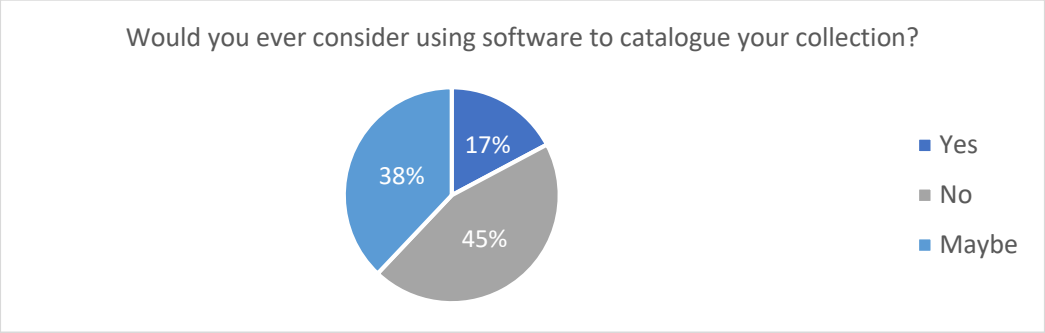


Question 5.a.

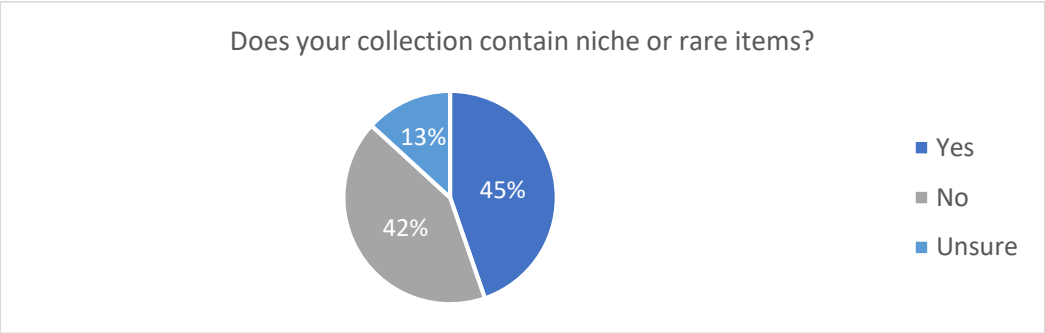
What were the positives and negatives of using the software?

- "Items are sorted into categories which made navigating my collection really easy. It is also pretty customisable – for example I can change which buttons appeared on the toolbar"
- "Very boring to look at but it was fairly efficient and easy to use"
- "It had a very powerful database, and I was never limited by how many items I could add"
- "The user interface could be worked on, I found it very cumbersome and outdated"
- "It has a clunky UI, but it is extremely customisable
- "Positive – It has an advanced search feature which helps me easily find items; Negative – it takes a lot of time to register items"
- "I can easily add new books I've purchased and split them into categories"
- "It was expensive, but the database was very robust, and the app rarely crashed"
- "I was pretty disappointed when I found out that I could only add items individually rather than having a bulk-upload feature. However, it has a mobile phone app and I use their online servers to access my stuff on different devices"

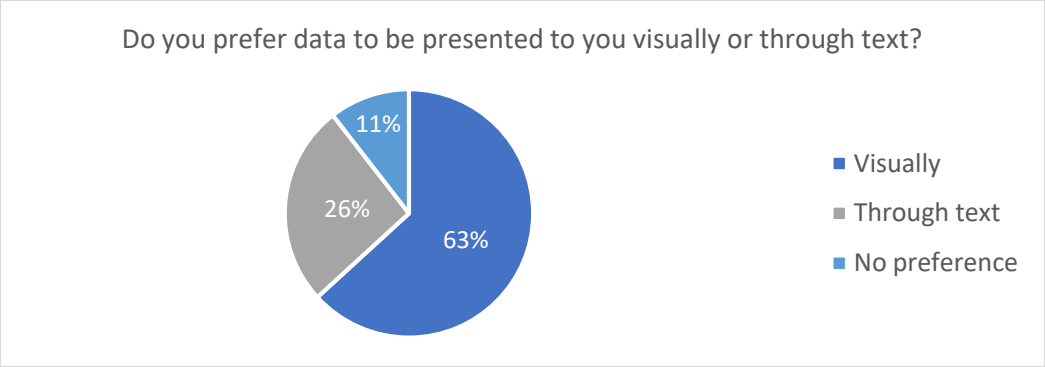
Question 5.b.



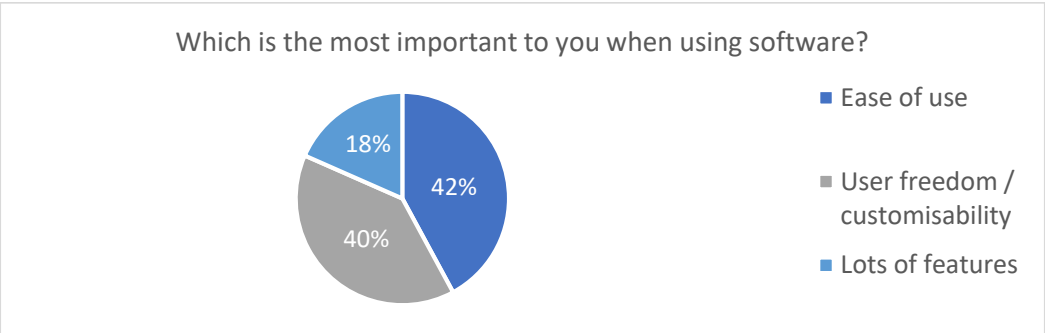
Question 6



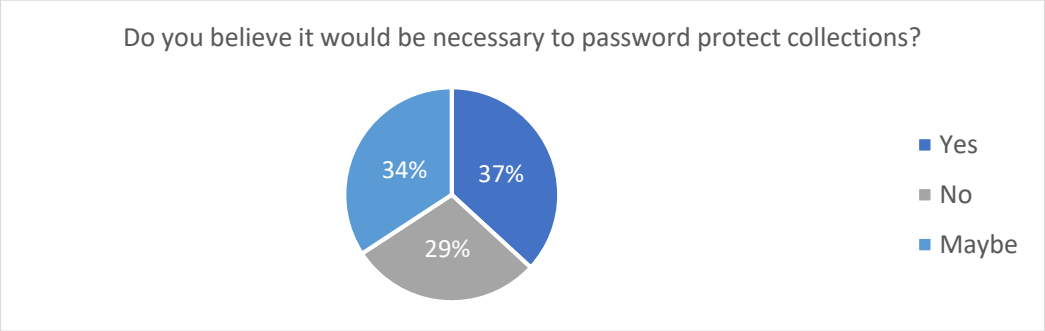
Question 7



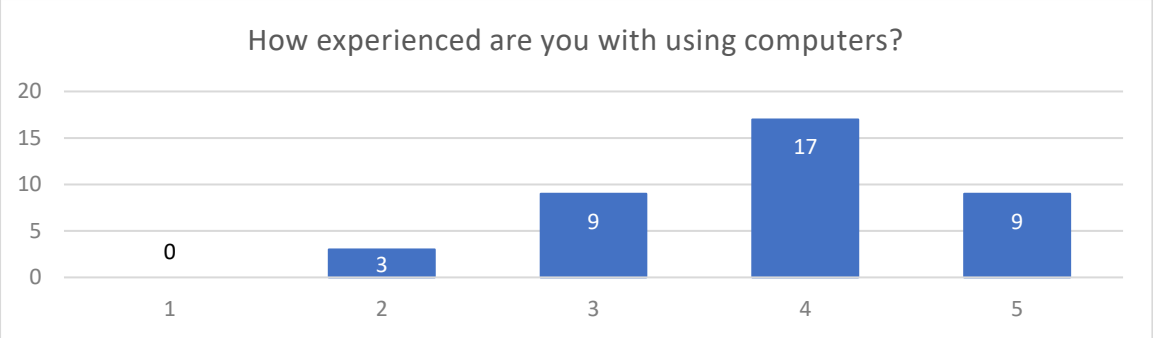
Question 8



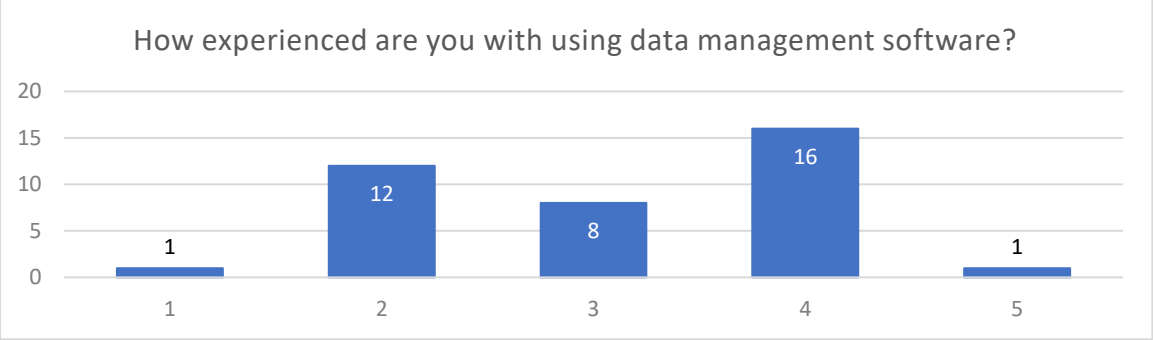
Question 9



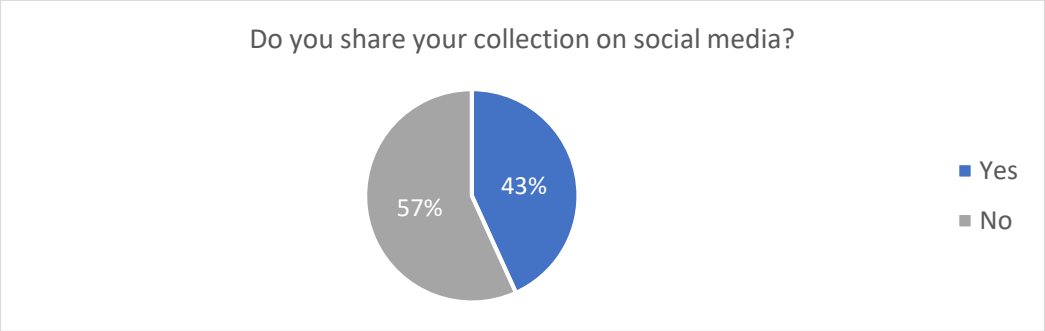
Question 10



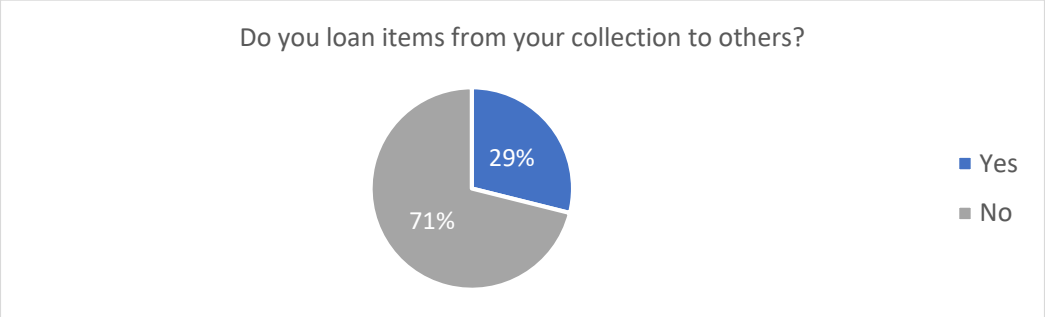
Question 11



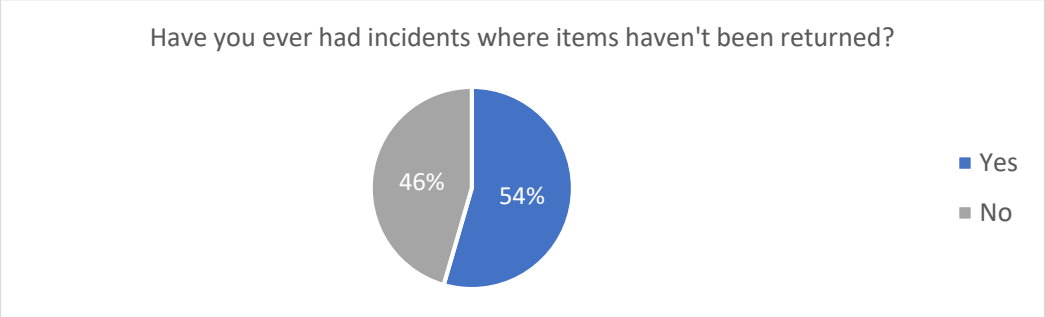
Question 12



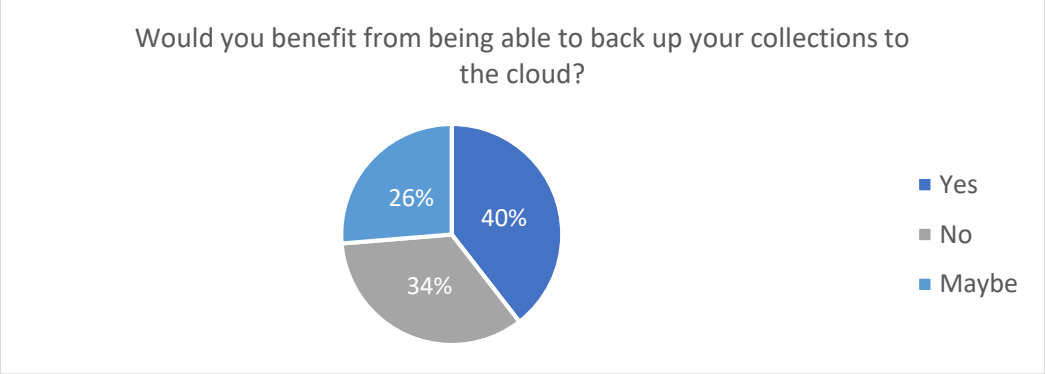
Question 13



Question 13.a.



Question 14



Question 15

What are the basic specifications of your computer?

Dell Inspiron 15 5000	MacBook Pro 2011
Lenovo ThinkPad :460	Intel core i5; 2x2.3GHz; 8GB RAM; x64
MacBook Air 2015	HP Pavilion x360
MacBook running MacOS High Sierra	MacBook Air 2017
Microsoft Surface Go	HP EliteBook 820
MacBook Pro	MacBook Air
Intel i3 core	iMac 2017
HP Pavilion	1.6 GHz Dual-Core Intel Core i5
Core i5	Windows PC – AMD A8-3800, 8GB RAM
Windows7 – 4GB RAM – 2.5GHz processor	Windows 10, HP Laptop
Acer Chromebook CB514-1H	iPad Pro
Eluktronics Pro Gaming Laptop	MacBook Air 2019
Mac Pro 128GB RAM	Alienware G5 15

Question 16

What key features would you expect a cataloguing software to have?

- "A way to search collections"
- "I'm unsure"
- "You could let users mark rare items with a special tag"
- "Advanced querying would be essential. Not necessarily full SQL, but some kind of query builder at least. Ability to access by mobile app would be pretty useful too, and to easily add items with the mobile app. The ability to archive and save eBay links and then query those would be very useful"
- "Items should be represented with thumbnails/icons"
- "Attractive user interface"
- "A way to easily search for items that are in a catalogue"
- "Easy to use"
- "Alphabetically ordered collections and the ability to sort items into categories"
- "Suggested items to add to your collection based on what's already in it"
- "I want to be able to store lots of details about items"
- "A way to update item information easily"
- "It has to be customisable; users can add data fields etc."
- "Some sort of notification for when loaned items are due in would be handy, I guess it could be done through email or text"
- "It should store eBay prices and let me track the value of items"
- "My requirements would be simple to use, a way to create graphs from your collection, and some way to automate adding items"
- "Lots of categories for items like director, producer, studio etc."
- "Simple, efficient, good visual design, no intrusive ads, shareable on social media, local or cloud backups"
- "Customisability, flexibility, label printing and report making would be important for me"
- "You should maximise customisability"
- "The ability to filter items based on conditions"
- "Some sort of barcode scanner to help speed up adding objects"
- "Being able to sort items by different categories"
- "Share to Facebook feature"
- "Should be available to use on both computer and phone"
- "I'd like different ways to sort items (alphabetical, chronological)"
- "Maybe the ability to scan my book's barcode into it so it enters easily"
- "Ease of use is paramount, and I need lots of space in the database"
- "Being able to export collections to Word and print them out would be really useful"
- "I'd personally need to be able to store the market values of items because I refurbish and resell items"
- "I've used Calibre in the past to manage my eBooks and I was impressed with how visual and intuitive it was, so I think you should bear that in mind when developing your app"
- "A way to keep track of items when I loan them out to my friends and the ability to sync between different devices like with the Amazon kindle app"
- "The ability to categorize items. I use a Chromebook so some sort of web platform would be required"
- "Advanced search options would help with larger collections"
- "Password protection and ability to share collections with friends"

- “Different ways to sort the catalogue and I’d like to be able to enter lots of information on each item”
- “User friendly design”
- “From personal experience, many of these apps don’t allow me to add enough detail when adding items to my catalogue, so I’ve always wanted the ability to store notes for each item so the user can store extra info alongside each item”

Analysis

The answers to question 1 show that the four most common types of collections are books, films, music and video games. In lieu of this, I will aim my solution to book, film, music and video game collectors so as to maximize its appeal. The average (mean) collection size was 219 which I would consider to be large enough to warrant some form of cataloguing. Within the data set, there was a large amount of variation from this average, with some collections being as small as 5 items and as large as 3,500. Question 3 was the final question concerning the respondent’s collecting habits. Most respondents rated themselves 4 out of 5 in terms of how serious they took collecting. I should therefore consider that most potential end-users will be firmly dedicated to their hobby when coding my solution.

The results of questions 4, 5, and 5.b. all prove that there is demand within my target market for cataloguing software. In addition, the fact that only 24% of respondents said that they currently use software to catalogue their collection in question 5 shows that the current solutions may not be attracting the majority of my target demographic, potentially for the reasons identified in the answers to question 5.a.

Respondents were asked question 5.a. if they answered “yes” to question 5. A consistent positive that many respondents identified was the ability to sort collections into categories. This is a vital feature that I must include in my solution. A negative that kept popping up was that many respondents felt the user interfaces of the software they used were “clunky”, “cumbersome” and “outdated”. This was also an issue that I personally identified when investigating existing solutions earlier on. My software needs to have a well-designed user interface that is pleasing to look at and makes the user experience as enjoyable as possible. One respondent pointed out that they would have liked their software to have a “bulk-upload feature” in order to make adding items less time consuming. This would not be a hard feature to implement, however, in advance, I need to consider how this feature would work best in order to make it as efficient as possible.

Question 6 had a fairly even split between those who did not collect rare items and those who did. My solution will have provision for collectors of rare items. This will come in the form of a special tag that users can apply to rare items, as suggested in one of the answers to question 16: “You could let users mark rare items with a special tag”.

The majority of responses to question 7 said that they preferred data to be presented to them visually. As a result, items will need to be displayed onscreen through their thumbnails or, as a compromise, items could be presented in a table through text and when users select an item its thumbnail could be displayed in the sidebar.

In question 8 my survey participants overwhelmingly preferred customisability and ease of use over an abundance of features. As a result of this, when coding my solution, I will need to prioritise these qualities. This will mean that every feature I choose to include should be well thought out and well designed, as opposed to throwing in lots of features at the expense of quality and usability.

Only 29% of answers to question 9 were “no”, so my solution will need an authentication system.

In both questions 10 and 11, most respondents answered 4 out of 5 in terms of their proficiency with computers and data management software. This means that I am free to include advanced features without having to worry too much about technophobic users.

Question 12 saw an almost even split between respondents who shared their collections on social media and those who didn't. As a result, I will consider adding this function to my solution, although I will first have to get to grips with the Facebook API.

Question 13 shows that most collectors don't loan items to friends. However, among those who did loan their items, 54% of respondents had incidents where items had not been returned. A loan management system would be extremely useful for those users. One response to question 16, which I thought was interesting, was that loan reminders "could be done through email or text". If I am to include an authentication system, I will already require user's emails, so it would be good to use email notifications.

40% of respondents answered "yes" to question 14 and 26% answered "maybe", so there is definitely demand for the ability to backup collections to the cloud. Implementation of this would be fairly straightforward using MySQL dumps and the Google Drive API.

Question 15 will be discussed in the Software and Hardware Requirements section.

Question 16 gave me a wide range of qualitative responses for interpretation. Because these results are qualitative, I have made an active effort in my analysis to be as unbiased as possible when interpreting them. Most respondents mentioned their need to be able to sort collections into categories and filter items based on conditions. My solution will need to have comprehensive sorting and filtering functions. Other respondents stressed the need for advanced search features: "Advanced querying would be essential. Not necessarily full SQL, but some kind of query builder at least". I also liked the idea of storing the market value of items and tracking prices, this could be implemented using web-scraping. One respondent said that "Being able to export collections to Word and print them out would be really useful". This would be a fun feature to code using python's file handling features and would be a way to allow users to share their collections to social media. Quite a few respondents also expressed their support for a barcode scanning feature. This was the standout feature of Microsoft's BookCloud and would be a good way to speed up the process of adding items to a collection. I would probably ask users to input the ISBN code and then use web-scraping to search the barcode online. Most people emphasised the importance of being able to store lots of information about each item, so this is something I must consider when designing my backend databases.

Essential Features of the Proposed Solution

After having taken into consideration the results from my questionnaire, I have now come to a conclusion as to what key features my software should include. These key features will be the core pillars of my software and by identifying them at this stage in development I will be able to ensure that my programming remains focused on the problem I have identified.

Virtual Catalogues

The primary function of my software should be allowing users to store items in virtual catalogues. These items should be displayed on the application's main screen in a visually pleasing manner (similar to Microsoft BookCloud). The data will be stored in a MySQL database.

Advanced Searching and Sorting Features

Several respondents to my questionnaire stressed the need for this feature in order to help with managing large collections. This will be implemented in the form of a query builder that users can use as an alternative to the more basic search function that I will also include. Users will be able to sort their collection using a dropdown menu.

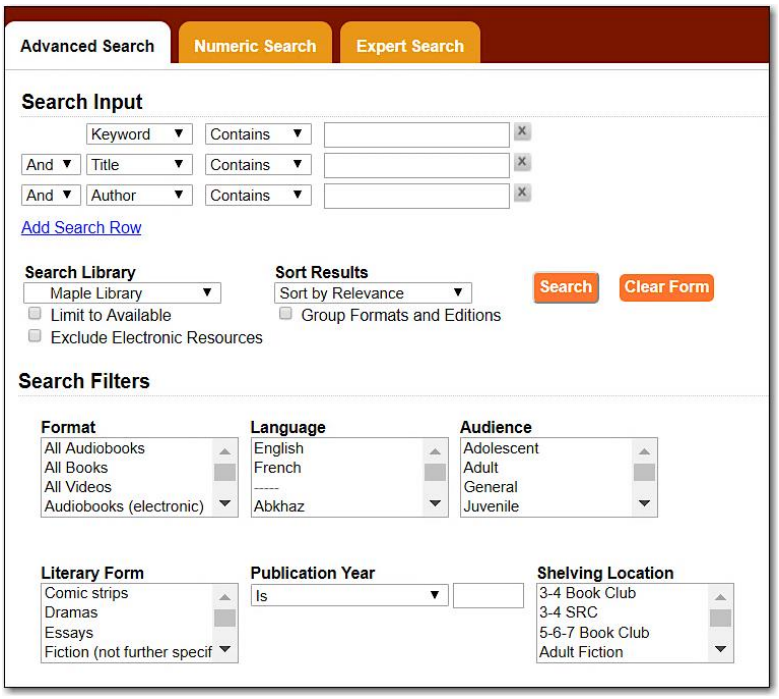


Figure 7: Example query builder

Loan Management System

This feature will enable users to select items in their catalogue that they would like to loan out, input the date it was loaned on and the date it is due in. The program will then notify them when the item is due in by sending an email or text message. I will send emails using the smtplib api. This feature will be a good way to differentiate my software from other alternatives as most collecting software don't have loan management systems.

Graphing

I would like to allow users to generate graphs from the data stored in their catalogues. These graphs could vary from pie charts presenting the proportions of publishers in a book collection to line graphs charting the growth in size of a collection over time. This would be programmed using matplotlib; a Python library specifically designed for embedding graphs into applications. These graphs could either be displayed in a separate window or in a sidebar in the main window (I will come to a final decision on this during the design stage). I will also give users the option to print these graphs out.

Login System

Only 29% of respondents to my questionnaire believed that it would not be necessary to password protect collections. A login system will secure all data stored in the application by preventing the malicious deletion or alteration of users' collections. A system using accounts will also mean that several different users can access their collections from the same device. I will need to encrypt usernames and passwords using a hashing algorithm. These hashes will then be stored in a MySQL database along with a username, email address and a foreign key to link the username and password to the database in which their collection is stored.

Exporting Collections
This feature would allow users to store backups of collections, print out paper copies of their collections and share their collections with others. If I were to allow users to upload collections to the cloud, I would have to use a company's API.

Barcode Searcher
This will allow users to input ISBN codes and automatically add the item to a collection along with the corresponding metadata. Lots of the collectors who answered my questionnaire felt that using software to catalogue collections could sometimes be time-consuming; I feel that this feature would help speed up the process.

Limitations

An important potential limitation of my solution will be efficiency. I need to make sure that my application is not slowed down when exchanging large amounts of data with my SQL database, otherwise performance for larger collections will suffer, diminishing the user experience. I will help mitigate this issue by ensuring that I plan my database's structure to be as efficient as possible in the design stage and, when programming, I may use concurrent execution to make several requests to the database simultaneously. Another limitation to consider, with regards to the barcode searcher and ratings finder, is the legality of web scraping. In the UK web scraping is legal, however copyrighted data that you scrape cannot be used for commercial purposes and collecting personal data could violate the Data Protection Act 1998. As long as I don't collect personal data and I stay away from copyrighted material, I shouldn't come across any issues. In terms of accessibility for disabled users, my software may not cater to all their needs. For example, the main inputs will come from the mouse and keyboard so users with a motor impairment may not be able to use the application. However, I will seek to use as many visual elements in my UI design such as thumbnails and graphs to help users with learning disabilities like ADHD and dyslexia use the software.

System Requirements

The minimum hardware specifications to run the software are:

- 576MB of RAM
- 1.6 GHz processor
- 16GB Hard drive space
- Certain peripherals such as a printer are required to use some features
- A stable internet connection

The minimum software requirements to run the software are:

- Windows, MacOS or Linux
- Python (The application will be written in Python, so python needs to be installed on the machine for it to run)
- MySQL will be used to build the databases because it is an industry standard
- MySQL Connector is a Python driver used to communicate with SQL servers
- PyQt will be used to program my Graphical User Interface as it is stable and generally does not suffer from cross-platform differences

I have settled on these requirements by considering the answers to question 15 of my questionnaire. I have also taken into account the minimum hardware requirements to run MySQL, and the operating systems on which PyQt will run.

Success Criteria

Necessary criteria are highlighted in green
Desirable criteria are highlighted in blue

	Criteria	Justification
1	Simple user interface that is intuitive, easy to use and aesthetically pleasing	If the UI is ugly or hard to use the software will not be worth using. It is imperative that I meet this criterion.
2	The software should be as optimised and lightweight as possible	This will ensure that the software will run well on low-spec computers and won't suffer from performance issues.
3	A table presenting items in a user's collection along with corresponding data	The table will have to be updated periodically with data from the backend database. A table will be used because it is a good way to organise, structure and present data.
4	The selected item's thumbnail and information should be displayed in the left sidebar	This will add to the visual aspect of the user interface, making it more visually pleasing and intuitive.
5	A login system to secure data and prevent unauthorised access and the ability to create new accounts	When the application is opened, users will be greeted by a login screen. If they get their password wrong, an error message will be displayed. Users will have the option to add or remove accounts.
6	The ability to create your own collections with user specified column names	This will allow users to create their own collections for any type of collection they want
7	The option to add and delete items from collections	This function is the core feature of the software. It should be an efficient and user-friendly process.
8	The option to delete duplicate items	This will help with the management of larger collections. This should be implemented as a button in the tool bar.
9	The option to edit the data for items already stored in a collection	This means that it won't be necessary to delete and add an item again if the user makes a mistake when inputting data.
10	A simple search function for quickly searching for an item	This function is intended to be used for quick, basic searches where the user does not require precise results. It will be implemented as a search bar.
11	An advanced search function for creating more complicated queries	A secondary window will be displayed allowing users to build queries to narrow the focus of their searches.
12	The option to filter/sort collections	The main window will have a sidebar on the right-hand side that will contain a tree widget breaking the collection up into different categories.
13	The ability to order collections in different ways (e.g., alphabetical, chronological, etc.)	There will be a dropdown menu allowing users to select the way they want to order their collection. Database queries will then be altered accordingly.
14	An error message should be displayed if a connection can't be made with the SQL server	The message will be displayed in the form of a popup window. This will mean that the user will

		know the specific issue that is present instead of having the app just crash with no explanation.
15	The option to mark rare or niche items with a special tag	This was a feature suggested by one of the respondents to my questionnaire. It will make rare items stand out.
16	The ability to store the market value of items	This criterion is meant for collectors who sell items for a profit as it gives them an easy way to store the price of items.
17	The option to store item ratings taken from online sources	These ratings will be taken from review aggregators like IMDB, Metacritic and Goodreads.
18	The ability to generate graphical breakdowns of collections	This will add to the visual aspect of the application and is another way to present data alongside the table.
19	The option to export graphs as images	Users will want to be able to save copies of graphs to their computers. There will be a file dialogue when the user presses the print button.
20	The option to export collections to a text file	This will allow users to print collections out, share them with friends or store local backups.
21	The ability to back collections up to a Google Drive account	The ability to back collections up will act as a disaster recovery policy in case data stored locally is corrupted or deleted. This will be implemented using the Google Drive API.
22	A loan management system	A second window will be opened. The user will be able to select an item to be loaned, input the date it was loaned and input the due date.
23	Generate email notifications when a loaned item is due	This will ensure that users are alerted when items are due. There should also be an option to turn notifications off in the settings. Emails will be sent using the smtplib api.
24	The ability to search for and add items using their ISBN codes (barcodes)	The purpose of this feature is to give users an option for adding items that is quicker than manually adding them. The program will search the barcode in an online database and then collect relevant information about the item using web scraping.
25	A settings menu	This will allow users to customise certain aspects of the software to suit their own preferences.
26	Keyboard shortcuts assigned to different functions	This will help power users maximise efficiency.
27	Data should be stored in an efficient, concise and well-designed database	Good database design will improve the efficiency of data transactions, improving the overall efficiency of the application

Measuring Success

At the end of my project, I will need to provide evidence that my software has met the above success criteria. For most criteria I will use screenshots or short video clips showcasing the software in action. Where relevant, I will also supply a screenshot of the corresponding code. For certain core features, such as the login system, I also need to supply evidence that I have extensively tested the functions. I will use end-user testimonies to show that I have fulfilled the more subjective requirements, such as those concerning the visual appeal of the user interface.

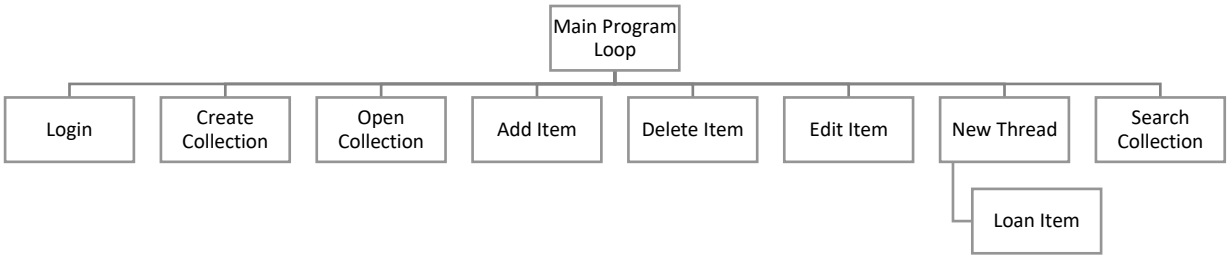
Section 2: Design

Now that the success criteria have been established, I can begin designing the software. The frontend of my program will be implemented using PyQt - a Python binding for the Qt library. PyQt has the benefit of being much more stable than other GUI libraries like Tkinter, and its cross-platform capabilities will mean that I won't have to worry about compatibility issues. The aesthetic appearance of my PyQt windows will be styled using an external CSS stylesheet. Separating the stylesheet from the core code will makes it easy to change the window's appearance without affecting its functionality. The backend of my application will use a MySQL database as MySQL is free, open-source and fairly lightweight in comparison to other Database Management Systems. The Python code will interface with the MySQL database using the MySQL-connector-python driver. I will use the PyCharm IDE in order to make use of its comprehensive debugging tools and library integration.

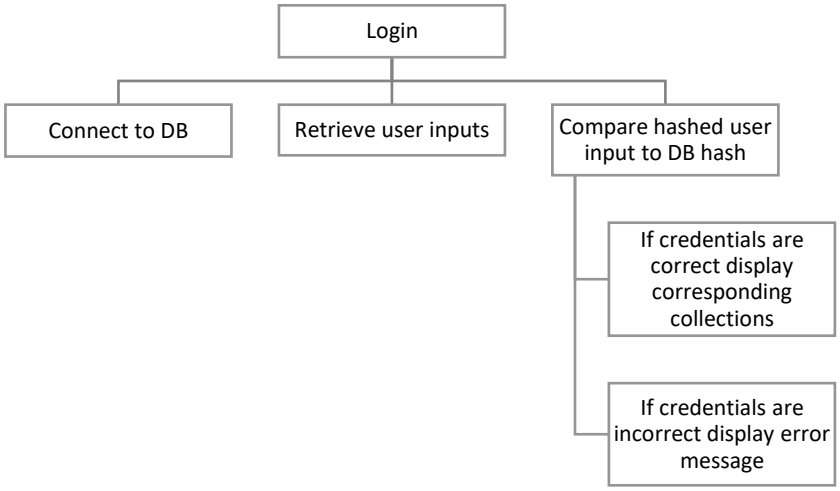
I will adopt an object-oriented approach to coding this program as PyQt requires each window to be implemented as an object with every element within that window (buttons, text boxes, labels, etc.) acting as an attribute of that window object.

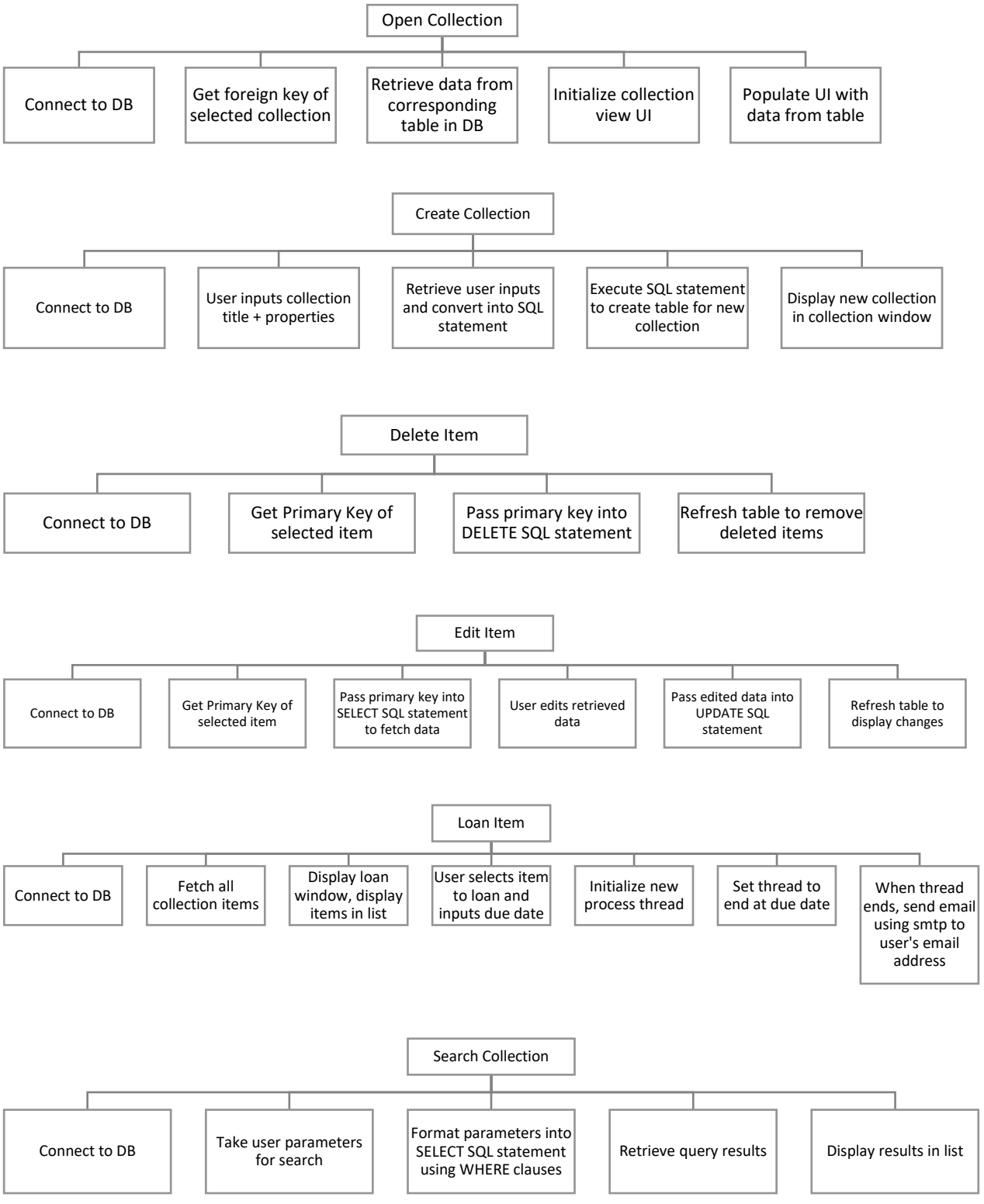
Top-Down Design

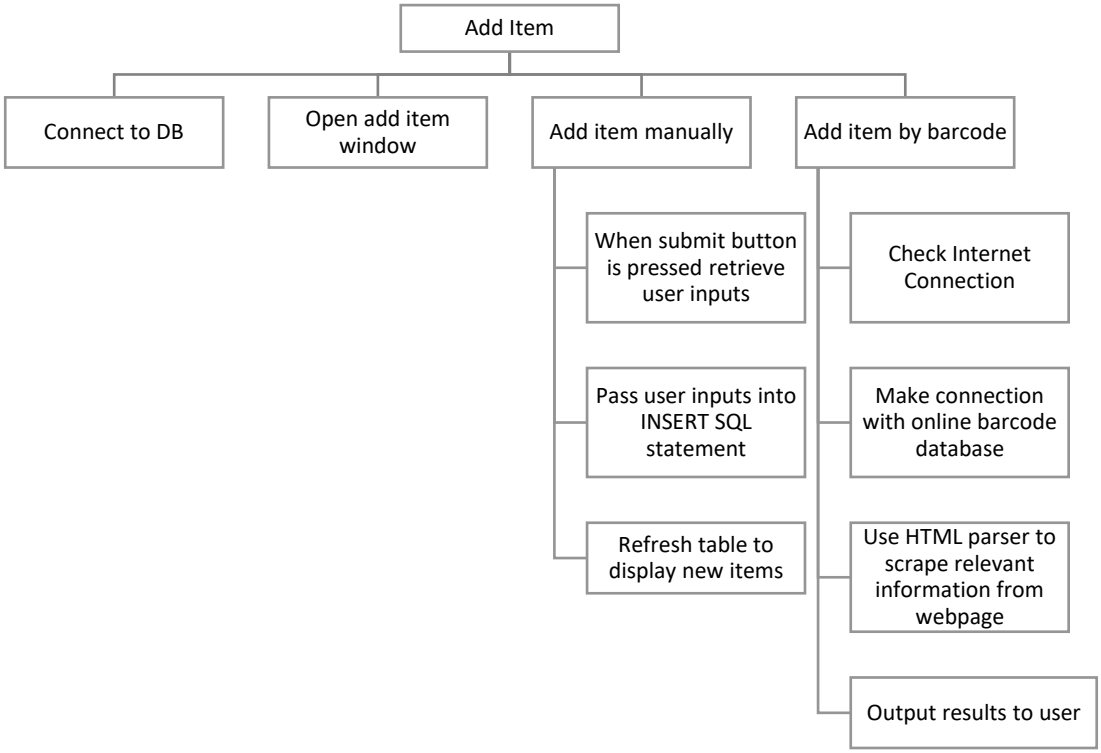
Decomposition is an important stage of the design process because splitting a complex problem into smaller subproblems makes it much easier to solve. In addition, the modular nature of the top-down design makes it easy to test the program and allows me to reuse modules when appropriate. I have decomposed each core component of my program into a series of simple, manageable steps presented below:



Each element of the above breakdown is further decomposed below:




Open Collection
Connect to DB
Get foreign key of selected collection
Retrieve data from corresponding table in DB
Initialize collection view UI
Populate UI with data from table
Create Collection
Connect to DB
User inputs collection title + properties
Retrieve user inputs and convert into SQL statement
Execute SQL statement to create table for new collection
Display new collection in collection window
Delete Item
Connect to DB
Get Primary Key of selected item
Pass primary key into DELETE SQL statement
Refresh table to remove deleted items
Edit Item
Connect to DB
Get Primary Key of selected item
Pass primary key into SELECT SQL statement to fetch data
User edits retrieved data
Pass edited data into UPDATE SQL statement
Refresh table to display changes
Loan Item
Connect to DB
Fetch all collection items
Display loan window, display items in list
User selects item to loan and inputs due date
Initialize new process thread
Set thread to end at due date
When thread ends, send email using smtp to user's email address
Search Collection
Connect to DB
Take user parameters for search
Format parameters into SELECT SQL statement using WHERE clauses
Retrieve query results
Display results in list


Add Item
Connect to DB
Open add item window
Add item manually
Add item by barcode
When submit button is pressed retrieve user inputs
Pass user inputs into INSERT SQL statement
Refresh table to display new items
Check Internet Connection
Make connection with online barcode database
Use HTML parser to scrape relevant information from webpage
Output results to user

Inputs, Processes and Outputs

Most of the inputs will be implemented as PyQt buttons, combo boxes, sliders and menus. These will be linked to their corresponding functions using PyQt's inbuilt signals and slots system.

Input	Process	Output
Login button	The application will run a validation function to check if the username and password match the hashed values stored in the SQL database	Launch main window and display the collections that correspond with the username and password
Create new collection button	A new window will be opened, allowing the user to enter the new collection title and which properties they would like to use. A SQL table for the collection should then be created	A new collection will be created and will be displayed in the collections window
Add/Delete/Edit buttons	PyQt will open a top-level window allowing the user to carry out the desired function	A new window will be opened
Sort collection menu	A dropdown menu will open with all the options for sorting the collection, once the user has selected their choice the collection view will refresh and display the collection in the selected order	The collection will be displayed in the order selected by the user
Loan item button	PyQt will open a new window and ask the user to select the item they would like to loan out and input the date they would like it returned	The user will receive an email when the item is due
Generate graphs button	Graphs will be generated from the data stored in the SQL database	Graphs will be displayed in a sidebar in the main window
Barcode search video feed	Each frame will be processed in order to search for a barcode. If a barcode is detected, the web scraping function will be called to search the barcode in an online directory	The results of the search will be returned to the user and they will have the option to add that item to their collection
Search /advanced search	Convert user query into SQL command and retrieve results	Display search results in user in table

User Interface Prototypes

Login Screen

Upon launching the application, this should be the first window that is displayed. The login screen contains a welcome message, text input boxes for the username and password, and a login button. I have also added an image on the right-hand side to increase the visual appeal of the window

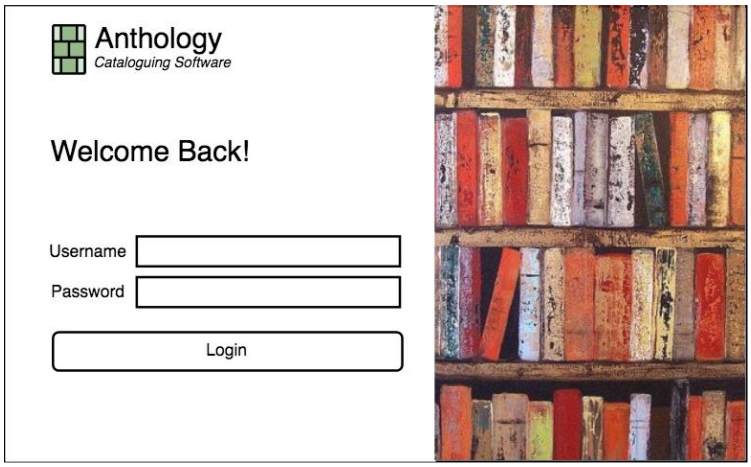


Figure 8

If the user inputs incorrect credentials, an error window will be displayed. The error window will tell the user how many attempts they have left before they are temporarily locked out of their account. If the user presses the "OK" button they will be returned to the login screen to try again.

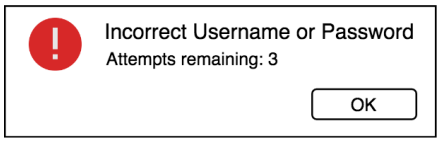


Figure 9

Display Collections

If the user successfully logs in, a window containing all the collections associated with their account will be created. Each collection will be displayed as a large, brightly coloured button in order to make them easy to find. The colour scheme I have chosen to use consists of a white background, black text and two shades of green for widgets, highlights and accents (Dark green: #96BA8A, Light green: #D5E8D4). The first button gives the user the option to create a new collection and, when pressed, opens up the Add Collection dialogue window. The user will also be able to add and delete collections using the file menu.

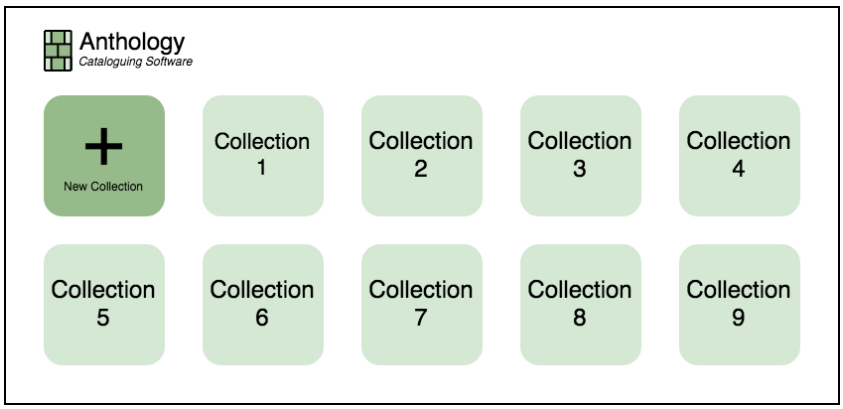


Figure 10

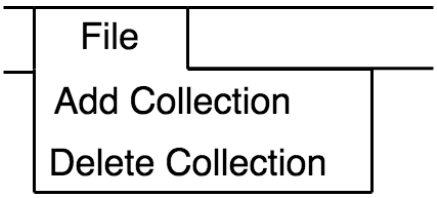


Figure 11

Add Collection

This window is displayed when the user presses the New Collection button in the Collections window. The first text box is for the title of the collection (e.g., books, stamps etc.) The user can then add properties by clicking the "Add New Property" button. Each property will become an attribute in the SQL table. The user is given a text box to input the name of the property. They can also use the dropdown menu to select the data type for that property. In order to keep things simple, there will be two options: number or text. When the "Create Collection" button is pressed, a new SQL table will be created for the collection; the Add Collection window will close; and the Collections window will be refreshed in order to display the new collection.

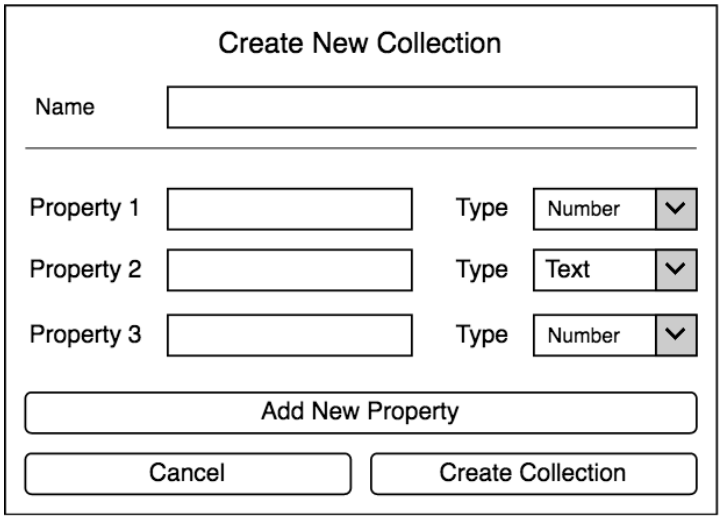


Figure 12

Open Collection (Main Window)

When the user double clicks on a collection, this window will be displayed. The top of the window contains the toolbar. Each button in the toolbar will consist of text and an icon and all the toolbar functions will also be accessible from the File Menu. Whenever a function in the toolbar is performed, the list will have to be updated. Below the tool bar is the regular search bar. This search bar is intended for quick, broad searches as opposed to the Advanced Search feature which is accessible via the toolbar. To the right of the search bar is a dropdown menu which will allow the user to specify how they would like their collection to be ordered. Collections can be ordered in alphabetical and date order and these orders can be ascending or descending. The focal point of the window is the list containing all the items. Each item in the list must be selectable as this allows the user to select the items they would like to perform operations on. The right-hand sidebar will display the thumbnail and data for the selected item. The left sidebar will contain categories that collection items can be sorted into. At the bottom of the window statistics, such as collection size, will be displayed.

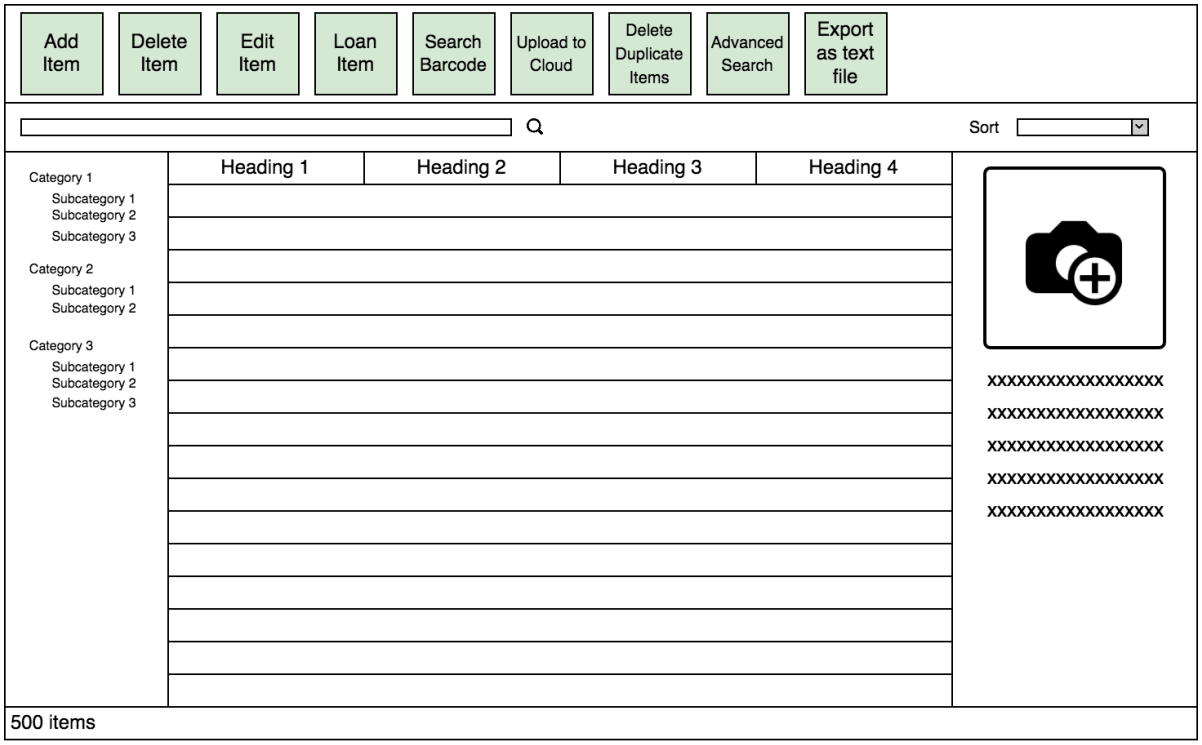


Figure 13

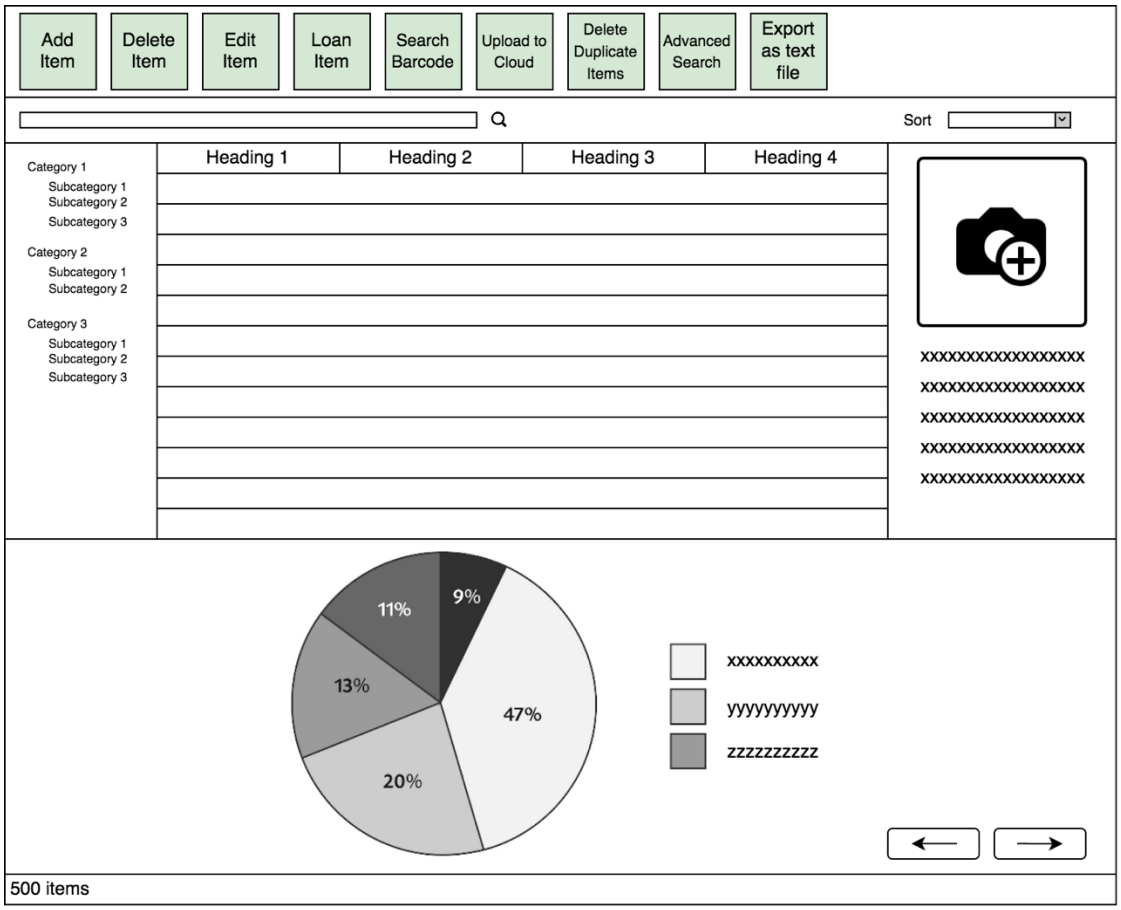


Figure 14

If the user chooses to generate graphs based on their collection, they will be displayed along the bottom of the window. The arrow buttons at the bottom allow the user to navigate between the graphs.

Add Item

This dialogue window allows the user to add a new item to their collection. They will be able to input values for each record within the collection table. The rare property will be a checkbox. When the "Upload Thumbnail" button is pressed, a file dialogue will be opened, allowing the user to select an image to use as a thumbnail

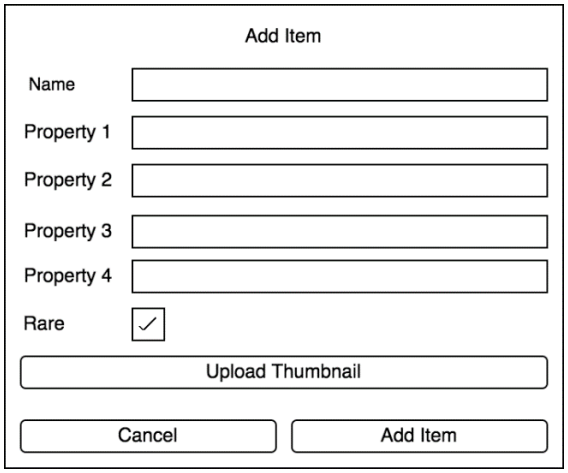


Figure 15

Advanced Search

The user can input the parameters of their search. Once the "Search Collection" button is pressed, the database will be queried, and the results will be displayed in the list in the main window. This window will have to be dynamically generated as each collection will have a different set of properties

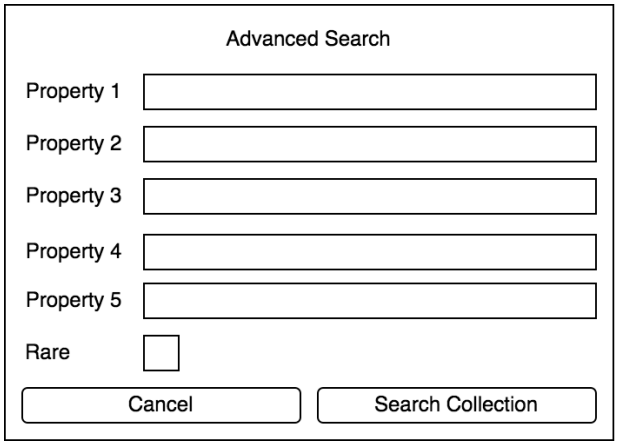


Figure 16

Loan Item

At the top of the window, a list will display all the collection items so that the user can select one to loan out. The user will then have to input the due date and time, so the program knows when to send the email notification.

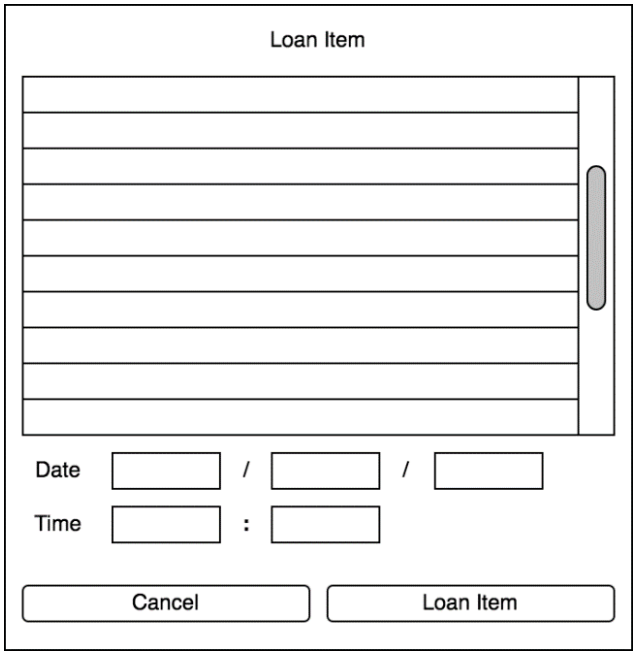


Figure 17

Search Barcode

The barcode search function will be implemented using OpenCV. I will use OpenCV to capture a live video feed which will be outputted onto a window. The user will then have to position the barcode into the frame. Each OpenCV frame will be passed through the pyzbar module which will search the frame for barcodes. Once a barcode has been recognized, it will be highlighted with a red rectangle and its numerical ISBN code will be passed into a webscraping function. The webscraping function will then search a number of online barcode databases for that ISBN code. If the web search is successful, a new Add Item window will be created, and the data returned by the search will be passed into that window. If the search fails, an error message will be outputted and the live video feed will resume, allowing the user to try again.

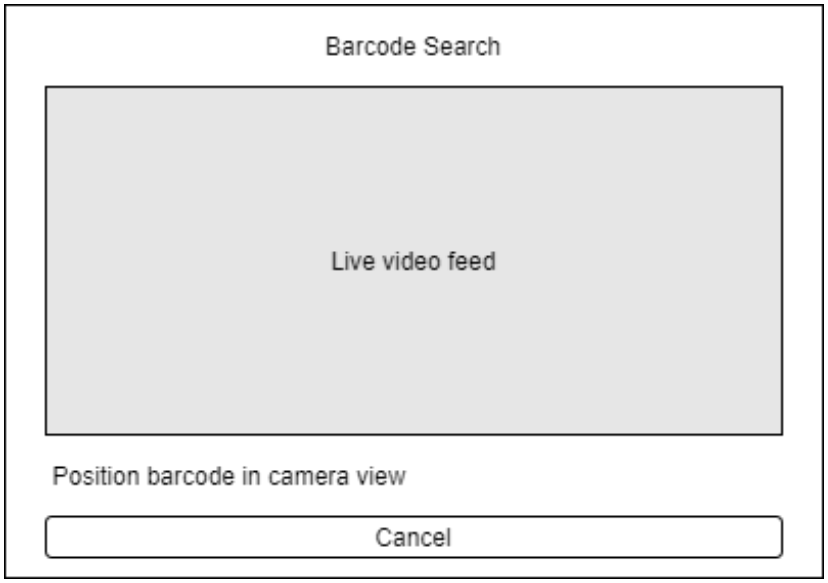


Figure 18

End-User Feedback on Prototypes

I sent the following email to three end-users along with the prototype pictures:

> *Dear [recipient]*
> *I'm currently in the design phase of my project. I've attached some prototype user interface designs for you to have a look at and give me some feedback on. I would particularly like to hear your feedback regarding how easy to use and accessible these designs appear to be. I could also do with your thoughts on the colour scheme I settled on.*
> *Looking forward to hearing your feedback!*
> *Sami*

Johnathan's reply:

> *Hi Sami,*
> *Really liking the designs so far. I think these blueprint designs look very sleek and modern. It would be nice if you used more icons, especially on buttons, for old users like myself who don't know their way around a computer very well. I really like how you've used big, colourful buttons to represent collections in the collection window and I think the colours you chose complement each other very well.*
> *Keep safe,*
> *John*

Julia's reply:

> *Hello,*
> *It's great to see some designs, I can now visualize what this app will look like. I think the colours, layouts and designs are great. I would like to mention however that I don't want to be overwhelmed by windows when using an app so I hope you keep it as simple as possible.*
> *Julia*

Sarah's reply:

> *Hi,*
> *These look very pretty. I think it's great that we can create our own collections, when I was using Collectorz there wasn't that same degree of user freedom. Something to consider, in your designs it isn't clear that there's a difference between the advanced search and the regular search, maybe you could set some text to appear when you hover over each button telling the user what they do.*
> *Sorry for not getting back to you sooner,*
> *Sarah*

Overall, the end-user feedback has been positive. It's great to hear that they think the designs have achieved a good balance between aesthetics and practicality. I will definitely take into account Sarah's suggestion of adding tooltips for widgets. I have also acted on Jonathan's suggestions by creating the icon set below (all icons are taken from open-source art repositories):

Accessibility and Usability

In order to make my application as accessible to elderly and visually impaired users, I have tried my best to include as many visual stimuli in my designs as possible. For example, in line with Jonathan's stakeholder input, I will include icons for all the toolbar buttons. Important buttons will be highlighted in green to make them easy to locate on the screen. I have also tried to incorporate as many graphical elements into the main window design, such as the graph view and the use of item thumbnails in the sidebar. In order to make navigating the application and all its different functions as easy as possible, I will program tooltips for the buttons in the application so users can learn each button's purpose by hovering their cursor over it.

Julia mentioned that she had concerns with being able to navigate all the different windows without becoming confused. I have addressed these concerns by giving each dialogue window a cancel button which will return the user to the main window when pressed. Furthermore, each window will have a descriptive title at the top to inform the user of the window's purpose.

For the user, the most complex part of the program will be creating collections. I have tried to abstract this process as much as possible by devising a system whereby all the user needs to do is add a property and then input its name and data type. Rather than confusing the user with a plethora of different data types for each property, I have decided to limit their choice to numbers or text. The program will then automate the actual creation of the table. Making sure the program was simple to use was one of the top priorities in my success criteria, so it is imperative that I implement this feature well.

PyQt applications are cross-platform, meaning that my application should be able to run on Linux, Windows and MacOS without having to make any changes to the code. As a result, my software will be accessible to users regardless of the operating system they use. Because PyQt apps run as native applications, my software will also be compatible with native accessibility tools such as MacOS' built in text-to-speech features or Windows' speech recognition tool. This helps maximise usability for disabled users.

Menu Flow Diagram

The Menu flow diagram below outlines how the different windows/layouts will interact with each other and when each window/layout will be displayed

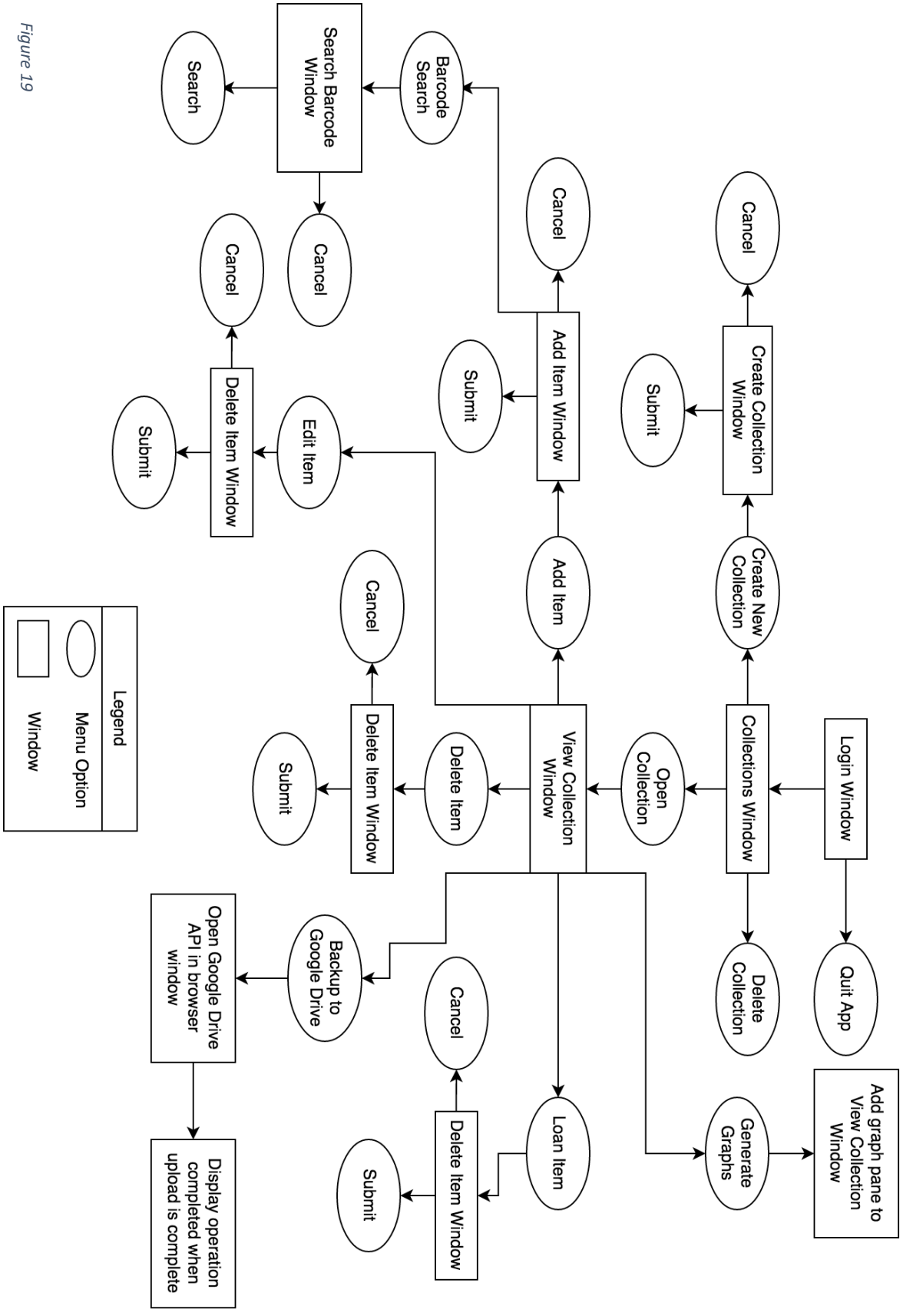


Figure 19

Database Design

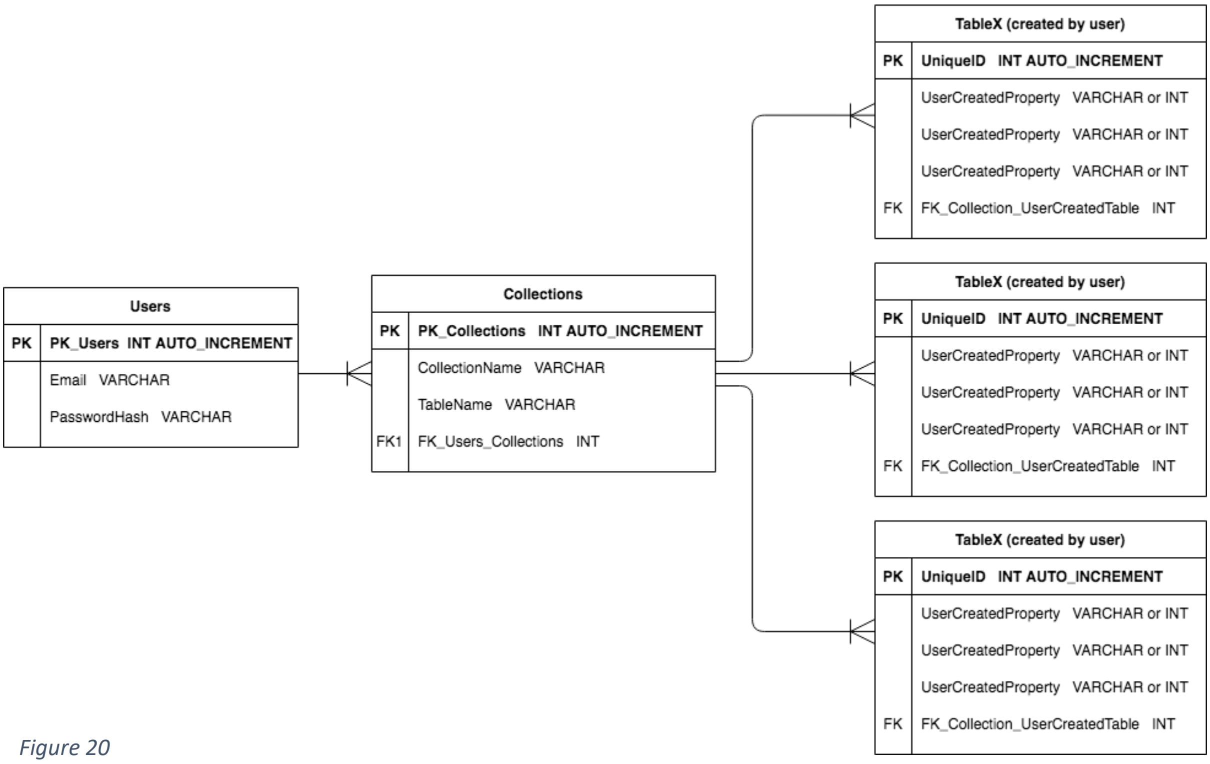


Figure 20

The entity relationship diagram inserted above provides a visualisation of the data structures I will be using in my SQL backend. The “Users” table contains login credentials and will be used when validating user logins. The foreign key of the “Collections” table references the primary key of the “Users” table in a one-to-many relationship as a single account can be associated with many collections. The “Collections” table acts as a directory of all the collections linked to each account. It will be used in the Collections window (figure 10) when showing the user their collections. “TableX” represent the tables that will be formed when the user creates a collection. For example, for a book collection the program would create a table with the attributes “title”, “author” and “publisher”. The foreign key in each user created table will reference the primary key of the “Collections” table. The column “TableName” in the “Collections” table stores the name of the table that corresponds with that collection. This is a way of overcoming the difficulties of having dynamic tables created by the user. Each collection has a different set of attributes so they have to be stored in separate tables. This means that the single “Collections” table has to reference multiple other tables, which can’t be done using primary and foreign keys. Instead, by storing the TableName alongside the Collection, I can just use the name in my select statements. For example, if I had a collection entitled “Books” with the corresponding TableName “Table3”, to retrieve the contents of the table I would first retrieve the TableName from the Collections table, then I would execute the statement “SELECT * FROM Table3”. To preserve referential integrity, I must ensure that each table name is unique. Therefore, table names will take the format TableX, where X is the primary key of the corresponding record in the Collections table.

All my primary keys follow the syntax “PK_<TableName>” and all my foreign keys follow the syntax "FK_<TargetTable>_<SourceTable>". Using these naming conventions will help enhance the clarity of my code and will help me easily pinpoint which attributes are being used as keys.

In order to maintain referential integrity, each foreign key will have the FOREIGN KEY constraint applied to it. I will also use an ON DELETE statement to configure the course of action taken when a foreign key's parent record is deleted.

In order to preserve domain integrity, attributes will have the NOT NULL constraint when necessary and primary and foreign keys will have the UNIQUE constraint. I will also use check constraints where necessary in order to validate data being inputted into the database.

Data Dictionary

The main variables used in the program are listed below:

Variable Name	Data Type	Description
ActiveUser	Integer	When a user logs in, this global variable will be set to the primary key of their account record in the Users table. This primary key corresponds with the foreign key for collections linked to this account in the Collections table. Whenever the program needs to know which user is logged in, this is the variable they will reference
ActiveCollection	String	When a collection is opened, this global variable will be set to the string stored in the TableName field in the Collections table that corresponds with the collection that has been opened. This variable will be used when querying the table that the user has opened
QueryResult	2D array containing Tuples	Whenever the program queries the database, the results of the query will be retrieved using the cursor.fetchall() command and stored under this variable. If I need to store multiple query results, I will name the variables QueryResult2, QueryResult3 etc.
DataForInsertion	Tuple	Data has to be passed into MySQL as tuples. So, before I can insert data, I will have to feed it into this tuple variable
GraphValues	Array of integers	This array will store all the values that will be passed into the graph object in order to generate the collection graphs. The values will have to be calculated beforehand, using data from the SQL database
GraphIndex	Integer	Stores the index representing which graph the user is currently viewing. This variable will be used with the two arrow buttons to allow the user to switch between graphs; when the user presses the forwards button, the index is increased by one and the next graph is displayed, the opposite happens when the user presses the back button
BinaryThumbnail	String (BLOB)	Images must be converted into raw binary data before they can be inserted into a SQL database. When the user selects a thumbnail, they would like to add to an item, the program will have to convert it into binary data and store it under this variable before passing the binary into the DataForInsertion tuple
SelectedRows	Array	Whenever the user wants to edit or delete items, they have selected in the item list, the id of the selected items will be stored in this array. The values of this array can then be passed into the relevant function. A variant of this variable will also be used in the loan item window to store the item the user chooses to loan out
DueDate	DateTime	The DateTime datatype is made available by importing the datetime module in Python. This variable stores the due date and time that the user inputs when loaning out an item. This variable will then be passed into a QThread class which will send the email at the specified time
UserProperties	2D array	Each matrix within the array will store a property the user has created in the Create Collection window and its corresponding data

		type. The variable will then be passed into a function which will create the user defined table
ParsingRequest	String	When webscraping, the result of a HTML request has to be stored as a variable in order for it to then be parsed using BeautifulSoup
Barcode	Integer	When the user scans a barcode, the image parsing function should return a 12- or 13-digit ISBN code which can then be used to perform a web search

The file outputs are listed below:

File	Path	Description
Backup.sql	Root/Temp	Before uploading a database back-up to the cloud, a local back-up has to be created using the mysqldump command line utility. This file will be stored in the Temp directory and will be deleted when the back-up operation is complete
Collection.txt	Root/Files	If the user chooses to export their collection to a text file, it will be stored in this file. The name of the file will be the name of the collection

Validation

All inputs into my system must be validated in order to prevent unexpected or erroneous data from crashing the system.

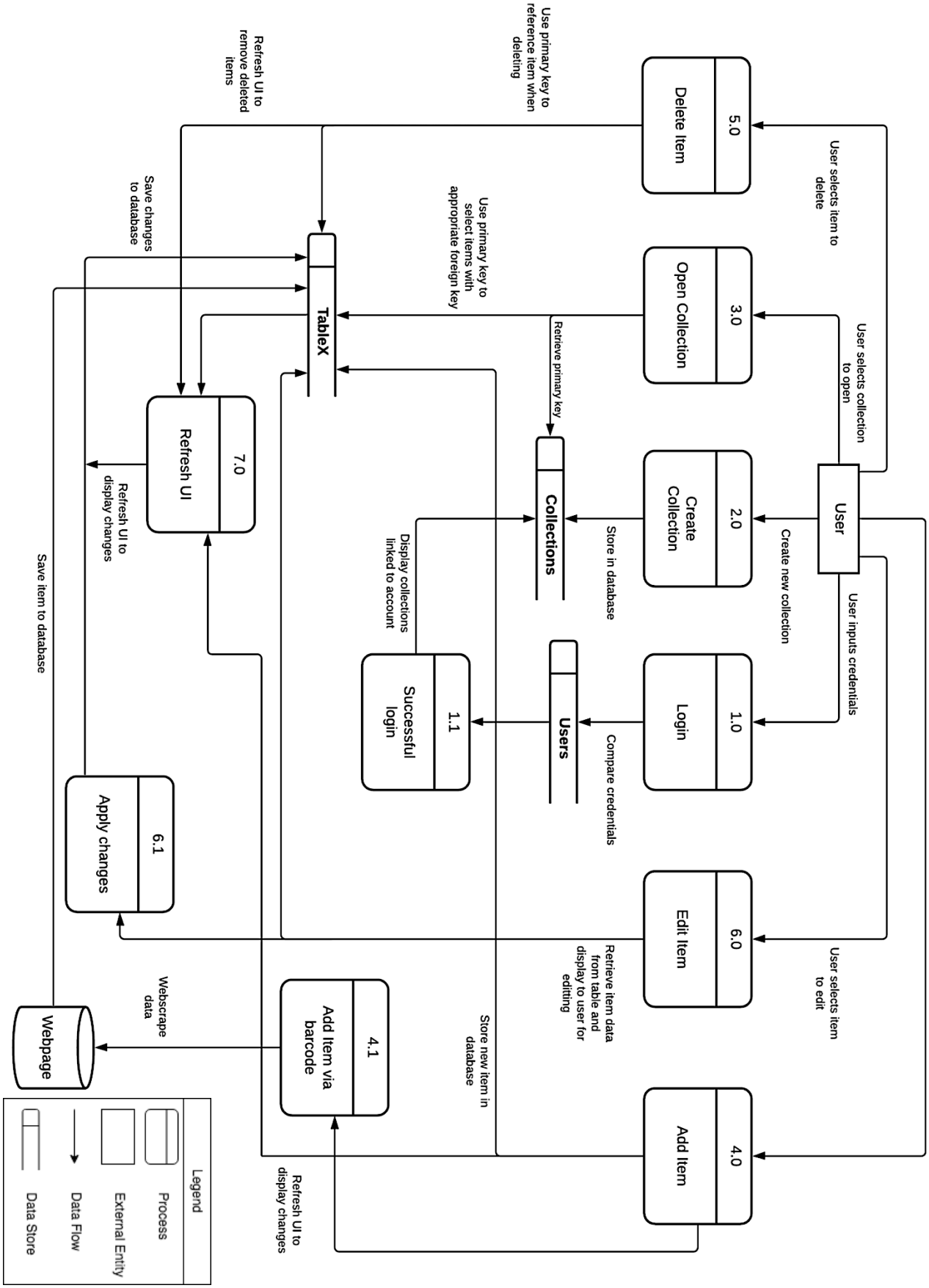
Most of the user input will come in the form of buttons, sliders, tables and combo boxes. These widgets are all PyQt classes and therefore should be stable and have inbuilt validation methods.

Text boxes require more work to validate. The first method of validation I will have to perform when the user inputs data into a text box will be a presence check to ensure that they have actually inputted something. If inputted data is going to be used in a SQL statement, I will have to strip the text of certain characters to protect against SQL injection attacks. In situations where only certain characters can be entered into a text box (e.g. the date and time input boxes in the loan item window should only accept integers), I will use the QValidator class to limit user input.

Another form of input will be the video feed for the barcode scanner. When pyzbar detects a barcode in the video feed, I will have to perform a length check to ensure that the ISBN code is 12 or 13 digits long, otherwise the barcode won't be valid.

Data Flow Diagram

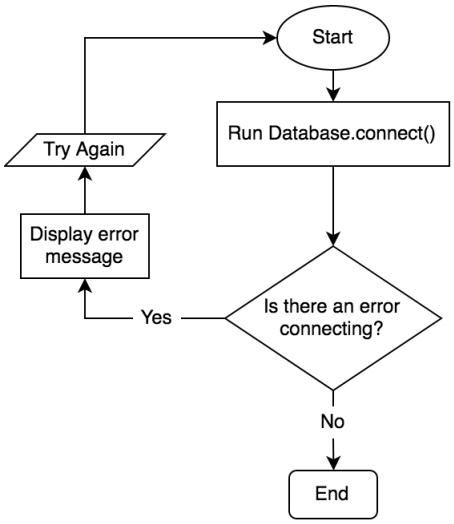
The data flow diagram below maps out how data will flow around my system and which processes will access which data stores. I have decided to use Gane and Sarson notation for my diagram.



Algorithms

Test Database Connection

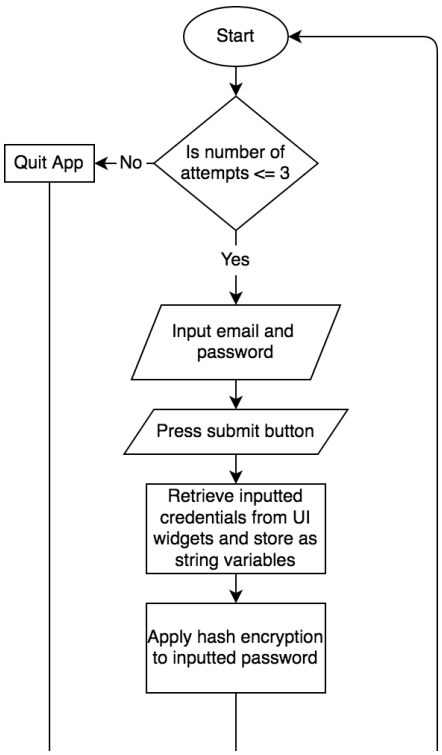
Upon start-up, the program needs to discern whether it can make a connection with the SQL database. Two separate threads will run on start-up (implemented via the QThread class). The first thread will display a loading screen and the second will perform the backend operations.




```
FUNCTION TestConnection():
   //TRY EXCEPT clause is used to see if there is
   an error when making connection
   TRY:
      MyDB = MySQL.connect()
      MyCursor = MyDB.cursor()
      OUTPUT LoginScreen
   EXCEPT:
      SHOW ConnectionErrorMessage
      IF TryAgainBTN.clicked() == TRUE:
         RUN TestConnection()
```

Login

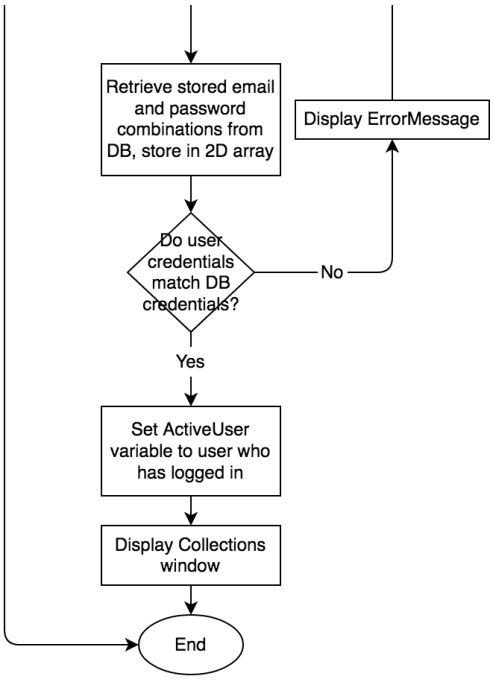
Once a connection has been established, the user will then have to input their credentials into a login form. Their hashed inputs will then be compared to the hashed credentials stored in the Users table. Credentials need to be encrypted using hashing to prevent malicious attackers from gaining login credentials. If the user inputs the correct credentials, the Collections window will be opened with the collections that correspond to their account. If they are incorrect, an error message will be displayed informing them of how many passwords attempts they have left. The user is restricted to 3 attempts at trying their password. If the user exceeds these attempts, the application will close.




```
FUNCTION Login():
   //Attempts variable keeps track of user attempts
   Attempts = 0
   WHILE Attempts < 3:
      //User inputs Email and hash password
      Email = INPUT
      Password = INPUT
      HashedPassword = HASH(Password)

      //Retrieve credentials
      MyCursor.execute(SELECT * FROM Users)
      MyResult = MyCursor.fetchall()

      //Iterate through DB credentials and check
      against user inputs
      FOR x in range (0, len(MyResult)):
         IF MyResult[x][0] == Email AND
         MyResult[x][1] == HashedPassword:
            GLOBAL ActiveUser = Email
            OUTPUT Collections Window
```

Continued on next page

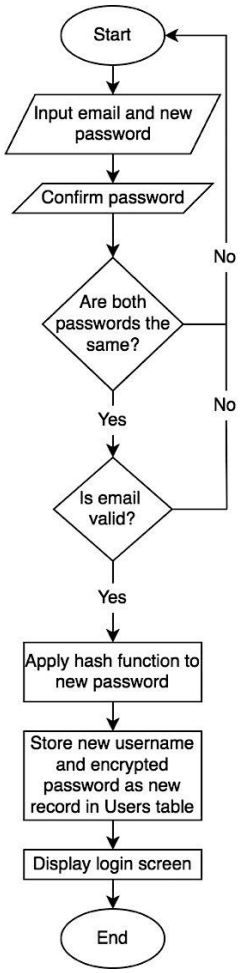
Continued from next page




```
        ELSE:
            x += 1

        // If credentials are incorrect show error
        message
        OUTPUT LoginErrorMessage

    //WHILE loop breaks when Attempts exceed 3
    Quit()
ENDFUNCTION
```

Create Account

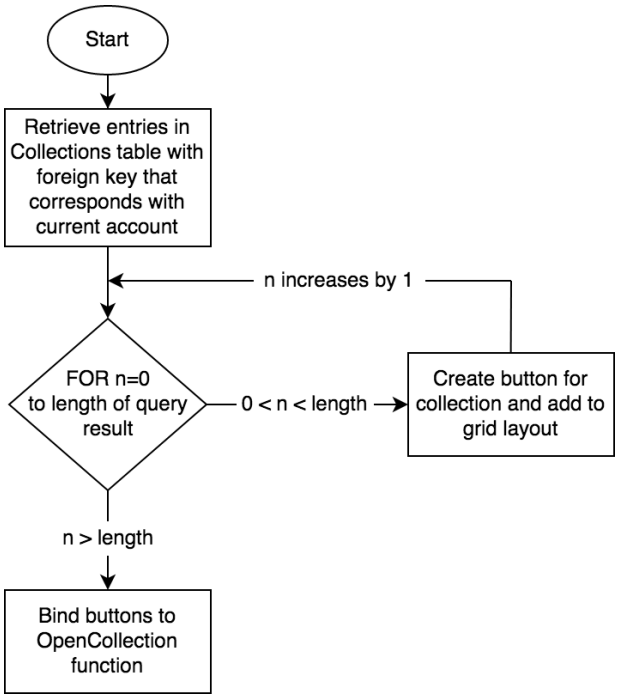
This function allows users to create a new account. The new account credentials will be stored in the Users table. Passwords will be hashed before they are stored in the SQL table.




```
FUNCTION CreateAccount():
    //User inputs credentials for new account
    NewEmail = INPUT
    NewPassword = INPUT
    ConfirmPassword = INPUT

    //Regex module returns true if email is valid and false if not
    VALIDATE NewEmail using regex

    //Check if both password inputs are the same
    IF NewPassword == ConfirmPassword and NewEmail is valid:
        //Encrypt new password before insertion into DB
        NewPasswordHash = HASH(NewPassword)
        MyCursor.execute(
                INSERT INTO Users VALUES(NewEmail, NewPasswordHash)
                )

        //Save any changes made to database
        MyDB.commit()
        //Return user to login screen
        OUTPUT LoginScreen

    //If password inputs are not the same, output error message and let
    user try again
    ELSE:
        OUTPUT ErrorMessage
ENDFUNCTION
```

Display Collections

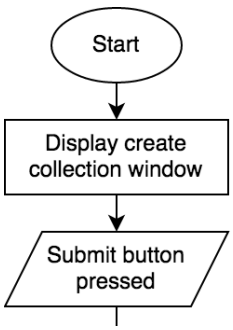
Once the user has logged in, the Collections window will be displayed. This window needs to be populated with buttons for each collection belonging to that account. PyQt uses layouts to arrange widgets, so these buttons will have to be arranged in a grid layout.




```
FUNCTION CollectionsWindow():
    //Select primary key for current account using global variable ActiveUser
    MyCursor.execute(
        SELECT PK_Users FROM Users
        WHERE Email = ActiveUser
    )
    ActivePK = MyDB.fetchall()

    //Select collections associated with open account
    MyCursor.execute(
        SELECT * FROM Collections WHERE
        FK_Users_Collections = ActivePK
    )
    ActiveCollections = MyCursor.fetchall()
    //Because there is a variable number of buttons, each button created in the for loop is added to this array and can then be referenced using its index
    CollectionButtons = []
    //Iterate through array and create a button for each collection within that array
    FOR x in range(0, len(ActiveCollections):
        CollectionButtons.append(QPushButton())
        //Set button text
        CollectionButtons[x].text = ActiveCollections[x]
        //Bind button to OpenCollection function
        CollectionButtons[x].clicked.connect(OpenCollection())
        //Add button to grid layout
        GridLayout.addWidget(CollectionButtons[x])
ENDFUNCTION
```

Create Collection

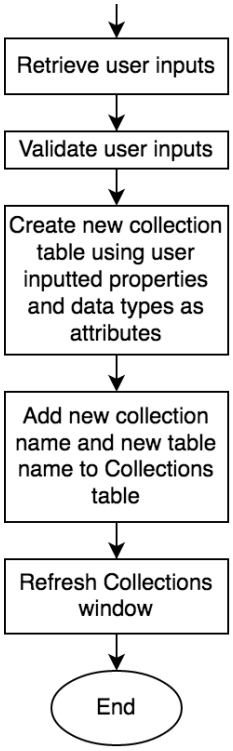
The create new collection function is called when the user presses the new collection button in the Collections window. It will display the Add Collection window. User inputs need to be validated. The user will construct a collection by giving it a title and giving it properties. The collection will then be created as a SQL table, with each user-defined property making up an attribute in the table. Certain properties, such as “Rating” will be automatically added to all collections.



Continued on next page

```
FUNCTION CreateCollection():
    //Show window for user to create collection in
    OUTPUT AddCollection Window
    //In the program these checks will be implemented using signals and slots
    IF AddPropertyBTN.clicked() == TRUE:
        Add new property textbox and datatype combobox
    IF SubmitBTN.clicked() == TRUE:
        RUN SubmitNewCollection()
```

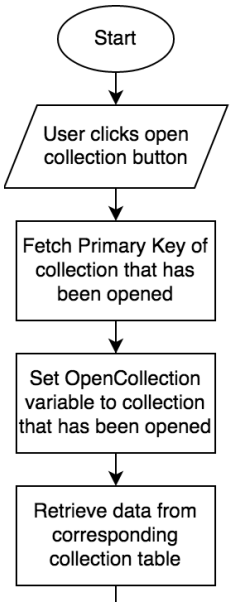
Continued from next page




```
FUNCTION SubmitNewCollection():
    //2D array stores pairs of property names and datatypes ready for
    SQL table creation
    NewProperties = [[]]
    //Iterate through all textboxes added to screen and copy textbox
    content + combobox contents into array
    FOR textboxes in Window:
        NewProperties.append(textboxes.text())
        NewProperties.append(combobox.text())
    //Store collection name as variable
    CollectionName = TitleTextBox.text()
    //Format NewProperties array contents into string for SQL statement
    S=””
    FOR x in range(0, len(NewProperties)):
        S.append(NewProperties[x][0] + “ ”)
        S.append(NewProperties[x][1].upper() + “, ”)
    //Create new table, pass CollectionName into SQL statement as table
    name and pass S in as attributes and datatypes
    MyCursor.execute(
        “CREATE TABLE” + CollectionName + “(id INT PRIMARY KEY
        AUTO_INCREMENT, ” + S + “Rating INT)”
    )
    //Insert new collection as record in Collections Table, store
    collection name and new table name
    MyCursor.execute(
        INSERT INTO Collections VALUES (CollectionName, CollectionName)
    )
    //Refresh Collections window to display new collections
    RUN CollectionsWindow()
```

Open Collection

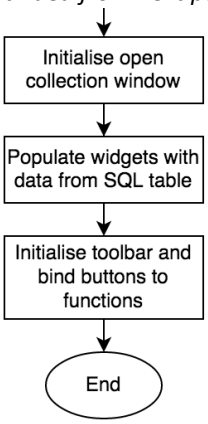
This function will be executed when a user double clicks on a collection in the Collections window. A new open collection window will be opened. All the widgets within that window will have to be initialised and populated with the data corresponding to the collection that has been opened.



Continued on next page

```
//OpenCollection coordinates the execution of other subroutines, it
takes the parameter ClickedButton
FUNCTION OpenCollection(ClickedButton):
    //Each button will have a value attached to it that corresponds with
    the primary key of the collection it represents in the Collections
    table
    SelectedPK = ClickedButton.PK

    //Select TableName entry in Collections table that corresponds with
    button pressed
    MyCursor.execute(
        “SELECT TableName FROM Collections WHERE PK_Collections = ” +
        SelectedPK
    )
    MyResult1 = MyCursor.fetchall()
    MyCursor.execute(
        “SELECT * FROM” + MyResult1[0][0]
    )
```

Continued from next page

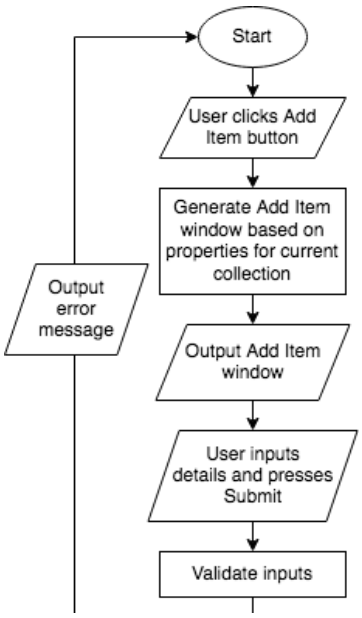



```
//MyResult2 stores all data from table
MyResult2 = MyCursor.fetchall()

MyCursor.execute(
    "SHOW COLUMNS FROM" + MyResult1[0][0]
)
//MyResult3 stores column names, to be used as headers
MyResult3 = MyCursor.fetchall()

//OUTPUT OpenCollectionWindow will initialise all the widgets in the
OpenCollection window
OUTPUT OpenCollectionWindow
RUN PopulateTable(MyResult2, MyResult3)
```

```
//PopulateTable function fills the table in the mainwindow with the collection data
FUNCTION PopulateTable(TableData, ColumnData):
   //First for loop sets the headers for the list
   FOR x in range(0, LEN(ColumnData):
      //TableList refers to the table outputting the collection in the open
      collection window
      TableList.HorizontalHeader[x].setText(ColumnData[x])

   //Because TableData is a 2D array, a nested for loop is needed to traverse it
   and assign each value to the appropriate cell in TableList
   FOR y in range(0, len(TableData)):
      FOR z in range(0, len(TableData[x])):
         TableList.setCellWidget(x, y, TableData[x][y])
```

Add Item

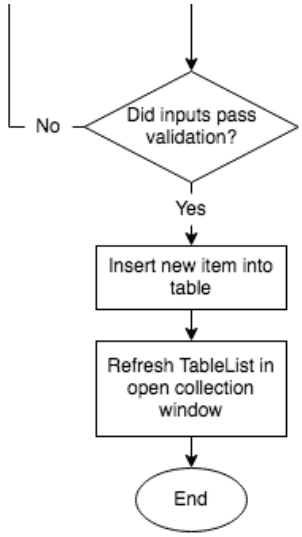
This function will be executed when the user presses the "Add Item" button in the toolbar of the Open Collection window. It will open a dialogue window in which the user can input the details for the new item. When the submit button is pressed, the collection will be added to the appropriate SQL table. All inputs must be validated to ensure they are of the correct datatype and to prevent against SQL injection attacks. Once the item has been added to the database, the function PopulateTable() will be executed in order to refresh the table list in the main window



Continued on next page

```
//This class generates the window and populates it with
widgets
CLASS ItemWindow():
   //ActiveTable is a global variable
   //SHOW FIELDS returns each column and its data type
   MyCursor.execute(
       "SHOW FIELDS FROM" + ActiveTable
   )
   MyResult1 = MyCursor.fetchall()
   //There will be a variable number of TextBoxes, so they
   will have to be stored in the TextBoxes array and
   referenced using their index
   TextBoxes = []
   //For each column, create a label and a text input box
   and add them to the window
   FOR x in range(0, LEN(MyResult1):
      AddItemWindow.addWidget(QLabel(MyResult1.Field))
      TextBoxes[x] = AddItemWindow.addWidget(QLineEdit())
```

Continued from next page

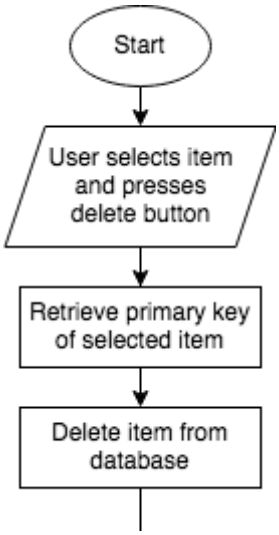



```
//If the data type for the current property is an
integer, we need to limit the inputs for that line edit
IF MyResult1.Type == INTEGER:
   //In PyQt validators are used to restrict the type of
   characters that can be inputted into a line edit
   TextBoxes[x].setValidator(Qt.OnlyInt)

//Bind submit button to SubmitItem function
SubmitBTN.clicked.connect(SubmitItem())
```

```
FUNCTION SubmitItem():
   DataForInsertion = []
   //Retrieve text stored in text boxes and add to array
   FOR x in range(0, LEN(TextBoxes)):
      DataForInsertion.append(TextBoxes[x].text())
   //Using string substitution makes the code less vulnerable to injection attacks
   SQLStatement = "INSERT INTO %s VALUES (" + "%s"*ColumnCount + ")"
   //Data must be passed into an INSERT statement as a tuple
   DataForInsertion = TUPLE(DataForInsertion)
   //Execute the SQL command with the values in DataForInsertion passed into it
   MyCursor.execute(SQLStatement, DataForInsertion)

   MyDB.commit()   //Save changes to database
   RUN PopulateTable()   //Refresh TableList
```

Delete Item

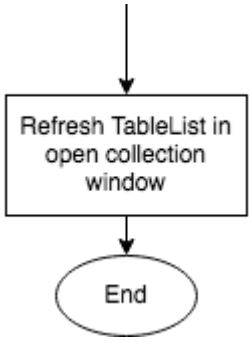
If the user selects an item and presses the delete button, that item needs to be deleted. This will be done by retrieving the primary key of the selected item and then executing a DELETE statement. The user may select multiple items to delete, so the function must work with a variable number of items. The user may also press the button without selecting any items, so there must be checks in place to deal with that occurrence.



Continued on next page

```
FUNCTION DeleteItem():
   SelectedRows = []
   SelectedIDs = []

   //Iterate through selected rows and add each row to the array
   FOR x in TableList.selectedRows():
      SelectedRows.append(x.row())

   //For each row in the array, find the corresponding primary key
   and add to second array
   FOR y in range(0, LEN(SelectedRows)):
      SelectedIDs.append(TableList.itemAt(SelectedRows[y], 0)

   //If the user has not selected any rows, end the function
   IF LEN(SelectedIDs) == 0:
      PASS
```

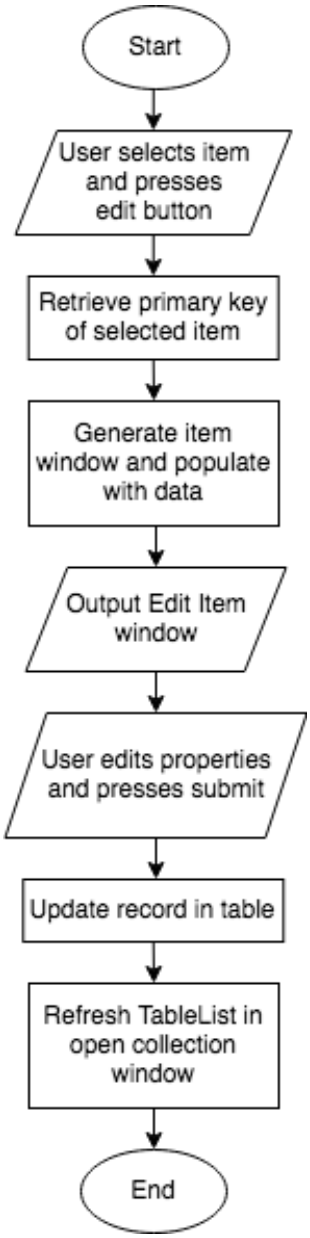
Continued from next page




```
    //Delete the entry corresponding to each ID within the array
    ELSE:
       FOR z in range(0, LEN(SelectedIDs):
          MyCursor.execute(
                "DELETE FROM" + ActiveTable + "WHERE ID = " +
                SelectedIDs[z]
          )

MyDB.commit()   //Save changes to database
RUN PopulateTable()   //Refresh TableList
```

Edit Item

This function will combine elements from the delete item and add item functions. The program will first have to retrieve the selected row and its ID. It will then fetch the data stored under that entry and generate an ItemWindow. Once that window has been generated, each text box needs to be populated with the corresponding data. The user can then make their changes and, once they are done, the SubmitItem function will be executed.

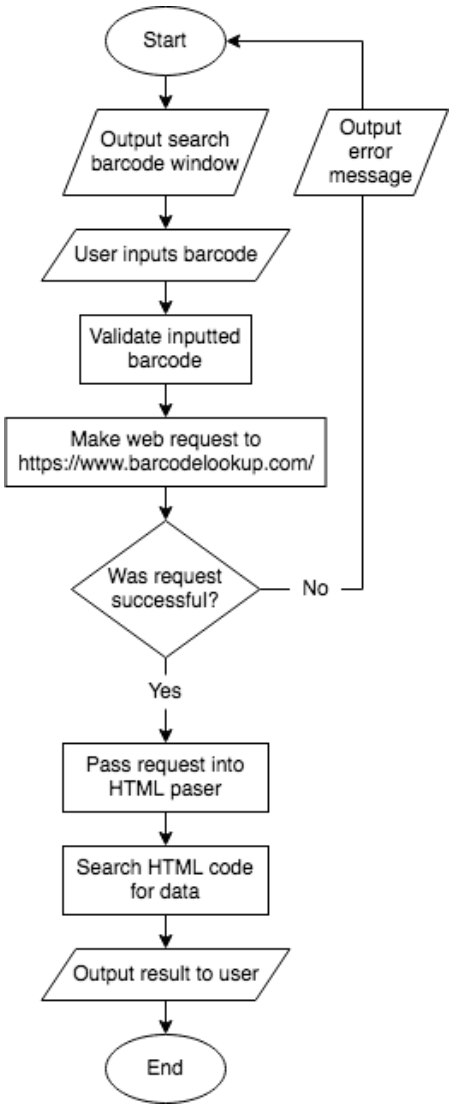



```
FUNCTION EditItem():
   //Find selected row and corresponding primary key
   SelectedRow = TableList.selectedRows().row()
   SelectedID = TableList.itemAt(SelectedRow, 0)

   //Select data from selected entry
   MyCursor.execute(
       "SELECT * FROM" + ActiveTable + "WHERE ID = " + SelectedID
   )
   MyResult = MyCursor.fetchall()

   //Create instance of ItemWindow class
   EditWindow = ItemWindow()

   //Insert retrieved data into text boxes in EditWindow
   Counter = 0
   FOR textboxes in EditWindow:
      textboxes.setText(MyResult[0][Counter])
      Counter += 1

   //Bind submit button to SubmitEdits function
   SubmitBTN.clicked.connect(SubmitEdits())

FUNCTION SubmitEdits():
   //Retrieve all columns from table for later use
   MyCursor.execute(
       "SHOW COLUMNS FROM" + MyResult1[0][0]
   )
   MyResult = MyCursor.fetchall()

   //Retrieve edited data from textboxes and store in array
   DataForUpdate = []
   FOR textboxes in EditWindow:
      DataForUpdate.append(textboxes.text())
```

```
//Construct SQL statement using column names and edited data
SQLStatement = "UPDATE " + ActiveTable + " SET "
FOR x in range(0, LEN(MyResult):
   SQLStatement = SQLStatement + MyResult[0][x] + "=" + DataForUpdate[x] + ", "
SQLStatement = SQLStatement + "WHERE ID =" + SelectedID)
//Execute statement
MyCursor.execute(SQLStatement)


MyDB.commit()   //Save changes to database
RUN PopulateTable()   //Refresh TableList
```

Search Barcode

This is an alternative method for adding an item to a collection. OpenCV will be used to output a live video feed allowing the user to scan a barcode. This barcode will then be searched via the website http://www.upcscavenger.com/. I will use the Requests library to send a HTTP request to the web server. The request will return a Response object which will then be passed into a HTML parser using the BeautifulSoup library. When the search is complete, the result will be presented to the user in an ItemWindow.



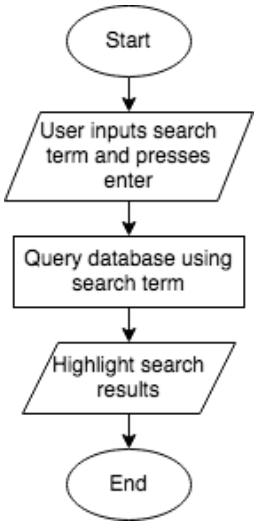

```
IMPORT OpenCV
IMPORT pyzbar
IMPORT requests
IMPORT BeautifulSoup

//This class displays the barcode search window
CLASS BarcodeSearchWindow():
   FUNCTION Init():

      //The live video feed will be implemented as a
      QLabel displaying a pixmap image
      FeedLabel = QLabel()
      InstructionLabel = QLabel(
       "Position Barcode in Camera View"
      )
      CancelBTN = QPushButton()
      //Create and start video capture thread
      Worker1 = Worker1()
      Worker1.start()
```

Pseudocode is continued on next page

```
//Worker1 inherits from the QThread class, it executes parallel to the main program
loop
//Its role is to capture each frame, check if the frame contains a barcode, and
then pass that frame to the main program loop for output via the FeedLabel
CLASS Worker1(QThread):
   FUNCTION run():
        ThreadActive = TRUE
        //Initialise connection with webcam
        Capture = cv2.VideoCapture(0)
        //WHILE loop periodically captures frames and passes them to the main
        program loop
        WHILE ThreadActive == TRUE:
               //Capture frame
               Frame = Capture.read()
               IF Frame:
                    //Set frame colour mode, flip frame and convert frame into Qt
                    compatible format
                    Image = cv2.cvtColor(Frame, cv2.COLOR_BGR2RGB)
                    FlippedImage = cv2.flip(Image, 1)
                    ConvertToQtFormat = QImage(FlippedImage)
                    //Pass frame into pyzbar
                    Barcode = pyzbar.decode(Frame)
                    //If pyzbar detects a barcode...
                    IF Barcode:
                       x, y, w, h = Barcode.rect
                       //Draw rectangle around barcode
                       RectImage = cv2.rectangle(FlippedImage, x, y, w, h)
                       RUN ImageUpdate(RectImage)
                       RUN BarcodeSearch(Barcode)
                    ELSE:
                       RUN ImageUpdate(FlippedImage)

   //This function passes each frame to the main program loop for outputting
   FUNCTION ImageUpdate(Image):
        FeedLabel.setPixmap(Image)

   //This function performs the online barcode search
   FUNCTION BarcodeSearch(Barcode):
        //Valid ISBN numbers must be 13 or 12 digits
        IF LEN(Barcode) < 12 OR LEN(Barcode) > 13:
               Pass
        TRY:
               //Make HTML request and pass request into parser
               Response = requests.get("http://www.upcscavenger.com/" + Barcode)
               Soup = BeautifulSoup(response.text, "html.parser")
               //Parse HTML file to find required data
               Name = soup.find(class_ = "col-md6-product).find("h4").get_text()
               RETURN Name
        //If TRY clause is unsuccessful, display error message
        EXCEPT
               RETURN Error
```

Simple Search

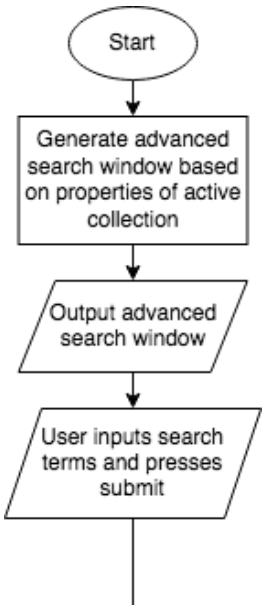
The search bar will be implemented as a QLineEdit object. The database will be searched based on the user's input and the resulting item(s) will be highlighted.




```
FUNCTION SimpleSearch():
   //Retrieve contents of search bar
   SearchTerm = SearchBar.text()

   //Construct SQL statement and format appropriately
   MyCursor.execute(
       "SHOW COLUMNS FROM" + ActiveTable]
   )
   MyResult1 = MyCursor.fetchall()
   SQLStatement = "SELECT * FROM " + ActiveTable + "WHERE"
   FOR x in range(0, LEN(MyResult1)):
      SQLStatement = SQLStatement + MyResult[x] + " LIKE %" +
      SearchTerm + "% OR"
   SQLStatement = SQLStatement[:-2]

   //Execute search
   MyCursor.execute(SQLStatement)
   MyResult2 = MyCursor.fetchall()

   //Highlight search results in TableList
   SearchResult = []
   FOR y in range(0, LEN(MyResult2):
      SearchResult.append(TableList.findItems(MyResult[y][0])
   FOR z in range(0, LEN(SearchResult):
      FOR x in SearchResult[z]:
         IF x.column() == 0:
            TableList.selectRow(x.row())
```

Advanced Search

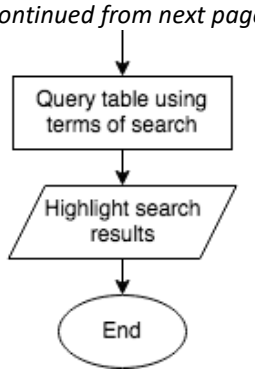
The advanced search function allows users to construct a very specific search query. The window will contain all the properties for the collection being searched and the user will then be able to input the terms of their search into this window.




```
//This class generates the window and populates it with widgets
CLASS SearchWindow():
   //ActiveTable is a global variable
   //SHOW FIELDS returns each column and its data type
   MyCursor.execute(
       "SHOW FIELDS FROM" + ActiveTable
   )
   MyResult1 = MyCursor.fetchall()
   //There will be a variable number of TextBoxes, so they will
   have to be stored in the TextBoxes array and referenced using
   their index
   TextBoxes = []

   //For each column, create a label and a text input box and
   add them to the window
   FOR x in range(0, LEN(MyResult1):
      SearchWindow.addWidget(QLabel(MyResult1.Field))
      TextBoxes[x] = SearchWindow.addWidget(QLineEdit())
```

Continued on next page

Continued from next page

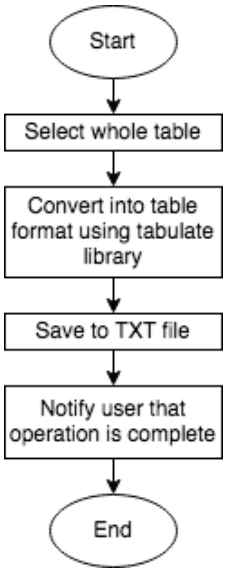



```
//If the data type for the current property is an integer, we need to limit the inputs for that line edit
IF MyResult1.Type == INTEGER:
   //In PyQt validators are used to restrict the type of characters that can be inputted into a line edit
   TextBoxes[x].setValidator(Qt.OnlyInt)

//Bind submit button to SubmitItem function
SearchBTN.clicked.connect(AdvancedSearch())
```

```
FUNCTION AdvancedSearch():
   //Retrieve column names from table
   MyCursor.execute(
       "SHOW COLUMNS FROM" + MyResult1[0][0]
   )
   MyResult1 = MyCursor.fetchall()

   //Pass data inputted into text boxes into array
   DataForSearch = []
   FOR x in range(0, LEN(TextBoxes)):
      //If nothing has been inputted into the text box, it is not added to the array
      IF TextBoxes[x].text() != NULL:
         DataForSearch.append([TextBoxes[x].text(), MyResult1[x]])
      ELSE:
         PASS

   //Construct SQL statement for advanced search
   SQLStatement = "SELECT * FROM " + ActiveTable + "WHERE"
   FOR y in range(0, LEN(DataForSearch):
      SQLStatement = SQLStatement + DataForSearch[y][1] + "=" + DataForSearch[y][0]
      + " AND "
   SQLStatement = SQLStatement[:-5]
   //Execute search
   MyCursor.execute(SQLStatement)
   MyResult2 = MyCursor.fetchall()

   //Highlight search results in TableList
   SearchResult = []
   FOR y in range(0, LEN(MyResult2):
      SearchResult.append(TableList.findItems(MyResult[y][0])
   FOR z in range(0, LEN(SearchResult):
      FOR x in SearchResult[z]:
         IF x.column() == 0:
            TableList.selectRow(x.row())
```

Export Collection to TXT File

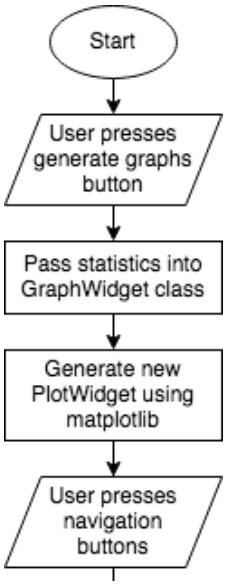
This function uses the Tabulate python library to format the result of a SQL query as an ASCII table. The output TXT file will be saved in the Root/Files directory.




```
IMPORT tabulate
IMPORT os

FUNCTION ExportToTXT():
    //Retrieve column names from table
    MyCursor.execute(
        "SHOW COLUMNS FROM" + MyResult1[0][0]
    )
    MyResult1 = MyCursor.fetchall()

    //Select contents of table
    MyCursor.execute(
        "SELECT * FROM " + ActiveTable
    )
    MyResult2 = MyCursor.fetchall()

    //Format table contents and column names as ASCII table
    Table = tabulate(MyResult2, headers = MyResult1, tablefmt = 'psql'

    //Write table to text file
    TextFile = open("/Root/Files/Output.txt", "w")
    TextFile.write(Table)
    TextFile.close()

    //Use AppleScript to notify user that export is complete
    os.system("osascript -e display notification \"Export Complete!")
```

Generate Graphs

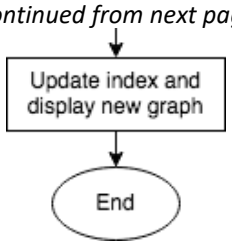
I will use matplotlib to generate my graphs. The graph widget will comprise of a PlotWidget alongside two buttons, one to move to the next graph and another to go back to the previous graph. The graph widget will be implemented as an object inheriting from the QWidget class. The program will calculate the values to be passed into the graph widget class in the main program loop. An instance of the class will then be created and the values will be fed into it.



Continued on next page

```
//Matplotlib is a python graphing library with native support for PyQt
IMPORT matplotlib

CLASS GraphWidget(QWidget):
    //The init function runs when a class instance is first created
    FUNCTION init():
        //Select all columns from table and store in array
        MyCursor.execute(
          "SHOW COLUMNS FROM " + ActiveTable
        )
        GraphType = MyCursor.fetchall()
        GraphIndex = 0

    FUNCTION InitGraph():
        //Initialise buttons and bind to functions
        BackBTN = QPushButton()
        BackBTN.clicked.connect(PreviousGraph())
        FwdBTN = QPushButton()
```

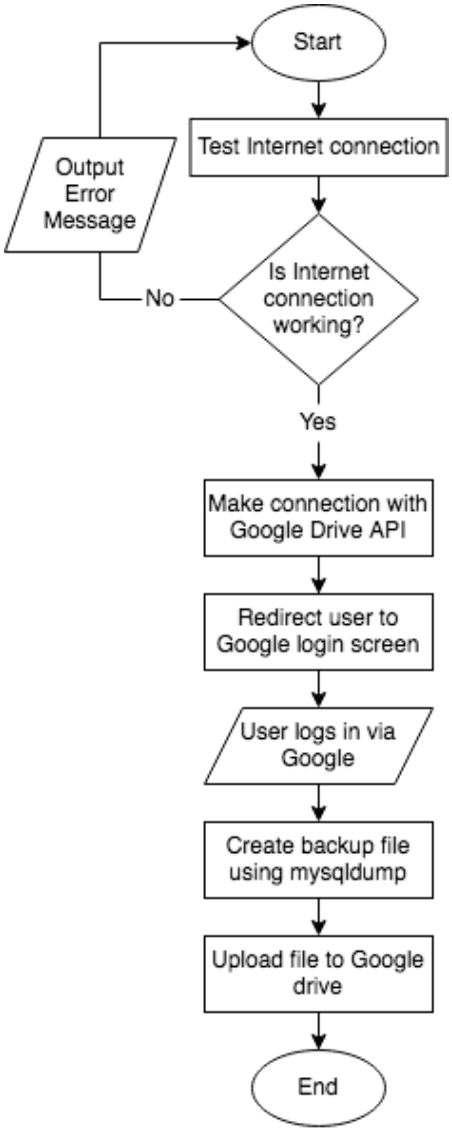
Continued from next page




```
    FwdBTN.clicked.connect(NextGraph())
    //Create figure and canvas for drawing graph on
    Figure = Figure()
    Canvas = FigureCanvas(Figure)

FUNCTION DrawGraph():
    Figure.clear()
    //Select the number of items from the field currently being
    viewed
    MyCursor.execute(
        "SELECT" + GraphType[GraphIndex] + ", COUNT(*) AS c FROM"
        + ActiveTable + "GROUP BY" + GraphType[GraphIndex]
    )
    MyResult = MyCursor.fetchall()

    //Labels stores the categories data is to be split into
    Labels = []
    //Pcent stores the percentage of each category
    Pcent = []
    //Total stores the total amount of items
    Total = 0
    FOR x in range(0, LEN(MyResult)):
        Total += MyResult[x][1]
    FOR y in range(0, LEN(MyResult)):
        Pcent.append((MyResult[y][1] / self.Total) * 100)
        Labels.append(MyResult[y][0])
    //Draw graph
    Canvas.draw(Pcent, Labels)

//These two functions handle switching between different graphs
FUNCTION PreviousGraph():
    IF GraphIndex <= 0:
        GraphIndex = LEN(GraphType) - 1
    ELSE:
        GraphIndex -= 1
    RUN DrawGraph()
FUNCTION NextGraph():
    IF GraphIndex >= LEN(GraphType):
        GraphIndex = 0
    ELSE:
        GraphIndex += 1
    RUN DrawGraph()
```

Upload Collection to Google Drive

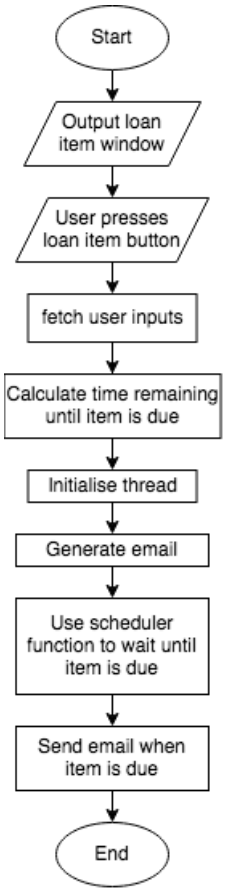
The first step is to test the user's internet connection using the requests library. This function will be implemented using the Google Drive python API (PyDrive) and the mysqldump tool to create back up .sql files. I have set up a development account with Google in order to use the API. The API works using a key system stored in a JSON file called "client_secrets.json". This file can be created via the google drive web platform. The file needs to be copied into the program's root directory for the authentication process to work.




```
IMPORT pydrive
IMPORT requests
IMPORT subprocess
IMPORT os

//Function uses TRY EXCEPT clause to test internet
connection
FUNCTION TestInternet():
   TRY:
      requests.urlopen('https://www.google.com/')
      RETURN TRUE
   EXCEPT:
      RETURN FALSE

FUNCTION BackupCollection():
   //If an internet connection is not present, the
   upload operation cannot continue
   IF TestInternet() == False:
      OUTPUT ErrorMessage
   ELSE:
      //Create an object to handle authentication
      GoogleLogin = GoogleAuth()
      //Opens browser and displays login screen
      GoogleLogin.LocalWebserverAuth()
      //Create Google Drive object to handle creating
      and uploading files
      Drive = GoogleDrive(GoogleLogin)

      //Create SQL back up and save to temp directory
      subprocess.Popen(
         mysqldump -h [DB host name]
         -u [DB username]
         -p [DB name]
         > /Root/Temp/Backup.sql
      )

      //Open file
      WITH OPEN ("/Root/Temp/Backup.sql", "r") as File:
         UploadFile = Drive.CreateFile(
               {"title":os.path.basename(File.name)}
         )
         UploadFile.setContentString(File.read())
         //Upload file
         UploadFile.upload()

      //Delete temp file after upload is complete
      os.remove("/Root/Temp/Backup.sql)
```

Loan Item

The first loan item function will display a dialogue window and collect the selected item along with the due date. The dialogue window will contain input boxes for the due date and time as well as a list containing all the items in that collection (this list will be initialised in the same manner that the list in the mainwindow is initialised). When the user presses the “loan item” button, a separate thread will be initialised using the QThread class. The thread will wait until the item is due, after which an email will be sent notifying the user that the item is due. Emails will be generated using the smtplib library.




```
IMPORT smtplib
IMPORT sched
IMPORT datetime
IMPORT dateutil

FUNCTION LoanItem():
   OUTPUT Loan Item Window

   //Retrieve user inputs
   SelectedItemName = List.itemAt(List.selectedRows())
   Day = DayTextBox.text()
   Month = MonthTextBox.text()
   Year = YearTextBox.text()
   Hour = HourTextBox.text()
   Minute = MinuteTextBox.text()

   //Get current date and time
   CurrentDateTime = datetime.now()
   DueDateTime = datetime.datetime(Year, Month, Day, Hour, Minute)
   //Calculate how long the scheduler has to wait until the due date
   SecondsToWait = (DueDateTime – CurrentDateTime).total_seconds()

   //Start new thread, set parameters to selected item and time
   remaining
   NewThread = Worker1(Item = SelectedItemName, Time = SecondsToWait)
   NewThread.start()

//This class is the thread that is executed when an item needs to be
loaned
CLASS Worker1(QThread):
   FUNCTION init(Item, Time)
      //Initialise scheduler and bind scheduler to SendEmail function
      Scheduler = sched.scheduler(time.time, time.sleep)
      Scheduler.enter(Time, 1, SendEmail())
      Scheduler.run()
   //This function handles sending the email
   FUNCTION SendEmail():
      EmailText = Item + “is due”
      //Setup smtp server
      with smtplib.SMTP_SSL("smtp.gmail.com", Port, context = Context)
      as Server:
         Server.login(Email, Password)   //Login
         Server.sendmail(SenderEmail, ReceiverEmail, EmailText)
         Server.quit()   //Quit server when email is sent
```

Testing Plan

White Box Testing

White box testing will take place during the development of my application in order to rigorously test the internal structures of my system and root out any logic, runtime or syntax errors. The main purpose of this phase of testing will be to ensure that the code is functional and to identify any bugs. I will use unit testing to regularly test each module as I code it. For simple tests such as input validation or logic tests, I will use a Python unit testing framework called PyTest which uses "assert" statements to test the expected output of a module. More complex tests will have to be carried out manually. Each test will comprise of valid, erroneous and borderline data. Before testing, I will have to record the expected output of the test, then, once the test has been performed, I will record the actual output and use that result to determine whether the module has passed the test or whether it needs further adjustment. If the program crashes, I will include a screenshot of the program console.

After each module clears the unit testing phase, I will progress to integration testing in order to ensure that the individual modules work together without any issues. I will use the "big bang" approach to integration testing in the hope that, as I code more of my project, the testing system will become larger until, hopefully, by the end of the project, I will have a completed system of integrated modules.

Alpha Black Box Testing

I will carry out alpha testing once the program is complete in order to ensure that I have fulfilled the success criteria and to deal with any minor bugs before I hand the program over to a sample of end users for beta testing. The table below outlines the initial round of tests I will carry out and the corresponding success criteria that they fulfil (please refer to page 21 for criteria numbers):

Criteria Number	Description	Test Data	Expected Outcome
2	Perform speed tests on key modules using the datetime library Start = datetime.now() Insert code here End = datetime.now() print(End – Start)	Start and End times	Test should output the time it takes for a block of code to execute. This output will then be used to determine whether the code is executing at an acceptable level of efficiency
5, 6, 7, 8, 9, 10, 11, 12	Test for SQL injection vulnerabilities in login screen and all other instances where inputted data is used in a SQL statement	Pass the string "- -" into SQL statements	The program should reject this input
16, 20, 23	Internet connectivity issues		An error message should be displayed giving the user the option to try another internet connection test
13	SQL server connectivity issues		An error message should be displayed giving the user the option to try another SQL connection test

25	All of the main functions should be mapped to keyboard shortcuts, these shortcuts need to be tested to ensure that they do not override any system shortcuts		Each shortcut should execute the relevant function. No shortcuts should be the same as system-wide shortcuts
26	Database must be ACID compliant		I will evaluate my database design and implementation against each of the ACID principles
23	Invalid barcode input	Random 13-digit number	The program should ignore this input
20	File corruption during Google Drive backup	Corrupted .sql file	The program should try uploading the file again
21	Invalid loan due date and time	3-digit number and letter inputs e.g. “zzz” or 003	The input box’s validator should reject these inputs
22	Thread class overload		Python’s inbuilt sigkill thread handler should deal with this issue. Changes to the SQL server should be periodically saved so no data should be lost
5	Brute force password attack	Random password guesses	The system should lock the user out if they input incorrect credentials more than 3 times

After carrying out these tests, I will use destructive testing to push the system to its limits. Destructive testing will involve using erroneous and borderline test data to try and uncover flaws in the program. The destructive test will be peer reviewed by a fellow classmate who is less familiar with the system.

The final phase of alpha testing will be a general usability test to ensure that the software is ready for beta testing.

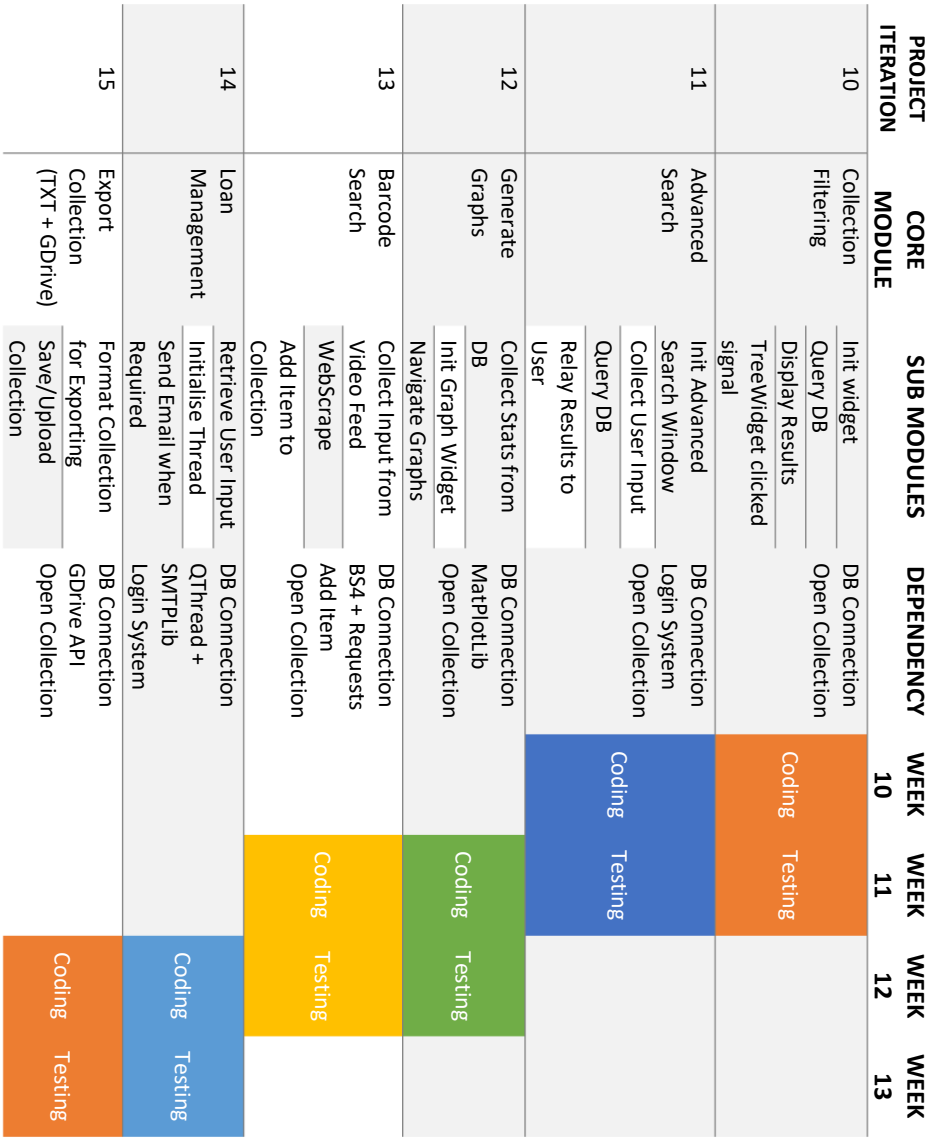
Beta Testing

During the beta testing phase, a sample of end-users will be able to test the program in order to observe how the program will work in a real environment. The end-users will be provided with the success criteria on page 21 and I will ask them to tick off the criteria that they believe the software has fulfilled. The end-users will also be asked to rate the software out of 5 based on certain aspects such as ergonomics, UI design and efficiency. They will also be asked long-form questions such as “How will this software benefit you and your collection?”. These results and any other feedback will then be collated and used in the Evaluation stage.

Section 3: Development

Planning and Organisation

PROJECT ITERATION	CORE MODULE	SUB MODULES	DEPENDENCY	WEEK 1	WEEK 2	WEEK 3	WEEK 4	WEEK 5	WEEK 6	WEEK 7	WEEK 8	WEEK 9
1	Write CSS Stylesheet	Each PyQt class will have its own sub module within the stylesheet		Coding	Testing							
2	Test / Initialise Database Connection	Try Connection Successful	MySql.connector	Coding	Testing							
		Unsuccessful										
3	Login System	Initialise UI	DB Connection		Coding and Testing							
		Successful Login										
		Unsuccessful Login										
		Lock Window										
		Create Account										
4	Display Collections	Query DB	Login System DB Connection			Coding	Testing					
		Generate Buttons										
		Btn Clicked Signal										
5	Create Collection	Add New Property	Login System DB Connection			Coding	Testing					
		Submit Collection										
6	Open Collection	Query DB	Display Collection Login System DB Connection					Coding	Testing			
		Initialise Table										
		Populate Table										
		Initialise Toolbar										
7	Add Item	Init Add Item Window	Open Collection DB Connection Login System							Coding	Testing	
		Insert item into DB										
		Populate Table										
8	Delete Item	Retrieve Selected Item PK	Open Collection DB Connection Login System							Coding	Testing	
		Delete from DB										
		Populate Table										
9	Edit Item	Retrieve Selected Item PK	Open Collection DB Connection Login System									Coding
		Init Window										
		Apply Changes										



The Gantt chart inserted above breaks down how I will timetable the programming of each module within my code. Each core module breaks down into several sub modules. Most modules are dependent on other modules, so I have structured the development timeline in a way that ensures I will only begin coding a module when all its dependencies have already been coded and tested. I will be using the Rapid Application Development methodology when developing the software. This means that each development iteration should produce a working prototype which can be tested and evaluated before moving on to the next prototype. I have also used the computational technique of pipelining to queue up the coding of different modules by having the testing stage of each module overlap with the coding stage of the next module. Certain modules which have similar functionalities or share methods will be developed in parallel to help speed up the development process.

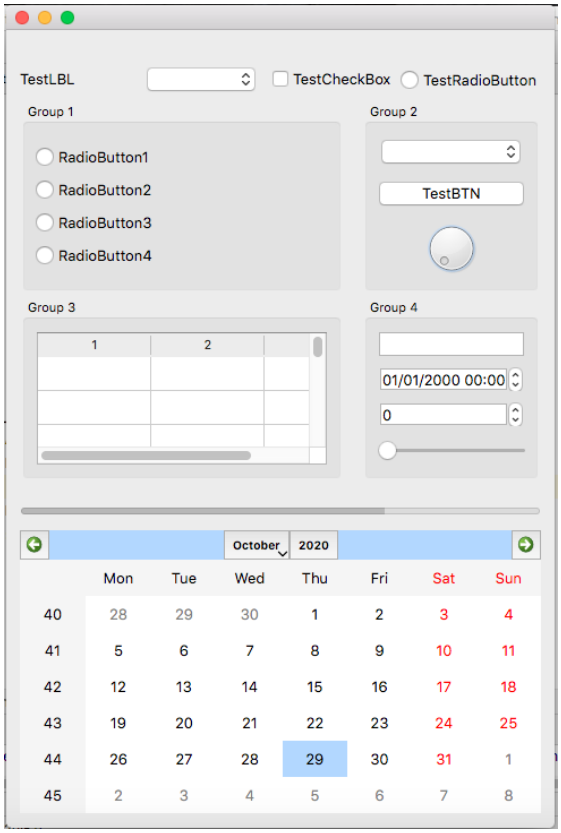
Iteration 1; CSS Stylesheet

The first stage in development involves coding the stylesheet which will be applied to the application. The colour scheme I have decided to use consists of a white background, black text, lime green as a primary colour and a darker shade of green for highlights. To apply the stylesheet to my PyQt window, the following code must be executed:

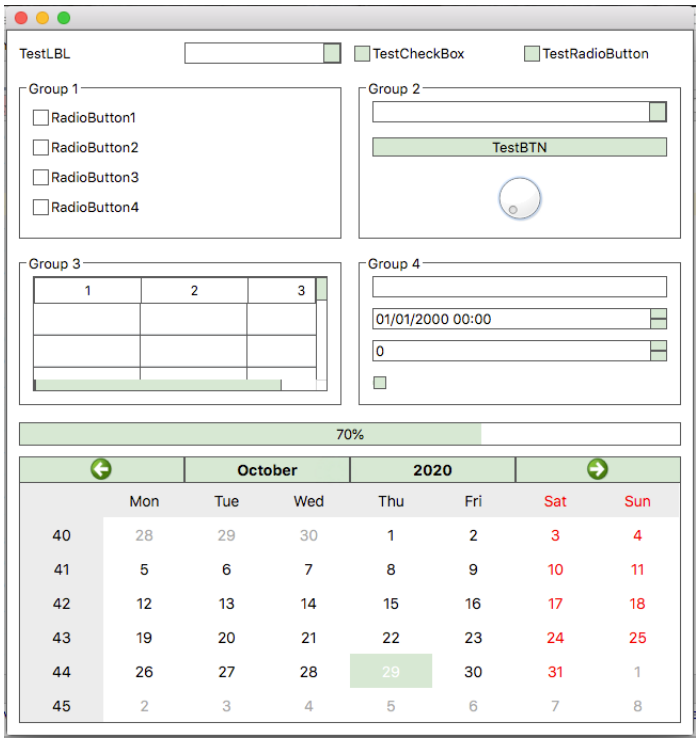
```
with open("Stylesheet/StyleSheet.txt", "r") as ss:
    Root.setStyleSheet(ss.read())
```

Root is the variable name of the PyQt window the stylesheet is being applied to.

I tested the stylesheet on the window pictured below. The window contains an instance of each of the most commonly used PyQt widgets:

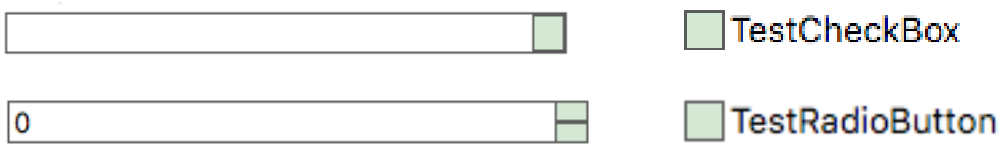


Window without stylesheet



First test of window with stylesheet

The first stylesheet test was largely successful, however some issues needed ironing out. First off, changing the stylesheet from the default meant that symbols such as the arrows on `QComboboxes` and the ticks in `QCheckBoxes` no longer appeared.



As the above images demonstrate, the arrow symbols will no longer render on `QComboBoxes` and `QSpinBoxes`. In addition, rather than displaying a tick or a circle when they are activated, checkboxes and radio buttons just display a green square.

I remedied this issue by binding icons to the indicator class of the checkbox/radiobutton objects and the buttons of the combobox/spinbox classes. This was carried out by inserting the following blocks of code into my stylesheet:

```
QCheckBox::indicator:checked {
    image: url(Stylesheet/Tick.png);
}
```

```
QRadioButton::indicator:checked {
    image: url(Stylesheet/Radio.png)
}
```

```
QAbstractSpinBox::down-arrow {
   width: 7px;
   height: 7px;
```

```
QComboBox::down-arrow {
   border: 0px solid #5A5A5A;
   background: #D5E8D4;
   width: 7px;
   height: 7px;
   image: url(Stylesheet/Down.png);
}
```

```
QAbstractSpinBox::up-arrow {
   width: 7px;
   height: 7px;
   border: 0px solid #5A5A5A;
   image: url(Stylesheet/Up.png);
}

QAbstractSpinBox::down-arrow {
   width: 7px;
   height: 7px;
   border: 0px solid #5A5A5A;
   image: url(Stylesheet/Down.png);
}
```

With these changes applied, checkboxes now display a tick icon when activated, radiobuttons display a green circle, comboboxes display a downwards arrow and spinboxes have upwards and downwards arrows on their buttons.

Another issue was that certain widgets were being rendered with borders when they weren't required. This issue was remedied by creating a subclass of `QWidget` with the id name `BorderlessWidget`:

```
QWidget#BorderlessWidget {
    border: 0px;
}
```

PyQt widgets which don't need borders can now be implemented by setting the `ObjectName` property to "`BorderlessWidget`".

The complete stylesheet can be found in the appendix.

Iteration 2; Test/Initialise Database Connection

I began this iteration by creating the program's SQL database

```
CREATE DATABASE Anthology
```

Before I populated the database with tables and data, I wrote the function which would test the SQL connection and either launch the login screen or display an error message accordingly.

I began by coding a simple function which uses a try/except clause to test the database connection.

```
import mysql.connector
# Tests the program's connection with the SQL database
def TestConnection():
    # Try establishing a connection with the database and creating a cursor
    try:
        MyDB = mysql.connector.connect(
            host = "localhost",
            user = "root",
            password = "[redacted]",
            database = "Anthology"
        )
        MyCursor = MyDB.cursor()
    # If the program encounters an error when trying to connect, an instance of
      the error popup is executed
    except:
        SQLErrorPopupInstance = SQLErrorPopup()
```

For development purposes, I have passed the value `localhost` into the host argument. This is because the SQL server is being hosted on the laptop I am developing the software on and I am therefore having to connect to it over a UNIX socket rather than a TCP/IP socket. If I were to distribute the software to other devices, I would have to set up a remote connection, however the `localhost` connection suffices for what I am currently doing.

If the try clause is executed successfully, the program will be able to interface with the database using the cursor object. If it encounters an error when executing the block of code in the try clause, the code in the except clause will be executed instead. `SQLErrorPopup()` is the object containing the error popup.

```
# This object inherits from the QMessageBox class which is used to create popup
  windows
# This popup is displayed when a connection with the SQL database can't be
  established
class SQLErrorPopup(QMessageBox):
    def __init__(self):
        super(SQLErrorPopup, self).__init__()

        # Initialise window elements
        self.setText("Database Error")
        self.setText("There was an error connecting to the database")
        self.setInformativeText(
        """
         We came across an error when trying to initialise a connection with our
         database.
         Please press the retry button to try connecting again.
         If the problem persists, please contact customer support
        """)
        self.setIcon(QMessageBox.Critical)
        self.setWindowFlags(Qt.FramelessWindowHint)

        self.setStandardButtons(QMessageBox.Retry | QMessageBox.Cancel)
```

```
        # If the user presses the retry button, the TestConnection function
           will be executed again
        self.button(QMessageBox.Retry).clicked.connect(TestConnection)

        # If they press the cancel button, the application will close
        self.button(QMessageBox.Cancel).clicked.connect(self.close)

        # Apply stylesheet to popup window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Initialize QMessageBox thread
        self.exec()
```

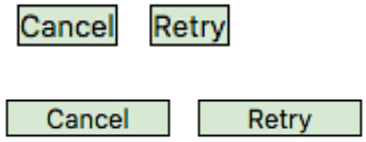
Finally, I initialized the program's main loop and ran the `TestConnection` function.

```
# Main program loop
if __name__ == "__main__":
    # A QApplication instance must be created before any windows are displayed
    App = QApplication(sys.argv)
    App.setWindowIcon(QIcon("Resources/Icon.png"))
    # Begin database connection test
    TestConnection()
```

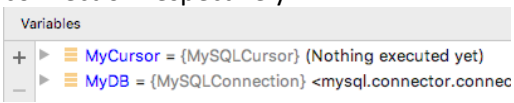
Later on in the development process, I plan to reuse some of the code written in this iteration in order to program a thread which will run alongside the main program loop and periodically test the database connection and display an error message if the connection is severed whilst the program is in use.

Testing

On a cosmetic level, the buttons in the popup window needed resizing

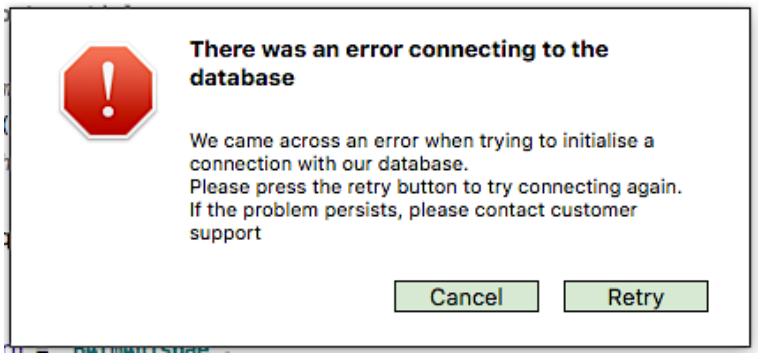



```
/*Change button width to 80 pixels if they are within a
QMessageBox*/
QMessageBox QPushButton {
    width: 80px;
}
```

Test	Expected Result	Actual Result
Run code whilst SQL server is active	The program should successfully connect with the server and create a cursor object	The program works as expected. The screenshot inserted below shows that, when a connection can be made with the server, two variables called MyCursor and MyDB are created which store the database cursor and the database connection respectively 
Run code whilst SQL server is not active	A popup window should be displayed with a suitable error message and buttons giving the user the option to cancel or retry. The user shouldn't be able to interact with any other	A popup is generated if a connection can't be established. The popup inherits from the `QMessageBox` class so it is automatically considered a modal window. I will discuss the appearance of the popup and a few minor issues I encountered further on

	program windows whilst this popup is being displayed	
Press retry button whilst SQL server remains inactive	The error message should be displayed again. The user should be able to continue pressing the retry button indefinitely or until a connection can be established	The popup is redisplayed, and the user can retry indefinitely
Run code whilst SQL server is not active but activate SQL server before pressing retry button	This test is intended to simulate how the program would behave if the connection drops out but is then re-established. The program should create a new SQL cursor once the new connection is initiated and the popup window should close	This test was successful. The only minor change that needed to happen was adding a line of code to close the error popup once the connection had been re-established

This is the popup window which is displayed when a connection can't be established with the SQL server:



The following error message appears in the console when the user presses the cancel button:

```
2020-11-08 19:12:53.092 Python[90583:45427981] modalSession has been exited prematurely - check for a reentrant call to endModalSession:

Process finished with exit code 0
```

The message does not affect the execution of the program. I managed to capture the following stack trace for the error:

```
Thread 0 Crashed.Dispatch queue: com.apple.main-thread
0 libobjc.A.dylib 0x00007fff8e3ed097 objc_msgSend + 23
1 com.apple.CoreFoundation 0x00007fff8c45243f CFRelease + 591
2 com.apple.CoreFoundation 0x00007fff8c465699 -[__NSArrayM dealloc] + 185
3 libobjc.A.dylib 0x00007fff8e3ef65a (anonymous
namespace).AutoreleasePoolPage.pop(void*) + 502
4 com.apple.CoreFoundation 0x00007fff8c471932 _CFAutoreleasePoolPop + 50
5 com.apple.Foundation 0x00007fff8f5d1e17 -[NSAutoreleasePool drain] + 147
6 com.apple.AppKit 0x00007fff8c712a48 -[NSApplication run] + 725
7 QtGui 0x000000010bd2e4b3
QEventDispatcherMac.processEvents(QFlags<QEventLoop.ProcessEventsFlag>) + 419
8 QtCore 0x000000010b96e464
QEventLoop.processEvents(QFlags<QEventLoop.ProcessEventsFlag>) + 68
9 QtCore 0x000000010b96e814 QEventLoop.exec(QFlags<QEventLoop.ProcessEventsFlag>) + 324
10 QtCore 0x000000010b97108c QCoreApplication.exec() + 188
```

The lines highlighted in red suggest that this is a problem specific to MacOS. A little extra research proved that this is a widely documented bug within Qt [5] [6] [7] and is therefore out of my control to amend. There are patches available, however I felt it best to not tamper with the library code as the error does not affect processing or cause the program to crash.

Review

At this stage in development, I have created the SQL database the program will use and programmed a function to test the database connection and handle any errors that arise with that connection. Testing during this iteration included running simple unit tests whilst I was writing the code and more comprehensive tests once the iteration was complete (these tests are discussed in the previous section). Because I haven't coded any other modules yet, I can't perform any integration testing. However, when iteration 3 has been completed, I will perform an integration test in order to determine whether the individual modules work together.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- An error message should be displayed if a connection can't be made with the SQL server (14)

Iteration 3; Login System

The first step of this iteration was creating the Users table. `PK_Users` is the primary key for the table; hence it takes the parameters `NOT NULL`, `AUTO_INCREMENT` and `PRIMARY KEY`. All passwords need to be hashed before they are stored in the table. Emails will be validated in the python frontend before they are written to the database.

```
CREATE TABLE Users
  (
     PK_Users     INTEGER(500) NOT NULL AUTO_INCREMENT PRIMARY KEY,
     Email        VARCHAR(320),
     PasswordHash VARCHAR(200)
  )
```

PK_Users	Email	PasswordHash
1	sami.hatna2003@gmail.com	3b4e255fb210ef2ef85358f14cee16c3beeceb716b5fcd153d9c7b12
2	14shatna@tythy.school	93595faac52074bcb7efe99ceb9b82fa5b3a1848ee24df33df0c819a

I implemented the login window as an object inheriting from the `QMainWindow` class. The window is formatted as a horizontal box layout containing two vertical box layouts, one for the artwork and one for the actual login form.

```
# This object inherits from the QMainWindow class
# It displays the login window, validates user credentials and grants the user access if
they provide valid login details
class LoginWindow(QMainWindow):
    def __init__(self):
        super(LoginWindow, self).__init__()

        # The central widget is a container for all the other widgets the window displays
        self.CentralWidget = QWidget()
        self.CentralWidget.setObjectName("BorderlessWidget")
        self.setCentralWidget(self.CentralWidget)
        self.setWindowTitle("Login")

        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Keeps track of how many unsuccessful login attempts have been made
        self.Attempts = 4
        # Keeps track of how many times the user has made more than 4 login attempts
        self.Magnitude = 1

        self.InitUI()
```

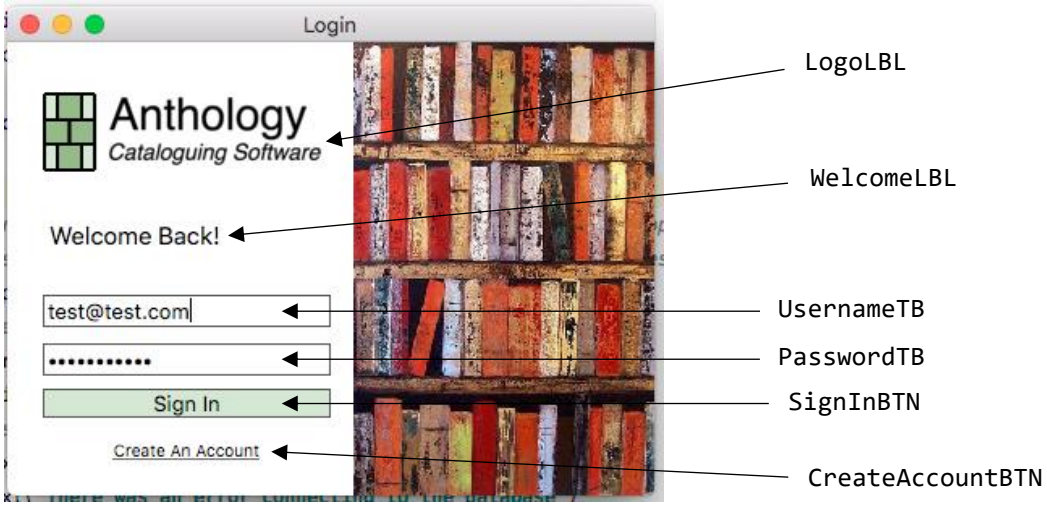
```
        # TestConnection has been repurposed as a method of the LoginWindow class
        self.TestConnection()
```

Once the window has been initialised, it needs to be populated with widgets:

```
    # This method initialises all the UI elements in the window
    def InitUI(self):
        self.LoginVBL1 = QVBoxLayout()

        # Displays an image of the app logo
        LogoLBL = QLabel()
        LogoPixmap = QPixmap("Resources/Logo.png")
        LogoLBL.setPixmap(LogoPixmap)
        LogoLBL.setAlignment(Qt.AlignCenter)
        self.LoginVBL1.addWidget(LogoLBL)

        self.WelcomeLBL = QLabel("Welcome Back!")
        self.WelcomeLBL.setStyleSheet("padding-top:20px; padding-bottom:20px; size:15px")
        self.LoginVBL1.addWidget(self.WelcomeLBL)

        # Text box for user to input username
        self.UsernameTB = QLineEdit()
        self.UsernameTB.setPlaceholderText("Username")
        self.LoginVBL1.addWidget(self.UsernameTB)

        # Text box for user to input password
        self.PasswordTB = QLineEdit()
        # EchoMode determines how the text will be displayed
        # In this case it will be hidden because the password is sensitive information
        self.PasswordTB.setEchoMode(QLineEdit.Password)
        self.PasswordTB.setPlaceholderText("Password")
        self.LoginVBL1.addWidget(self.PasswordTB)

        self.SignInBTN = QPushButton("Sign In")
        # Bind sign in button to CheckCredentials function
        self.SignInBTN.clicked.connect(self.CheckCredentials)
        self.LoginVBL1.addWidget(self.SignInBTN)

        # This button is used to create a new account
        self.CreateAccountBTN = QPushButton("Create An Account")
        self.CreateAccountBTN.setStyleSheet("""
        QPushButton:pressed { color: D5E8D4; }
        QPushButton:hover:!pressed { color: #96BA8A; }
        QPushButton { border: 0px; background-color: white; font-size: 10px; }
        """)
        ButtonFont = QFont()
        ButtonFont.setUnderline(True)
        self.CreateAccountBTN.setFont(ButtonFont)
        # Bind CreateAccountBTN to CreateAccountMethod function
        self.CreateAccountBTN.clicked.connect(self.CreateAccountMethod)
        self.LoginVBL1.addWidget(self.CreateAccountBTN)

        # Displays artwork alongside login form
        ArtLBL = QLabel()
        ArtPixmap = QPixmap("Resources/LoginArt.png").scaledToHeight(280)
        ArtLBL.setPixmap(ArtPixmap)

        # Combine layouts and add layouts to central widget
        self.LoginHBL1 = QHBoxLayout()
        self.LoginHBL1.addLayout(self.LoginVBL1)
        self.LoginHBL1.addWidget(ArtLBL)
        self.LoginHBL1.setContentsMargins(0, 0, 0, 0)
        self.LoginVBL1.setContentsMargins(22, 22, 0, 22)
        self.CentralWidget.setLayout(self.LoginHBL1)
```

```
        # Prevent user from being able to resize window
        self.setFixedSize(400, self.minimumHeight())
```

This code produces the following output:



The next step is to write the function (called `CheckCredentials`) that will take the inputted username and password and determine whether these credentials match the ones stored in the database. Passwords are stored in the database as hashes, so the user input must be hashed before it can be compared with the database contents. This is carried out using a python library called `hashlib` which uses the SHA224 hashing algorithm.

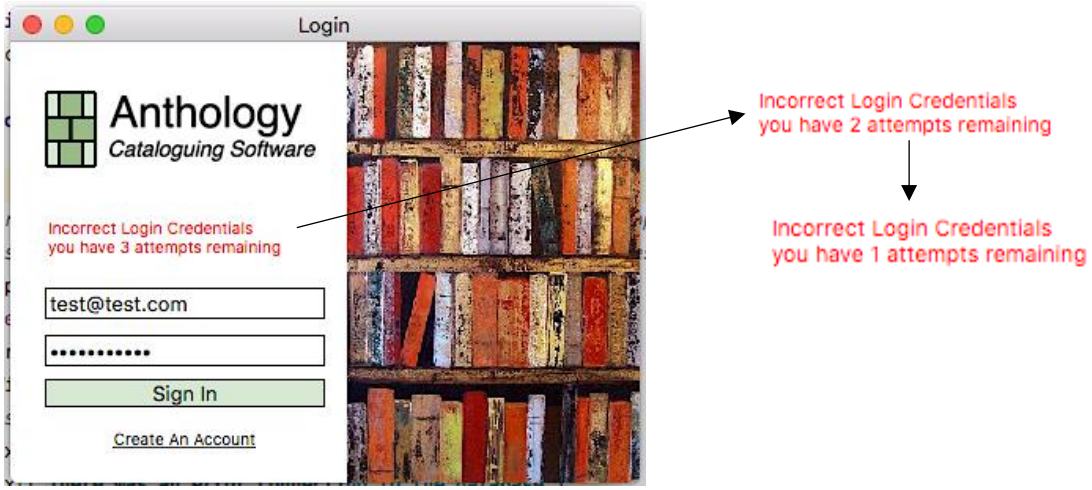
```
self.InputPassword = hashlib.sha224(
                                    bytes(self.PasswordTB.text(), encoding = "utf-8")
                                   ).hexdigest()
```

The `CheckCredentials` function makes use of the attribute `Attempts` to keep track of how many failed login attempts the user has made. Users are limited to four attempts. If they exceed this limit, they won't be able to login for an amount of time. The amount of time is a multiple of ten seconds which increases every time the user exceeds four login attempts. So, the first time the user passes the limit they will have to wait ten seconds, they will have to wait 20 seconds the second time, 30 seconds the third time etc. Because they are attributes, `self.Attempts` and `self.Magnitude` are both initialized in the `__init__` method.

```
# This function checks user inputted credentials against the credentials stored in the
  database
def CheckCredentials(self):
    # If the user hasn't exceeded the maximum amount of attempts permitted...
    if self.Attempts > 1:
        self.InputUsername = self.UsernameTB.text()
        # Passwords stored in the database are hashed so the user input has to be hashed
          before it can be compared
        self.InputPassword = hashlib.sha224(bytes(self.PasswordTB.text(), encoding = "utf-
                             8")).hexdigest()

        # Retrieve all entries from the Users table
        self.MyCursor.execute("SELECT * FROM Users")
        MyResult = self.MyCursor.fetchall()

        # Boolean value indicates whether a successful login has been made
        self.SuccessfulLogin = False

        # Iterate through query result and compare each username and password combination
          with the user inputs
        for x in range(0, len(MyResult)):
            if MyResult[x][1] == self.InputUsername:
```

```
                if MyResult[x][2] == self.InputPassword:
                    self.SuccessfulLogin = True
                    self.ActiveUserID = MyResult[x][0]
                    break

        # If the user has inputted correct credentials, they will be logged into the
          application and the login window will close
        if self.SuccessfulLogin:
            # Log user in
            self.CollectionWindow = CollectionWindow.CollectionWindow(MyCursor =
            self.MyCursor, MyDB = self.MyDB, ActiveUserID = self.ActiveUserID)
            self.close()

        # If their credentials are incorrect, an error message is displayed informing them
          of how many attempts they have remaining
        else:
            self.Attempts -= 1
            self.WelcomeLBL.setText("Incorrect Login Credentials\nyou have " +
                                    str(self.Attempts) + " attempts remaining")
            self.WelcomeLBL.setStyleSheet("color: red; font-size: 10px; padding-top: 20px;
                                          padding-bottom: 10px;")

    # If user has exceeded max attempts, login screen is disabled for a period of time
    else:
        self.WelcomeLBL.setText("Too many incorrect login attempts\nLogin disabled for {0}
                                seconds".format(10 * self.Magnitude))
        self.SignInBTN.setEnabled(False)
        self.UsernameTB.setEnabled(False)
        self.PasswordTB.setEnabled(False)
        self.UsernameTB.setStyleSheet("border-color: red;")
        self.PasswordTB.setStyleSheet("border-color: red;")
        self.SignInBTN.setStyleSheet("border-color: red; background-color: #ececec")
        self.setCursor(Qt.ForbiddenCursor)
        # Time period for which window is disabled is calculated by multiplying 10 seconds
          by the variable self.Magnitude
        # Each time the user exceeds the max, self.Magnitude is incremented so they will
          have to wait 10 seconds longer
        QTimer.singleShot(10000 * self.Magnitude, partial(self.TimerSlot))
```

When the user inputs invalid credentials, the program now displays an error message informing them of how many attempts they have remaining:



In the above code, I have used the `QTimer` class to schedule the execution of a function. `QTimer` creates a concurrent thread which waits for a specified amount of time before executing the function `self.TimerSlot`. This is a superior approach to using `time.sleep` because it means the main program loop doesn't freeze whilst the program is waiting to execute the function.

I now need to code the function `self.TimerSlot` which is executed when the window needs to be reenabled.

```
# TimerSlot is called when the QTimer is complete
# It re-enables the login window, resets the no. of attempts and increments self.Magnitude
def TimerSlot(self):
    self.SignInBTN.setEnabled(True)
    self.UsernameTB.setEnabled(True)
    self.PasswordTB.setEnabled(True)
    self.WelcomeLBL.setText("Welcome Back!")
    self.WelcomeLBL.setStyleSheet("padding-top: 20px; padding-bottom: 20px; font-
    size: 15px;")
    self.UsernameTB.setStyleSheet("border-color: black; border: 1px solid
    #5A5A5A;")
    self.PasswordTB.setStyleSheet("border-color: black; border: 1px solid
    #5A5A5A;")
    self.SignInBTN.setStyleSheet("border-color: black; border: 1px solid #5A5A5A;")
    self.setCursor(Qt.ArrowCursor)
    self.Attempts = 4
    self.Magnitude += 1
```

Now, when the user exceeds their four logon attempts, `self.UsernameTB`, `self.PasswordTB` and `self.SignInBTN` will be disabled for an amount of time dependent on the value of `self.Magnitude`. When the `QTimer` is done counting down that time, `self.TimerSlot` will be executed in order to reenable the login window.

The final stage of this iteration is creating the system for users to create a new account. This will be implemented as a modal window which is displayed over the login screen.

Three forms of validation are required when users are creating a new account:

- A presence check to ensure that the user has filled in all the required fields
- The user is required to confirm their password by inputting it twice. The program must therefore check that the two passwords are the same before storing it in the database
- Users will use their email address as a username. I used regex to verify that the user input is a valid email address:

```
EmailCheck = re.match("[^z]+@[^z]+\.[^z]+", self.EmailTB.text())
```

`self.MyCursor` and `self.MyDB` aren't attributes of `CreateAccountWindow` because they are both created within the `TestConnection` method. They therefore have to be passed into `CreateAccountWindow` as arguments of the class.

```
# This object inherits from the QMainWindow class
# It displays the create account window, validates user inputs and adds the new account to
  the database
class CreateAccountWindow(QMainWindow):
    def __init__(self, RootPos, MyCursor, MyDB):
        super(CreateAccountWindow, self).__init__()

        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # The database cursor and connection have to be passed into the object as
          parameters
        self.MyCursor = MyCursor
        self.MyDB = MyDB

        self.CentralWidget = QWidget()
        self.setCentralWidget(self.CentralWidget)
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint))
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setWindowModality(Qt.ApplicationModal)

        # Position new window in top center of parent window
        self.move(int(RootPos.x() - ((250 - 400) / 2)), int(RootPos.y()))

        self.InitUI()
        self.show()
        self.setFixedSize(250, 158)

    def InitUI(self):
        # Initialise layouts
        self.CreateAccountVBL1 = QVBoxLayout()
        self.CreateAccountVBL2 = QVBoxLayout()

        self.HeaderLBL = QLabel("Let's Create Your Account!")
        self.HeaderLBL.setAlignment(Qt.AlignCenter)
        self.HeaderLBL.setStyleSheet("font-size: 14px; padding: 5px; background-color:
        #D5E8D4; border-left: 1px solid #5A5A5A; border-right: 1px solid #5A5A5A; border
        top: 1px solid #5A5A5A;")
        self.CreateAccountVBL1.addWidget(self.HeaderLBL)
        self.CreateAccountVBL1.setContentsMargins(0, 0, 0, 0)
        self.CreateAccountVBL2.setContentsMargins(10, 0, 10, 10)

        # Text box for user to input their email address
        self.EmailTB = QLineEdit()
        self.EmailTB.setPlaceholderText("Email Address")
        self.CreateAccountVBL2.addWidget(self.EmailTB)

        # Text box for user to input their password
        self.PasswordTB = QLineEdit()
        self.PasswordTB.setPlaceholderText("Password")
        self.PasswordTB.setEchoMode(QLineEdit.Password)
        self.CreateAccountVBL2.addWidget(self.PasswordTB)

        # Text box for user to input their password a second time to make sure user doesn't
          mistype password
        self.ConfirmPasswordTB = QLineEdit()
        self.ConfirmPasswordTB.setPlaceholderText("Confirm Password")
        self.ConfirmPasswordTB.setEchoMode(QLineEdit.Password)
        self.CreateAccountVBL2.addWidget(self.ConfirmPasswordTB)

        self.CreateAccountBTN = QPushButton("Create Account")
        self.CreateAccountBTN.clicked.connect(self.SubmitNewAccount)

        self.CancelBTN = QPushButton("Cancel")
        self.CancelBTN.clicked.connect(self.close)
```

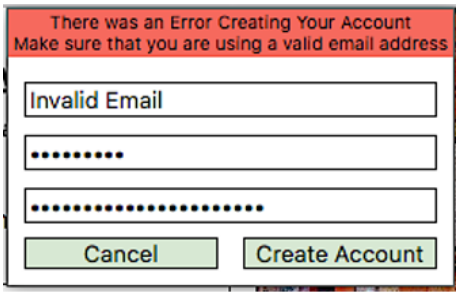
```
        # Combine layouts and add layouts to central widget
        self.CreateAccountHBL1 = QHBoxLayout()
        self.CreateAccountHBL1.addWidget(self.CancelBTN)
        self.CreateAccountHBL1.addWidget(self.CreateAccountBTN)
        self.CreateAccountVBL2.addLayout(self.CreateAccountHBL1)
        self.CreateAccountVBL1.addLayout(self.CreateAccountVBL2)
        self.CentralWidget.setLayout(self.CreateAccountVBL1)
```

This produces the following output:

`SubmitNewAccount` is a method of `CreateAccountWindow`. It is the function that will validate the values the user has inputted and write them to the database or display an error message.

```
# This function validates user inputs and then adds credentials to database or displays an
  error message accordingly
def SubmitNewAccount(self):

    # Use regex to verify input is a valid email address
    EmailCheck = re.match("[^z]+@[^z]+\.[^z]+", self.EmailTB.text())

    # If any text boxes are empty, the two password inputs don't match or the email
      address isn't valid, an error message is displayed
    if (self.ConfirmPasswordTB.text() != self.PasswordTB.text() or
        not self.ConfirmPasswordTB.text() or
            not self.PasswordTB.text() or
            not self.EmailTB.text() or
            EmailCheck == None):
        self.HeaderLBL.setStyleSheet("font-size: 10px; background-color: #FF6961; border
        left: 1px solid #5A5A5A; border-right: 1px solid #5A5A5A; border-top: 1px solid
        #5A5A5A;")
        self.HeaderLBL.setText("There was an Error Creating Your Account\nMake sure that
        you are using a valid email address")



    # If user inputs are valid, they are stored in the Users table
    else:
        # Hash password
        PasswordHash = hashlib.sha224(bytes(self.PasswordTB.text(), encoding = "utf
                       8")).hexdigest()
        self.MyCursor.execute("INSERT INTO Users (Email, PasswordHash) VALUES (%s, %s)",
                              (self.EmailTB.text(), PasswordHash))
        self.MyDB.commit()
        self.close()
```

With this function written, the program now displays an error message when the user makes a mistake:



If there are no problems with the inputted data, the new account is written to the Users table

Testing

Test	Expected Result	Actual Result
Login Window		
Input valid username and password combination	The user should be allowed into the application	The application isn't finished yet, so a message is printed to the python console instead as a temporary indicator that the login process was successful
Input valid username and invalid password	The user should not be allowed into the application and an error message should be displayed	This test worked as expected
Input valid username and valid password but credentials aren't paired together	The user should not be allowed into the application and an error message should be displayed	Initially, the program failed this test, so I went back and added a nested for loop to the `CheckCredentials` method to ensure that credentials are verified as pairs
Input invalid credentials	The user should not be allowed into the application and an error message should be displayed	This test worked as expected
Exceed 4 login attempts	The window should be disabled for an amount of time calculated by multiplying `self.Attempts` by `self.Magnitude` When the window is reenabled, `self.Magnitude` should be incremented so the window is disabled for longer next time	This test worked as expected I decided to give the disabled widgets a red border to better convey that the window was disabled to the user
Create Account Window		
Input valid email and two matching passwords	If the user inputs pass validation, they should be stored as a new entry in the Users table	This test worked as expected
Input invalid email	The regex check should return false if the email is invalid and an error message should be displayed	Initially, the regex check worked with email addresses which only had one domain level (.com, .org etc.) however it didn't work with addresses like .co.uk which have multiple domain levels. This was

		amended after testing and the function now works as expected
Input passwords which don't match	An error message should be displayed	This test worked as expected
Leave an empty input box	An error message should be displayed	This test worked as expected

During integration testing, I found that the login module did not work in conjunction with the SQL connection test module coded in iteration 2. This was because PyQt requires its main loop to be initialized using the function `sys.exit(App.exec())` before it can process any signals. The absence of this line of code meant that the login window would display but the user wouldn't be able to interact with it because the signal and slot handler in the main loop had not been initialized. I amended this issue by repurposing the `TestConnection` function as a method of the `LoginWindow` class rather than having it be an external method. Now, the main loop can be properly initialized and `TestConnection` is called in the constructor of `LoginWindow`. If the SQL connection can be established, `LoginWindow` can proceed with initialization, if there is an error connecting with the database, the error popup is displayed and the program proceeds accordingly.

Review

At this stage in development, I have coded a fully functioning login system. This system allows users to log into their accounts and create new accounts. After integration testing, the login system works with the SQL connection test coded in iteration 2. I have implemented security measures to protect against malicious attacks such as hiding user inputted passwords in textboxes and hashing all sensitive data stored in my database. The next stage in development is to create the Collections table and then develop the window which displays the collections belonging to the user who is currently logged in.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- A login system to secure data and prevent unauthorised access (5)
- Data should be stored in an efficient, concise and well-designed database (27)

Iteration 4 and 5; Display and Create Collections

I began this iteration by creating the SQL table for storing collections. `FK_Users_Collections` is a foreign key used to link entries in this table with accounts stored in the `Users` table. `TableName` stores the name of the SQL table which corresponds with that collection (see page 36 for more info).

```
CREATE TABLE Collections
  (
     PK_Collections       INTEGER(255) NOT NULL AUTO_INCREMENT PRIMARY KEY,
     CollectionName       VARCHAR(320),
     TableName            VARCHAR(320),
     FK_Users_Collections INTEGER(255)
  )
```

PK_Collections	CollectionName	TableName	FK_Users_Collections
34	Books	Table34	1
35	Records	Table35	1
41	Video Games	Table41	1
43	Films	Table43	1
44	Coins	Table44	1
45	Perfume	Table45	1
46	Comic Books	Table46	1
47	Action Figures	Table47	1
48	Stamps	Table48	1
49	Trading Cards	Table49	1

The next step was to program the window which displays all these collections. This window is initialised in the `CheckCredentials` method of the `LoginWindow` object when the user makes a successful login attempt. It will contain a status bar at the top containing a logout button and a label showing which user is currently logged in. Below this status bar is a `QScrollArea` populated with buttons for each collection. I have changed the design of this window from the initial design in the design section by having users navigate the scroll area using left and right arrow buttons rather than a horizontal scroll bar.

```
# This object stores the window which displays collections belonging to the logged in
  user and allows them to create new collections
class CollectionWindow(QMainWindow):
    def __init__(self, MyCursor, ActiveUserID, MyDB):
        super(CollectionWindow, self).__init__()

        # The database connection, cursor and the id of the user currently logged in
          are passed into this object as arguments
        self.ActiveUserID = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window widget and initialise main layouts
        self.CentralWidget = QWidget()
        self.setCentralWidget(self.CentralWidget)
        self.CollectionVBL1 = QVBoxLayout()
        self.CollectionVBL1.setContentsMargins(0, 0, 0, 0)
        self.CollectionGL1 = QGridLayout()
        self.CentralWidget.setLayout(self.CollectionVBL1)
```

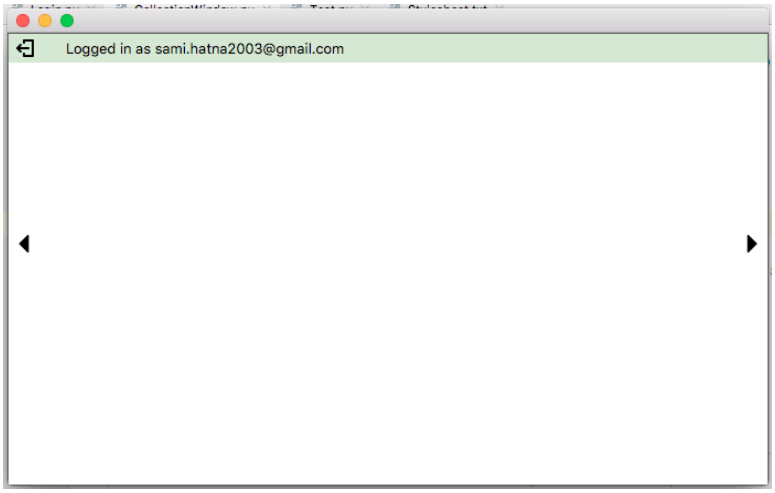
```
        # ContainerWidget contains the status bar displayed at the top of the window
        # The status bar conatins a logout button and a label outputting the currently
          logged in user
        self.ContainerWidget = QWidget()
        self.ContainerWidget.setObjectName("ContainerWidget")
        self.CollectionHBL1 = QHBoxLayout()
        self.CollectionHBL1.setContentsMargins(0, 0, 0, 0)
        LogoutBTN = QPushButton()
        LogoutBTN.clicked.connect(self.Logout)
        LogoutBTN.setIcon(QIcon("Resources/LogoutIcon.png"))
        LogoutBTN.setStyleSheet("width: 20px; padding: 5px; border-bottom: 0px;
        border-right: 0px;")
        self.MyCursor.execute("SELECT Email FROM Users WHERE PK_Users = " +
        str(self.ActiveUserID))
        SignedInUserLBL = QLabel("Logged in as "+str(self.MyCursor.fetchall()[0][0]))
        SignedInUserLBL.setStyleSheet("background-color: #D5E8D4; padding: 5px;
        border-top: 1px solid #5A5A5A;")
        self.CollectionHBL1.addWidget(LogoutBTN)
        self.CollectionHBL1.addWidget(SignedInUserLBL)
        self.CollectionHBL1.addStretch()
        self.ContainerWidget.setLayout(self.CollectionHBL1)
        self.CollectionVBL1.addWidget(self.ContainerWidget)

        self.InitUI()

        # Horizontal layout contains the navigation buttons and the scroll area used
          to display the collections
        self.CollectionHBL2 = QHBoxLayout()
        self.CollectionHBL2.setContentsMargins(10, 0, 10, 0)
        self.CollectionVBL1.addLayout(self.CollectionHBL2)

        # Initialise scroll area and set the widget which will be displayed within it
        self.CollectionScrollArea = QScrollArea()
        self.CollectionScrollArea.setObjectName("BorderlessWidget")
        # The user will navigate the scroll area using buttons, so the scroll bars are
          set to hidden
        self.CollectionScrollArea.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.CollectionScrollArea.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.CollectionScrollArea.setWidgetResizable(True)
        self.CollectionWidget = QWidget()
        self.CollectionWidget.setObjectName("BorderlessWidget")
        self.CollectionWidget.setLayout(self.CollectionGL1)
        self.CollectionScrollArea.setWidget(self.CollectionWidget)

        # LeftArrowBTN and RIghtArrowBTN are used to navigate left and right within
          the scroll are
        self.LeftArrowBTN = QPushButton()
        self.LeftArrowBTN.setIcon(QIcon("Resources/LeftArrow.png"))
        self.LeftArrowBTN.setStyleSheet("background-color: white; border: 0px;")
        self.LeftArrowBTN.clicked.connect(self.MoveLeft)
        self.LeftArrowBTN.setFixedSize(10, 300)

        self.RightArrowBTN = QPushButton()
        self.RightArrowBTN.setIcon(QIcon("Resources/RightArrow.png"))
        self.RightArrowBTN.setStyleSheet("background-color: white; border: 0px;")
        self.RightArrowBTN.clicked.connect(self.MoveRight)
        self.RightArrowBTN.setFixedSize(10, 90)

        # Add widgets to layout
        self.CollectionHBL2.addWidget(self.LeftArrowBTN)
        self.CollectionHBL2.addWidget(self.CollectionScrollArea)
        self.CollectionHBL2.addWidget(self.RightArrowBTN)
```

```
        self.setFixedSize(690, 410)
        self.CollectionVBL1.addStretch()

        # Display window
        self.show()
```

This short method is executed when the user presses the logout button:

```
# This function is called when the user wants to log out of the current account
# It creates a new instance of the LoginWindow class and closes the current window
def Logout(self):
    self.LoginWindowInstance = Login.LoginWindow()
    self.close()
```

When run, the code produces the following output. The white space in the middle is the `QScrollArea`, however it hasn't been populated with collection buttons yet. Coding the function that carries this out is the next step.



Before I coded this function, however, I created a subclass of the `QAbstractButton` object called `CollectionButton`. These are the buttons that will represent each collection in this window.

```
# Subclass of QPushButton used to create buttons for each collection
class CollectionButton(QPushButton):
    # This signal is emitted when the user selects Delete Collection in the widget's
      context menu
    DeleteCollectionSignal = pyqtSignal(int)
    # The only argument passed into this object is the name and primary key of the
      collection it is supposed to represent
    def __init__(self, Data):
        super(CollectionButton, self).__init__()
        self.Data = Data
        self.setText(self.Data[1])
        # Properties are values attached to a widget, this will be used when the user
          wants to open or delete the collection by clicking on the button
        self.setProperty("Key", self.Data[0])
        self.setObjectName("CollectionButton")
        self.setFixedWidth(140)
```

`self.InitUI` works by clearing the `QScrollArea` of widgets every time it is executed. It then fetches all the entries in the `Collections` table and creates an instance of `CollectionButton` for each collection and adds this button to the scroll area. It also creates a button for creating new collections. The scroll area uses a grid layout, so the buttons are displayed as a grid.

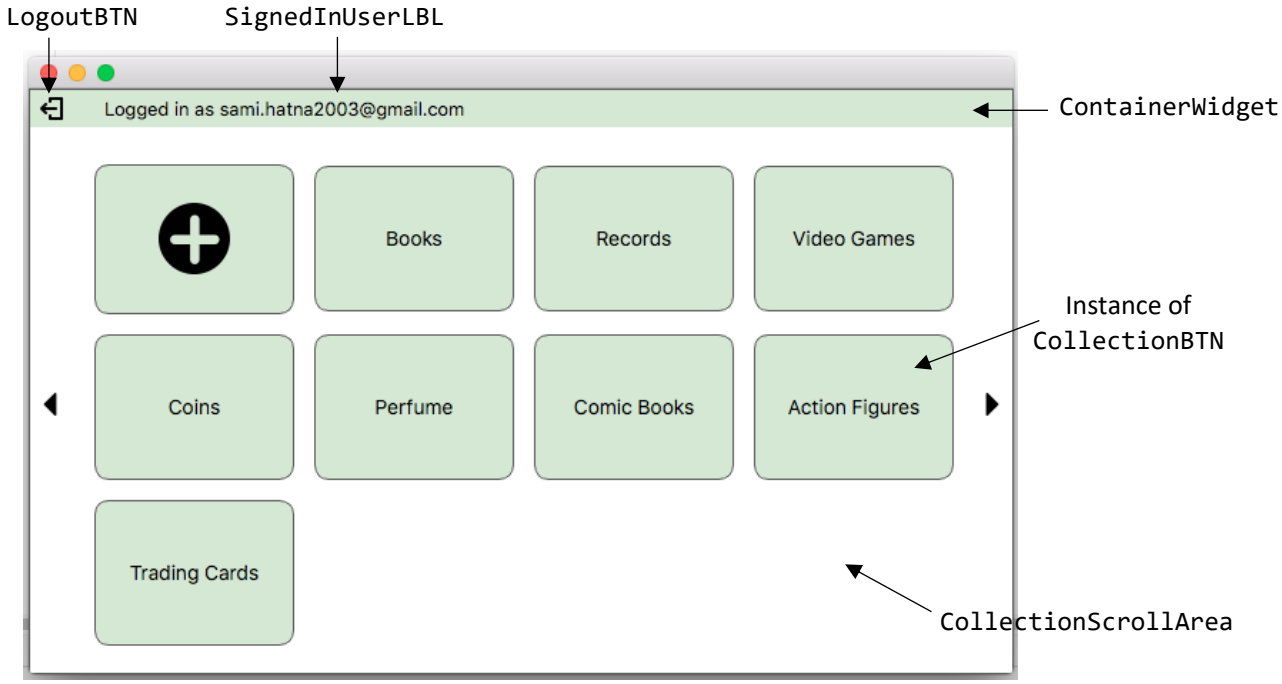
```
# This function populates the scroll area with buttons for each collection and a
  button for creating collections
def InitUI(self):
    # Remove any widgets already in the layout before repopulating it
    for i in reversed(range(self.CollectionGL1.count())):
        self.CollectionGL1.itemAt(i).widget().setParent(None)

    # Initialise create collection button and add to grid layout
    self.CreateCollectionBTN = QToolButton()
    self.CreateCollectionBTN.setIcon(QIcon("Resources/AddItemIcon.png"))
    self.CreateCollectionBTN.setIconSize(QSize(50, 50))
    self.CreateCollectionBTN.setToolTip("Create a new collection")
    self.CreateCollectionBTN.setObjectName("CollectionButton")
    self.CreateCollectionBTN.setFixedWidth(140)
    self.CreateCollectionBTN.clicked.connect(self.CreateNewCollection)
    self.CollectionGL1.addWidget(self.CreateCollectionBTN, 0, 0)

    # Retrieve all collections that belong to the currently logged in user from the
      database
    self.MyCursor.execute("SELECT * FROM Collections WHERE FK_Users_Collections = " +
    str(self.ActiveUserID))
    self.MyResult1 = self.MyCursor.fetchall()

    # The following code iterates through the result of the database query and creates
      a button for each collection which is added to the grid layout
    # Counters are used to keep track of which row and column the program is in
    Populated = False
    RowCounter = 0
    ResultCounter = 0
    ColumnCounter = 1
    self.CollectionBTNArray = []
    Division = math.ceil(len(self.MyResult1) / 3)
    while not Populated:
        if ResultCounter == len(self.MyResult1):
            Populated = True
        elif ColumnCounter <= Division:
            self.CollectionBTNArray.append(CollectionButton(Data
            self.MyResult1[ResultCounter]))
            self.CollectionBTNArray[ResultCounter].DeleteCollectionSignal.connect(
                    self.DeleteCollectionSlot
            )
            self.CollectionGL1.addWidget(
                    self.CollectionBTNArray[ResultCounter], RowCounter, ColumnCounter
            )
            ColumnCounter += 1
            ResultCounter += 1
        else:
            RowCounter += 1
            ColumnCounter = 0
```

With the window populated, I went on to write two functions for the left and right buttons to allow users to navigate left and right.

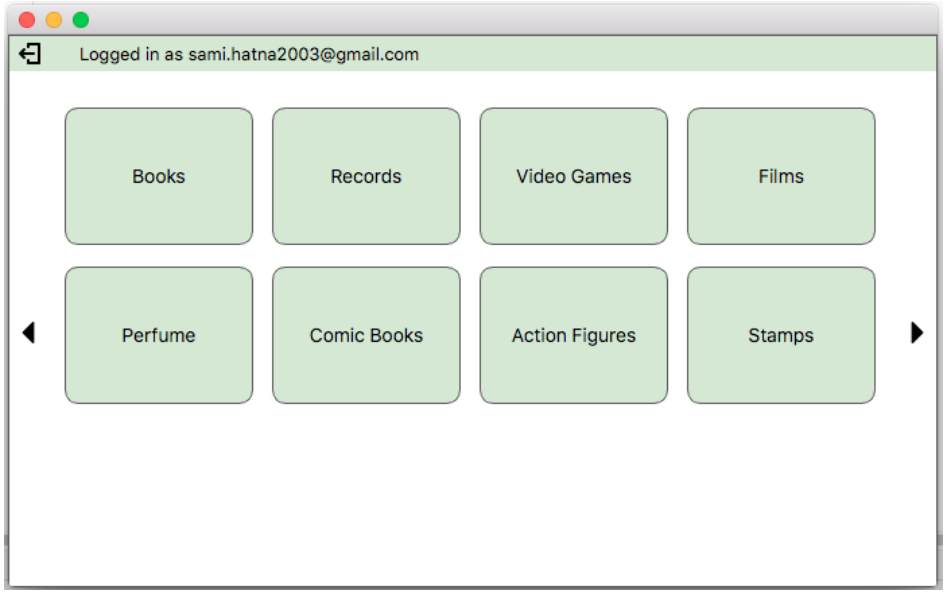
```
# MoveLeft and MoveRight are executed when the left and right navigation buttons are
  pressed
# They simply move the scrollbar along by 160 pixels in the appropriate direction
def MoveLeft(self):
    self.CollectionScrollArea.horizontalScrollBar().setValue(
        self.CollectionScrollArea.horizontalScrollBar().value() - 160)

def MoveRight(self):
```

```
    self.CollectionScrollArea.horizontalScrollBar().setValue(
        self.CollectionScrollArea.horizontalScrollBar().value() + 160)
```

The code now produces this output:



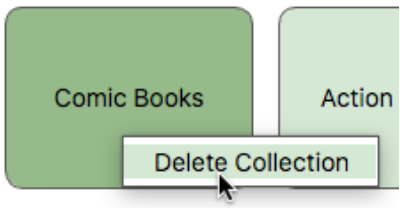
If the user presses one of the arrow buttons, the display will be shifted in the appropriate direction:



To allow users to delete collections I had to make use of PyQt's signals and slots system. Signals allow different classes/windows to interact with each other. Each instance of `CollectionButton` has a context menu with the option to delete that collection. When the user selects this function, the signal `DeleteCollectionSignal` is emitted. You can pass parameters into signals which will be emitted alongside the signal. In this case, I passed the primary key of the collection being deleted into this signal. When the signal is emitted, it will be received by the event handler of the class `CollectionWindow` and the method/slot `DeleteCollectionSlot` will be executed. `DeleteCollectionSlot` carries out the actual process of deleting the collection from the SQL database.

The `ContextMenuEvent` of `CollectionButton`:

```
# This event occurs when the user right clicks on a button
# It create a context menu with the option to delete a collection
def contextMenuEvent(self, event):
    PropertyWidgetContextMenu = QMenu(self)
    DeleteAction = PropertyWidgetContextMenu.addAction("Delete Collection")
    Action = PropertyWidgetContextMenu.exec_(self.mapToGlobal(event.pos()))
    # If the user chooses to delete the collection, DeleteCollectionSignal emits
    if Action == DeleteAction:
        self.setParent(None)
        self.DeleteCollectionSignal.emit(self.Data[0])
```



The method of `CollectionWindow` which is executed when the signal is emitted:

```
# This slot is executed when this class receives the DeleteCollection signal from the
  CollectionButton object
# Its purpose is to delete the collection and its corresponding table from the
  database and refresh the window
def DeleteCollectionSlot(self, IDForDeletion):
    self.MyCursor.execute("DROP TABLE Table" + str(IDForDeletion))
    self.MyCursor.execute("DELETE FROM Collections WHERE PK_Collections = " +
    str(IDForDeletion))
    self.MyDB.commit()
    self.InitUI()
```

The next phase of this iteration is programming the collection creation system. When the user presses the create collection button, the following function is executed:

```
def CreateNewCollection(self):
    # Create an instance of the AddCollectionWindow object
    self.AddCollectionWindowInstance = AddCollectionWindow(
                                            MyCursor = self.MyCursor,
                                            ActiveUserID = self.ActiveUserID,
                                            MyDB = self.MyDB,
                                            RootPos = self.geometry())
    self.AddCollectionWindowInstance.CreateCollectionSignal.connect(self.InitUI)
```

The add collection window itself consists of a vertical box layout containing a text input box for the name of the collection, a scroll area containing widgets for creating collection properties, and action buttons. The primary key of the logged in user and the SQL cursor are passed in as arguments.

```
# This object creates the window which allows users to create new collections
class AddCollectionWindow(QMainWindow):
    # This signal is emitted to inform the CollectionWindow class that a new
      collection has been added to the database and the display needs to be refreshed
    CreateCollectionSignal = pyqtSignal()
    def __init__(self, ActiveUserID, MyCursor, MyDB, RootPos):
        super(AddCollectionWindow, self).__init__()

        self.ActiveUserID = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB
```

```
        # Instances of PropertyWidget are created dynamically, so they will be stored
          in this array
        self.PropertyWidgetArray = []
        # Counter keeps track of how many instances of PropertyWidget exist
        self.PropertyWidgetCounter = 0

        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint))
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setWindowModality(Qt.ApplicationModal)

        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set central widget and initialise vertical box layout
        self.CentralWidget = QWidget()
        self.setCentralWidget(self.CentralWidget)
        self.CreateCollectionVBL1 = QVBoxLayout()
        self.CentralWidget.setLayout(self.CreateCollectionVBL1)

        self.InitUI()

        self.setFixedSize(450, 400)
        self.move(int(RootPos.x() + 112), int(RootPos.y()))

        self.show()

    def InitUI(self):
        HeaderLBL = QLabel("Create New Collection")
        HeaderLBL.setStyleSheet("font-size: 16px;")
        HeaderLBL.setAlignment(Qt.AlignCenter)
        self.CreateCollectionVBL1.addWidget(HeaderLBL)

        # The first text input box is for inputting the name of the collection
        self.CreateCollectionHBL1 = QHBoxLayout()
        self.CreateCollectionHBL1.addWidget(QLabel("Name"))
        self.NameTB = QLineEdit()
        self.CreateCollectionHBL1.addWidget(self.NameTB)
        self.CreateCollectionVBL1.addLayout(self.CreateCollectionHBL1)

        # The user creates collections by adding named properties, these property
          widgets are displayed in a scroll area
        self.CreateCollectionScrollArea = QScrollArea()
        self.CreateCollectionScrollArea.setWidgetResizable(True)
        self.ScrollAreaWidget = QWidget()
        self.ScrollAreaWidget.setObjectName("BorderlessWidget")
        self.ScrollAreaLayout = QVBoxLayout()
        self.ScrollAreaLayout.setAlignment(Qt.AlignTop)
        self.ScrollAreaWidget.setLayout(self.ScrollAreaLayout)
        self.CreateCollectionScrollArea.setWidget(self.ScrollAreaWidget)
        self.CreateCollectionVBL1.addWidget(self.CreateCollectionScrollArea)

        # Button for adding a new property
        self.AddNewPropertyBTN = QPushButton("Add New Property")
        self.AddNewPropertyBTN.clicked.connect(self.AddNewProperty)
        self.CreateCollectionVBL1.addWidget(self.AddNewPropertyBTN)


        # Two buttons, one for cancelling the operation and one for submitting the new
          collections
        self.CreateCollectionHBL2 = QHBoxLayout()
        self.CancelBTN = QPushButton("Cancel")
```

```
        self.CancelBTN.clicked.connect(self.close)
        self.CreateCollectionHBL2.addWidget(self.CancelBTN)
        self.CreateCollectionBTN = QPushButton("Create Collection")
        self.CreateCollectionBTN.clicked.connect(self.CreateCollection)
        self.CreateCollectionHBL2.addWidget(self.CreateCollectionBTN)
        self.CreateCollectionVBL1.addLayout(self.CreateCollectionHBL2)
```

Every time the user presses `AddNewPropertyBTN`, an instance of `PropertyWidget` must be created and added to `CreateCollectionScrollArea`. `PropertyWidget` is a subclass of `QWidget` and consists of an input box for the name of the property and a combobox for the user to select the datatype for that property (text or number).

```
# Subclass of QWidget
# Each property widget consists of a text input for the name of the property and a
  combobox for its datatype
class PropertyWidget(QWidget):
    # This signal is emitted when the user wants to delete a property
    DeletePropertySignal = pyqtSignal(QObject)
    def __init__(self, Number):
        super(PropertyWidget, self).__init__()

        self.setObjectName("BorderlessWidget")
        # Initialise widget layout
        self.PropertyWidgetHBL1 = QHBoxLayout()
        self.setLayout(self.PropertyWidgetHBL1)

        PropertyLBL = QLabel("Property " + str(Number))
        self.PropertyWidgetHBL1.addWidget(PropertyLBL)

        # Text box for inputting property name
        self.PropertyNameTB = QLineEdit()
        self.PropertyWidgetHBL1.addWidget(self.PropertyNameTB)

        TypeLBL = QLabel("Type")
        self.PropertyWidgetHBL1.addWidget(TypeLBL)

        # Combo box for specifiying if property datatype is text or numerical
        self.TypeCB = QComboBox()
        self.TypeCB.addItems(["Text", "Number"])
        self.PropertyWidgetHBL1.addWidget(self.TypeCB)

    # Event executed when user right clicks on property widget
    def contextMenuEvent(self, event):
        PropertyWidgetContextMenu = QMenu(self)
        DeleteAction = PropertyWidgetContextMenu.addAction("Delete Property")
        Action = PropertyWidgetContextMenu.exec_(self.mapToGlobal(event.pos()))
        # If user clicks Delete Property, emit signal
        if Action == DeleteAction:
            self.setParent(None)
            self.DeletePropertySignal.emit(self)
```

This is what an instance of `PropertyWidget` looks like:

This is the function which is executed when the user presses `AddNewPropertyBTN`:

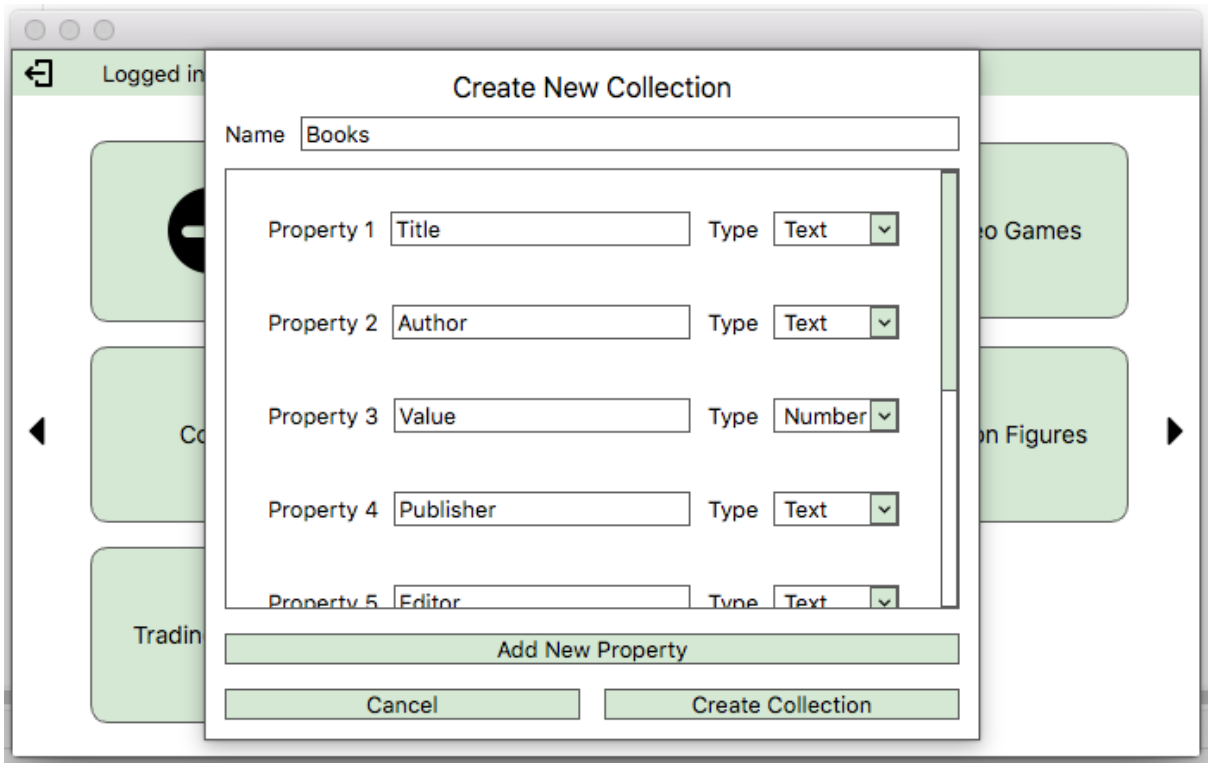
```
# This function adds a new property widget to the scroll area
def AddNewProperty(self):
    # Create new instance of PropertyWidget and add to list
    PropertyWidgetArray.append(
```

```
                PropertyWidget(Number=self.PropertyWidgetCounter+ 1)
                                )
    # Add new PropertyWidget to scroll area
    self.ScrollAreaLayout.addWidget(
                self.PropertyWidgetArray[self.PropertyWidgetCounter]
                                )
    PropertyWidgetArray[PropertyWidgetCounter].DeletePropertySignal.connect(
                self.DeletePropertySlot
                                )
    self.PropertyWidgetCounter += 1
```

`PropertyWidget` also makes use of signals and slots to facilitate deleting instances of itself from the scroll area. When `DeletePropertySignal` is emitted, this function is executed in the `AddCollectionWindow` class:

```
# This function is executed when the Delete Property signal is received
# It deletes the specified property widget
def DeletePropertySlot(self, Sender):
    self.PropertyWidgetArray.remove(Sender)
    self.PropertyWidgetCounter -= 1
    print(self.PropertyWidgetArray)
```

`AddCollectionWindow` output:



The next method that needs programming is the function which is executed when the user presses `CreateCollectionBTN`. `CreateCollection` begins by inserting a new entry into the Collections table. It then fetches all the properties for that collection and their corresponding data types and creates a new table with those properties as fields in the table. The name of this table is devised by fetching the primary key of the entry for that collection in the Collections table and adding the prefix “Table”. Each table will also automatically have the columns Thumbnail (Longblob), Rating (Integer out of five) and Rare (Boolean). After the new table has been created, the entry in the Collections table is updated to store the new table name under the column TableName. By using the primary key in the table name, we are ensuring that each table name is unique.

For example, if I were creating a book collection with the properties title, author and publisher, the program would begin by inserting this entry into the Collections table (for this example I am logged in as the user with primary key 1):

PK_Collections	CollectionName	TableName	FK_Users_Collections
34	Books	NULL	1

The program will then create a new table called “Table34” (note that the number in the table name is the same as the primary key in the Collections table) which uses the properties the user has inputted in `CreateCollectionWindow` as column names:

Table34						
PK_Table34	Title	Author	Publisher	Rare	Rating	Thumbnail

The final step is to update the Collections entry and add the name of the new table to the TableName column:

PK_Collections	CollectionName	TableName	FK_Users_Collections
34	Books	Table34	1

In terms of validation, I implemented a presence check to ensure that no user input fields were left blank before writing to the database. All user inputs also need to be typecast as strings before they are formatted into any SQL commands

This is how I implemented this function programmatically:

```
# This function stores a new entry in the Collections table and create the
  collection's corresponding table
# See database design section of report for more info (page 36)
def CreateCollection(self):
    # Presence check to ensure all fields are filled in
    EmptyString = False
    EmptyPropertyWidgetIndexes = []
    for x in range(0, len(self.PropertyWidgetArray)):
        if self.PropertyWidgetArray[x].PropertyNameTB.text() == "":
            EmptyString = True
            EmptyPropertyWidgetIndexes.append(x)

    # If presence check flags an unfilled field, execute error function
    if EmptyString or self.NameTB.text() == "":
        self.CreateCollectionError(EmptyPropertyWidgetIndexes)
    else:
        # Get name of new collections
        Name = self.NameTB.text()
        # Insert new entry into Collections table
        self.MyCursor.execute("INSERT INTO Collections (CollectionName,
        FK_Users_Collections) VALUES ('{0}', '{1}')".format(Name, self.ActiveUserID))
        self.MyCursor.execute("SELECT MAX(PK_Collections) FROM Collections")
        self.MyResult1 = self.MyCursor.fetchall()

        # New table name consists of "table" + primary key of the entry that has just
          been stored in Collections table
        SQLStatement = "CREATE TABLE Table" + str(self.MyResult1[0][0]) + " (PK_Table"
        + str(self.MyResult1[0][0]) + " INTEGER(255) NOT NULL AUTO_INCREMENT
        PRIMARY KEY, "
        # Piece together SQL statement for creation of new table
        for x in range(0, len(self.PropertyWidgetArray)):
            # Get property name and datatype and use as column in new table
            PropertyName = self.PropertyWidgetArray[x].PropertyNameTB.text()
            DataType = self.PropertyWidgetArray[x].TypeCB.currentText()
```

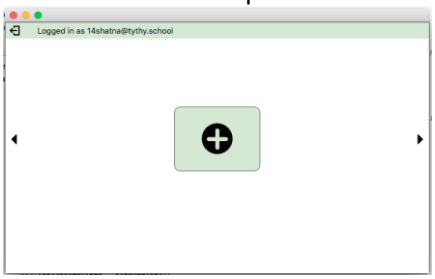
```
            if DataType == "Number":
                DataType = "Integer"
                SQLStatement = SQLStatement + "`" + PropertyName + "` " + DataType +
                "(255), "
            else:
                DataType = "Varchar"
                SQLStatement = SQLStatement + "`" + PropertyName + "` " + DataType +
                "(500), "
        # All collections will have a Rare column, a Rating column and a Thumbnail
          column
        SQLStatement = SQLStatement + " Rare BOOLEAN, Rating INTEGER(20), Thumbnail
        LONGBLOB)"
        self.MyCursor.execute(SQLStatement)
        # Add new table name to the entry in Collections
        self.MyCursor.execute("UPDATE Collections SET TableName = 'Table" +
        str(self.MyResult1[0][0]) + "' WHERE PK_Collections = " +
        str(self.MyResult1[0][0]))
        # Save changes to database
        self.MyDB.commit()
        # Emit signal to inform CollectionWindow that a new collection has been
          created and it needs to be refreshed
        self.CreateCollectionSignal.emit()
        self.close()
```

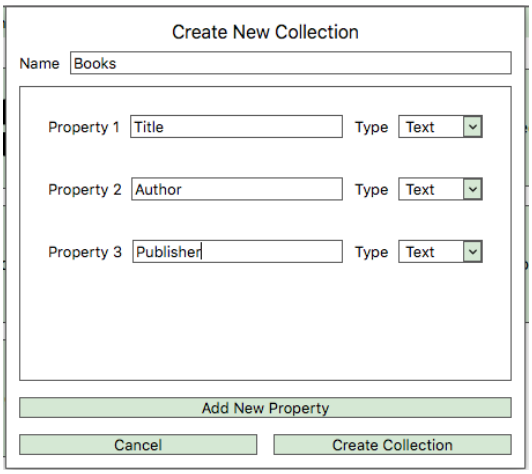
This is the function which is executed when the user leaves an input box empty. It simply highlights all the empty input boxes in red.

```
# Function is executed when a field has been left empty
# It takes a list containing the indexes of the empty widgets as a parameter and sets
  the border of these widgets to red
def CreateCollectionError(self, OffendingWidgets):
    print(OffendingWidgets)
    for x in OffendingWidgets:
        self.PropertyWidgetArray[x].PropertyNameTB.setStyleSheet("border: 1px solid
        #FF0000")
```

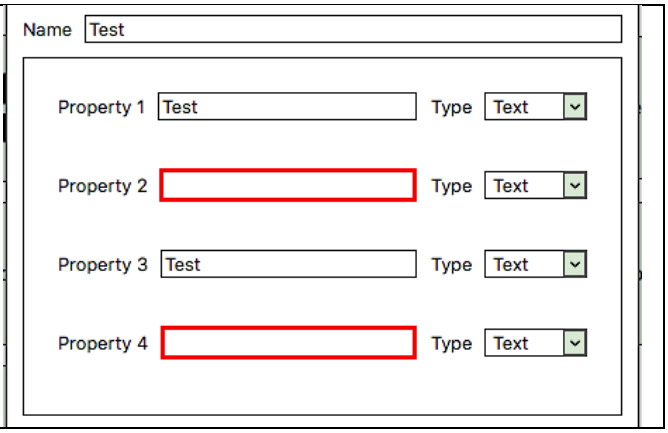
If a collection is successfully created, the collection emits `CreateCollectionSignal`. This signal is connected to the method `InitUI` which refreshes the window in order to display the new collection.

Testing

Test	Expected Result	Actual Result
Collections Window		
Login with an account that has no collections	The window should only display the create collection button	The test works as expected: 
Delete Collection	The entry into the Collections table and the corresponding table should be deleted	I used breakpoints and debugging mode to test this feature. The program first deletes the table and then deletes the entry in Collections.

<table>
<tr><td>Are there any instances when a collection table exists but an entry into the Collections table does not or vice versa?</td><td>This should not be the case as otherwise; the referential integrity of the database will be compromised</td><td>Luckily, because there is only ever one client accessing the collections belonging to a user at a time, I have not come across this issue during testing. If I decided to allow multiple users to access the same collections, I would have to implement some sort of record locking procedure to prevent against this</td></tr>
<tr><td>Test logout function</td><td>The current window should be closed and the login window should be redisplayed</td><td>This function works as expected. However, the first time I tested it I had to add a line of code which ensured that the variable ActiveUserID was reset when the user logs out</td></tr>
<tr><td>Test delete collection</td><td>The entry for that collection in the Collections table and the corresponding table should be deleted</td><td>This function works as expected. Checking in my DBMS proves that both the actual table and the entry are deleted</td></tr>
<tr><td colspan="3">Create Collections</td></tr>
<tr><td>Create a new collection</td><td>The program should work as outlined at the top of page 88</td><td>This is the collection I submitted for creation:
 

This is the entry in the Collections table which was created:
<table>
<tr><th>PK_Collections</th><th>CollectionName</th><th>TableName</th><th>FK_Use</th></tr>
<tr><td>34</td><td>Books</td><td>Table34</td><td>1</td></tr>
</table>
And this is the table for that collection which was created:
<table>
<tr><th>Title</th><th>Author</th><th>Publisher</th><th>Rare</th><th>Rating</th><th>Thumbnail</th></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table></td></tr>
<tr><td>Submit collection with blank field</td><td>If the program finds any empty input boxes, it should change their outlines to red. Nothing</td><td>There was initially a logic error concerning the array used to store the indexes of empty input boxes. After this was remedied, the following was outputted when an input box was left blank:</td></tr>
</table>

	should be written to the database	

This iteration passed the integration testing phase with no faults as it was simply a matter of creating an instance of `CollectionWindow` in the `LoginWindow` class when the user makes a successful log in attempts. As detailed in the above table, the only fault I came across during integration testing was ensuring that the value of `ActiveUserID` is reset when the user logs out of their account.

Review

At this stage in development, I have built the window that displays the collections belonging to the currently logged in user. I have also programmed the processes which allow users to delete collections and create new ones. The use of navigational buttons rather than a scrollbar is one of the first instances when the UI design of one of my windows has been altered from the prototype in the design stage. I believe this change was a welcome one as it improves the accessibility and simplicity of the user interface. The next stage in development is developing the OpenCollection class which will be executed when the user clicks on one of the collection buttons.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- The ability to create your own collections with user specified column names (6)

End-User Feedback

At this point in development, I wanted to get some feedback on the work I had done so far, specifically concerning how easy the process of creating and deleting collections was. During the feedback on the design stage (page 33), Jonathan was the end-user most concerned with accessibility and usability, so I sent the source code for him to test alongside instructions on installing the required libraries and running the scripts. This is the email he sent me:

> *Sami,*
> *Hope you're doing well. This app looks very similar to the prototypes you sent me! I must say I love the colour scheme. I like the system of adding and deleting properties, I think it makes creating collections very intuitive. The only problem I came across was that text boxes would sometimes have red borders and the window wouldn't go away, is this an error?*
> *John*

I'm glad that Jonathan believes the collection creation process was simple and user friendly. I have remedied the confusion surrounding the red borders by adding an error message to the window which informs the user of exactly what mistake they have made. Another change I have made to improve accessibility is adding tooltips to buttons, so users know exactly what each button does.

Iteration 6; Open Collection

This iteration involves programming the window that will display the collection that the user has opened; populating the window with items from the collection; coding the simple search function and the slot which displays an item's thumbnail when the user selects it. I began this iteration by initializing all the widgets used in the window and arranging them in layouts. Broadly, the window is made up of a central vertical box layout and a status bar running horizontally along the bottom of the window. The central vertical layout contains three horizontal sub-layouts: one for the toolbar, one for the simple search bar and the order combobox and one containing the table widget and thumbnail groupbox. The window will also have a menu bar with a File menu. At this point in development, the only option under the file menu will be a button for closing the current collection and returning to the previous window. The class `OpenCollectionWindow` takes the database cursor and the primary keys of the active user and the open collection as arguments.

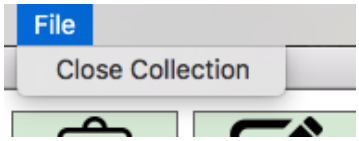
The first function of `OpenCollectionWindow` is the `init` method which initializes the window, stores the parameters as global variables, fetches the table name of the current collection and creates the menu bar:

```
# This object is the window which displays the collection the user has chosen to open
class OpenCollectionWindow(QMainWindow):
    # The window takes the cursor, the database connection and the primary key of the
      logged in user as arguments
    # It also takes the primary key of the collection which is currently open
    def __init__(self, MyCursor, ActiveUserID, MyDB, OpenCollectionID):
        super(OpenCollectionWindow, self).__init__()

        self.ActiveUserID = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.OpenCollectionID = OpenCollectionID
        self.MyCursor.execute("SELECT TableName FROM Collections WHERE PK_Collections
        = " + str(self.OpenCollectionID))
        # OpenCollectionTable is the name of the collection which is currently open
        # It is obtained by joining the string "TABLE" with the primary key of the
          collection in the Collections table
        self.OpenCollectionTable = self.MyCursor.fetchall()[0][0]

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window's central widget and initialise main layout
        self.CentralWidget = QWidget()
        self.OpenCollectionVBL1 = QVBoxLayout()
        self.CentralWidget.setLayout(self.OpenCollectionVBL1)
        self.setCentralWidget(self.CentralWidget)

        self.InitUI()

        # Initialise menu bar
        self.MainMenu = QMenuBar()
        # Add file menu to menubar
        self.FileMenu = self.MainMenu.addMenu(" &File")
        # Create action for closing current collection window and going back to
          CollectionWindow
        self.CloseCollectionAction = QAction("Close Collection", self)
        # Bind action to function
        self.CloseCollectionAction.triggered.connect(self.CloseCollection)
        # Add action to file menu
        self.FileMenu.addAction(self.CloseCollectionAction)
```

```
        # Display window at maximum size
        self.showMaximized()
        self.show()
```

`CloseCollectionAction` is bound to the function `CloseCollection`, which closes the current window and returns the user to the window which displays all their collections.

```
# This function is executed when the user selects Close Collection from the file menu
# It closes the current window and reopens the collection window
def CloseCollection(self):
    self.CollectionWindowInstance = CollectionWindow.CollectionWindow(
                                            MyCursor = self.MyCursor,
                                            MyDB = self.MyDB,
                                            ActiveUserID = self.ActiveUserID
                                                    )
    self.close()
```



Within the `init` method, the function `InitUI()` is called. The purpose of this function is to populate the window shell created in the `init` method with widgets.

```
# Initialise UI elements
def InitUI(self):
    # InitToolbar is the function which creates the window toolbar
    self.InitToolbar()

    # Add 20 pixels of space between toolbar and search bar
    self.OpenCollectionVBL1.addSpacing(20)

    # SearchBarHBL is the layout which stores the search bar and the sorting menu
    self.SearchBarHBL = QHBoxLayout()
    self.SearchBarHBL.addWidget(QLabel("Search:"))
    # SearchBarTB is the text box in which the user inputs search terms
    self.SearchBarTB = QLineEdit()
    self.SearchBarTB.setFixedWidth(150)
    # When the text in a QLineEdit changes, it emits the textChanged signal
    # I have binded this signal to the function SimpleSearch so that whenever the user
      changes the text in the search bar, the function will be executed
    self.SearchBarTB.textChanged.connect(self.SimpleSearch)
    self.SearchBarHBL.addWidget(self.SearchBarTB)
    self.SearchBarHBL.addSpacing(50)

    self.SearchBarHBL.addWidget(QLabel("Order:"))
    # OrderCB is a combobox which allows user to choose which order they would like to
      display their collection in
    self.OrderCB = QComboBox()
    self.OrderCB.addItems(["Date Added", "Ascending", "Descending"])
    self.OrderCB.setFixedWidth(130)
    # When the user selects a new item in the combobox, it emits a signal, I have
      bound this signal to the function PopulateTable
    # This means that if the user selects a different order for items, the table will
      be repopulated in that order
    self.OrderCB.currentTextChanged.connect(self.PopulateTable)
    self.SearchBarHBL.addWidget(self.OrderCB)

    # Add horizontal box layout to main layout
    self.OpenCollectionVBL1.addLayout(self.SearchBarHBL)
```

```
        self.SearchBarHBL.addStretch()

        # StatusBar displays a horizontal bar along the bottom of the window for
          presenting status information
        self.StatusBar = QStatusBar()
        self.setStatusBar(self.StatusBar)
        # StatusLBL will be used to convey information to the user such as collection size
          or when processes are complete
        self.StatusLBL = QLabel("")
        self.StatusLBL.setStyleSheet("padding-left: 5px")
        self.StatusBar.addWidget(self.StatusLBL, 1)
        self.CreditLBL = QLabel("Anthology v1.0 created by Sami Hatna")
        self.CreditLBL.setStyleSheet("padding-right: 5px")
        self.StatusBar.addPermanentWidget(self.CreditLBL)

        # This horizontal box layout displays the table in which the collection is
          presented and the group box for displaying item thumbnails
        self.OpenCollectionHBL1 = QHBoxLayout()
        # CollectionTable is the table which displays collection items
        self.CollectionTable = QTableWidget()
        self.CollectionTable.setShowGrid(False)
        # Disable table editing, if users want to edit items they have to do it using the
          edit toolbutton
        self.CollectionTable.setEditTriggers(QAbstractItemView.NoEditTriggers)
        # The first column is hidden because it stores the primary key of each item
        self.CollectionTable.setColumnHidden(0, True)
        self.CollectionTable.horizontalHeader().setStretchLastSection(True)
        # Users can only select rows rather than individual cells
        self.CollectionTable.setSelectionBehavior(QAbstractItemView.SelectRows)
        self.CollectionTable.verticalHeader().setVisible(False)
        # When the user selects a new row in the table, DisplayImage is executed to
          display the thumbnail for that item in ThumbnailGroupBox
        self.CollectionTable.itemSelectionChanged.connect(self.DisplayImage)
        self.OpenCollectionHBL1.addWidget(self.CollectionTable)

        self.PopulateTable()

        # ThumbnailGroupBox displays the thumbnail and metadata of the currently selected
          item
        self.ThumbnailGroupBox = QGroupBox()
        self.ThumbnailGroupBox.setFixedWidth(150)
        # Set groupbox layout
        self.ThumbnailLayout = QVBoxLayout()
        self.ThumbnailLayout.setContentsMargins(5, 5, 5, 5)
        self.ThumbnailGroupBox.setLayout(self.ThumbnailLayout)
        # ThumbnailImageLabel is the label which will display the thumbnail of the
          selected item
        # Images in PyQt are displayed by loading a QPixmap onto a QLabel
        self.ThumbnailImageLabel = QLabel()
        self.ThumbnailImageLabel.setAlignment(Qt.AlignCenter)
        self.ThumbnailLayout.addWidget(self.ThumbnailImageLabel)
        # This list stores the labels which will be used to display info about the
          currently selected item
        # A label is created for each column in the SQL table and used to display its
         corresponding contents
        self.ThumbnailLabelsArray = []
        # Get the number of columns in the SQL table
        self.MyCursor.execute("SELECT COUNT(*) FROM information_schema.columns WHERE
        table_name = '" + self.OpenCollectionTable + "'")
        # For each column in the table, create a label (excluding the Rare, Rating and
          Thumbnail columns)
        for x in range(1, self.MyCursor.fetchall()[0][0] - 3):
```

```
            Temp = QLabel("")
            Temp.setWordWrap(True)
            self.ThumbnailLayout.addWidget(Temp)
            # Because there is a variable amount of labels, they are each appended to this
              list after they are created
            self.ThumbnailLabelsArray.append(Temp)
        self.ThumbnailLayout.addStretch()
        # Add groupbox to horizontal box layout
        self.OpenCollectionHBL1.addWidget(self.ThumbnailGroupBox)
```

At the start of `InitUI`, the method `InitToolbar` is called. `InitToolbar` works by creating a horizontal box layout and adding each button to that layout. The layout is then added at the top of the main vertical box layout. The first step in programming `InitToolbar` was creating a class called `ToolBarBTN`, which inherits from `QToolButton`. Each button in the toolbar will be an instance of this object.

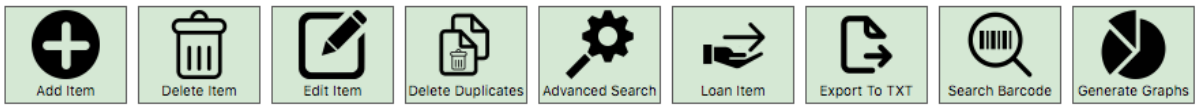
```
# Inherits from QToolButton
# Used to create toolbar buttons in InitToolbar()
class ToolBarBTN(QToolButton):
    # Takes the arguments Text and IconPath
    # IconPath is the file path for the icon the button displays
    def __init__(self, Text, IconPath):
        super(ToolBarBTN, self).__init__()
        self.setText(Text)
        self.setIcon(QIcon(IconPath))
        self.setIconSize(QSize(50, 50))
        self.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)
        self.setFixedWidth(90)
```

Once this object was created, I wrote a simple function which created an instance of `ToolBarBTN` for each button and added it to the layout.

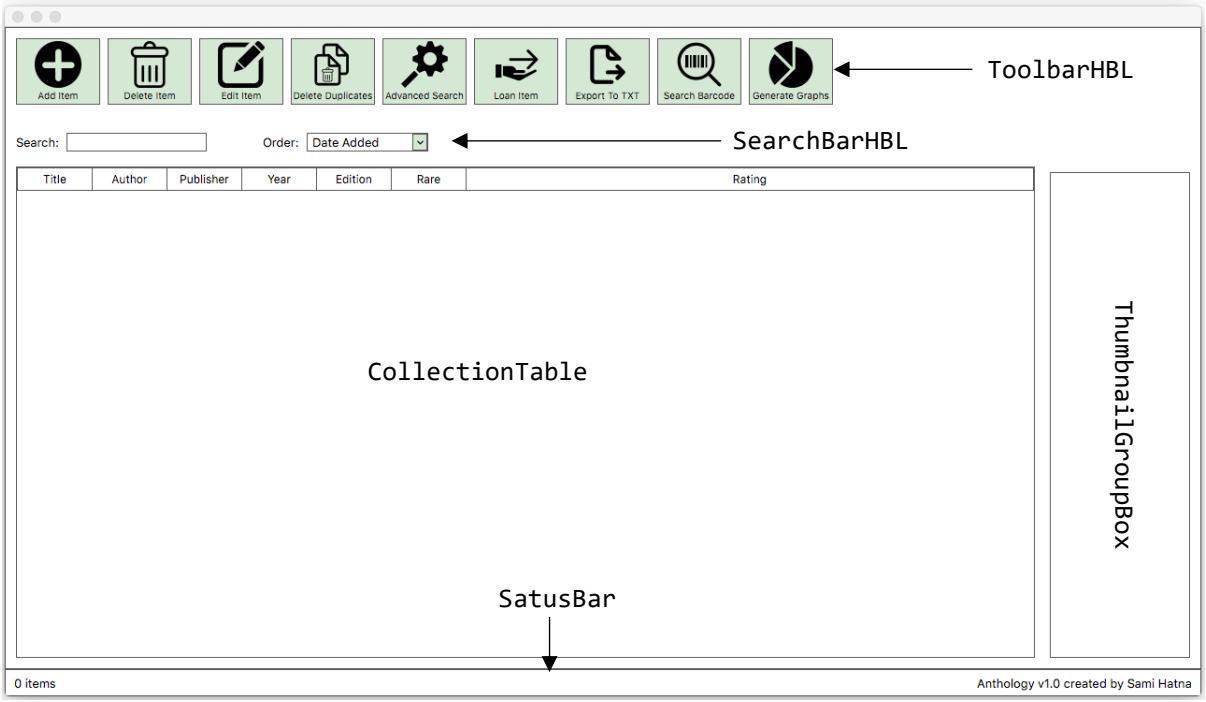
```
# Initialise window toolbar
def InitToolbar(self):
    # Toolbar buttons are displayed in a horizontal box layout
    self.ToolBarHBL = QHBoxLayout()
    self.OpenCollectionVBL1.addLayout(self.ToolBarHBL)

    # Each Toolbar button is an instance of the ToolBarBTN class
    # AddItemBTN is used for adding new items to the collection
    self.AddItemBTN = ToolBarBTN(
                          Text = "Add Item", IconPath = "Resources/AddItemIcon.png"
                                )
    self.ToolBarHBL.addWidget(self.AddItemBTN)

    # DeleteItemBTN deletes the selected item(s) from the collection
    self.DeleteItemBTN = ToolBarBTN(
                          Text = "Delete Item", IconPath = "Resources/DeleteIcon.png"
                                  )
    self.ToolBarHBL.addWidget(self.DeleteItemBTN)

    # EditItemBTN allows the user to edit the data stored about the selected item
    self.EditItemBTN = ToolBarBTN(
                          Text = "Edit Item", IconPath = "Resources/EditItemIcon.jpg"
                                 )
    self.ToolBarHBL.addWidget(self.EditItemBTN)

    # DeleteDuplicatesBTN deletes duplicate items from the collection
    self.DeleteDuplicatesBTN = ToolBarBTN(
                          Text = "Delete Duplicates",
                          IconPath = "Resources/DeleteDuplicatesIcon.png"
```

```
                                        )
        self.ToolBarHBL.addWidget(self.DeleteDuplicatesBTN)

        # AdvancedSearchBTN opens the advanced search window for constructing complex
          queries
        self.AdvancedSearchBTN = ToolBarBTN(
                                Text = "Advanced Search",
                                IconPath = "Resources/AdvancedSearchIcon.png"
                                           )
        self.ToolBarHBL.addWidget(self.AdvancedSearchBTN)

        # LoanItemBTN is for loaning out items
        self.LoanItemBTN = ToolBarBTN(
                                Text = "Loan Item", IconPath = "Resources/LoanItemIcon.png"
                                          )
        self.ToolBarHBL.addWidget(self.LoanItemBTN)

        # ExportTxtBTN exports the current collection as a TXT file
        self.ExportTxtBTN = ToolBarBTN(
                                Text = "Export To TXT",
                                IconPath = "Resources/ExportTxtIcon.png"
                                           )
        self.ToolBarHBL.addWidget(self.ExportTxtBTN)

        # BarcodeSearchBTN opens the barcode search window which allows users to add items
          to collections using their barcode
        self.BarcodeSearchBTN = ToolBarBTN(
                                Text = "Search Barcode",
                                IconPath = "Resources/BarcodeSearchIcon.png"
                                           )
        self.ToolBarHBL.addWidget(self.BarcodeSearchBTN)

        # GenerateGraphsBTN generates and displays graphs based on the user's collection
        self.GenerateGraphsBTN = ToolBarBTN(
                                Text = "Generate Graphs",
                                IconPath = "Resources/GenerateGraphsIcon.png"
                                           )
        self.ToolBarHBL.addWidget(self.GenerateGraphsBTN)

        self.ToolBarHBL.addStretch()
```

This is the output of `InitToolbar`:



At this point in development, the window looks like this:



The next step will be to program the function which will populate the table with items from the SQL table. Before I can do this however, I will have to insert some items into the table. For the time being, I will have to insert items from the MySQL command line as I haven't yet programmed the system for adding items through the app. For testing purposes, I will be using a book collection with the properties Title, Author, Publisher, Year and Edition.

```
INSERT INTO
   Table55 ( Title, Author, Publisher, Year, Edition, Rare, Rating )
VALUES
   (
      "The Ragged Trousered Philanthropists", "Robert Tressell",
      "Wordsworth", 1914, "Modern Greats", FALSE, 5
   )
```

	PK_Ta...	Title	Author	Publisher	Year	Edition	Rare	Rating	Thumbnail
▶	1	The Night Is Darkening Ro...	Emily Bronte	Penguin	1846	Penguin Classics	1	5	BLOB
	2	How Much Land Does a M...	Leo Tolstoy	Penguin	1886	Penguin Classics	0	5	BLOB
	3	Hamlet	William Shakesp...	Penguin	1603	Essential Shakes...	0	3	BLOB
	4	In Praise of Disobedience	Oscar Wilde	Verso	1891	Radical Thinkers	1	4	BLOB
	5	Capital	Karl Marx	Wordswo...	1867	Modern Greats	0	5	BLOB
	6	Notes on Nationalism	George Orwell	Penguin	1945	Penguin Modern	0	3	BLOB
	7	Paradise Lost	John Milton	Collins	1667	Collins Classics	1	3	BLOB
	8	The Ragged Trousered Ph...	Robert Tressell	Wordswo...	1914	Modern Greats	0	5	BLOB

I then created two objects called `RareWidget` and `RatingWidget`. `RareWidget` is the tag which is displayed in `CollectionTable` for any item which is marked as true for the Rare column. `RatingWidget` displays an image ranging from 1 to 5 stars based on the value of the Rating column

Code for `RareWidget`:

```
# RareWidget is a styled QLabel enclosed within a QWidget
# It is displayed in the table widget for any item that is tagged as rare
class RareWidget(QWidget):
    def __init__(self):
        super(RareWidget, self).__init__()
        RarityLBL = QLabel("Rare")
        RarityLBL.setStyleSheet("background: #7C5295; color: white; font-weight: bold;
```

```
        font-size: 12px; border-radius: 5px;")
        RarityLBL.setAlignment(Qt.AlignCenter)
        RarityLBL.setFixedSize(40, 20)
        self.setObjectName("BorderlessWidget")
        self.setStyleSheet("background-color: rgba(0,0,0,0%)")
        RarityLayout = QHBoxLayout(self)
        RarityLayout.addWidget(RarityLBL)
        RarityLayout.setAlignment(Qt.AlignVCenter)
        RarityLayout.setContentsMargins(0, 0, 0, 0)
```

The Resources folder in my programming directory contains five images of stars called “1 stars.png”, “2 stars.png”, “3 stars.png” etc. `RatingWidget` works by formatting the value stored in the Rating column into the file name and displaying the resulting image.

```
# RatingWidget displays an image of stars out of five based on the value stored in the
Rating column
class RatingWidget(QWidget):
    # Takes the argument Rating, this is just an integer out of five
    def __init__(self, Rating):
        super(RatingWidget, self).__init__()
        RatingLBL = QLabel()
        RatingPixmap = QPixmap()
        # Format value of Rating into file path to get correct picture
        RatingPixmap.load("Resources/{0} stars.png".format(Rating))
        RatingLBL.setPixmap(RatingPixmap.scaledToHeight(20))
        RatingLBL.setAlignment(Qt.AlignCenter)
        self.setObjectName("BorderlessWidget")
        self.setStyleSheet("background-color: rgba(0,0,0,0%)")
        RatingLayout = QHBoxLayout(self)
        RatingLayout.addWidget(RatingLBL)
        RatingLayout.setAlignment(Qt.AlignVCenter)
        RatingLayout.setContentsMargins(0, 0, 0, 0)
```

`PopulateTable()` is a crucial method as it will be called every time a change is made to the SQL table in order to refresh the PyQt table. `PopulateTable` begins by setting up the columns for `CollectionTable` and naming their headings. It then retrieves the value from the order combobox which tells the program what order the items should be displayed in. The program then retrieves the contents of the SQL table, using the `ORDER BY` clause to order the items as the user has specified. Once the results of the query have been fetched, it iterates through these results, storing each item of data in the appropriate cell of `CollectionTable`. For the Rare and Rating columns, the program will have to create instances of `RareWidget` and `RatingWidget` to store in the table cells. Populate table also uses a list called `LargestArray` for storing the width of each column based on the largest item in that column, so that no columns are too big or too small.

```
# This function populates the PyQt table widget with data from the SQL table
def PopulateTable(self):
    # Get columns from SQL table
    self.MyCursor.execute("SHOW COLUMNS FROM " + self.OpenCollectionTable)
    MyResult1 = self.MyCursor.fetchall()
    ColumnNames = []
    ColumnCounter = 0
    # LargestArray stores the size of each column
    # As the function progresses, LargestArray finds the size of the largest string
      for each column and appends it
    LargestArray = []

    # Iterate through the result of the SQL query and get amount of columns as well as
     column names
    for x in range(0, len(MyResult1) - 1):
        ColumnNames.append(MyResult1[x][0])
```

```
            ColumnCounter += 1
            LargestArray.append(40)
        # Set column count of table widget to column count of SQL table
        self.CollectionTable.setColumnCount(ColumnCounter)
        # Set the horizontal headings of the table widget to the headings of the SQL
          columns
        self.CollectionTable.setHorizontalHeaderLabels(ColumnNames)
        # Hide the first column as it stores the primary key for each item
        self.CollectionTable.setColumnHidden(0, True)

        # Clear table of rows before repopulating table
        self.CollectionTable.setRowCount(0)
        # Get number of entries in table
        self.MyCursor.execute("SELECT COUNT(*) FROM " + self.OpenCollectionTable)
        RowCount = self.MyCursor.fetchall()[0][0]
        # Set number of rows in table widget to number of entries in SQL table
        self.CollectionTable.setRowCount(RowCount)
        # Set label in status bar to display the number of items in the collection
        self.StatusLBL.setText(str(RowCount) + " items")
        # Select which order the items should be displayed in based on the value in
          OrderCB
        if self.OrderCB.currentText() == "Ascending":
            # Order items in ascending alphabetical order based on the first column
              (excluding the primary key)
            self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " ORDER BY
            " + str(MyResult1[1][0]) + " ASC")
        elif self.OrderCB.currentText() == "Descending":
            # Order items in descending order based on the first column
            self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " ORDER BY
            " + str(MyResult1[1][0]) + " DESC")
        else:
            # Order items in the order that they have been added to the collection
            self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable)

        # Retrieve results of SQL query
        MyResult2 = self.MyCursor.fetchall()
        # Iterate through each entry in the table
        for x in range(0, len(MyResult2)):
            # Store the primary key in the first (hidden) column of the table widget
            self.CollectionTable.setItem(x, 0, QTableWidgetItem(str(MyResult2[x][0])))
            # For loop here begins at 1 because we have already stored the primary key in
              a table cell
            # len(MyResult2[x]) - 3 excludes the Rare, Rating and Thumbnail columns as
              they are displayed in special widgets
            for y in range(1, len(MyResult2[x]) - 3):
                # Create a label and set label text to text value of current field the
                  loop is on
                Temp = QLabel(str(MyResult2[x][y]))
                # Add label to corresponding cell in the table widget
                self.CollectionTable.setCellWidget(x, y, Temp)
                # If the width of the label is greater than the width currently stored in
                  LargestArray, change the value at that index to the new largest width
                if int(Temp.width()) > LargestArray[y]:
                    LargestArray[y] = int(Temp.width())

            # If the item is tagged as rare...
            if MyResult2[x][len(MyResult2[x]) - 3] == 1:
                # Create an instance of RareWidget and add to table widget
                RarityCell = RareWidget()
                self.CollectionTable.setCellWidget(x, (len(MyResult2[x]) - 3), RarityCell)
            # Create an instance of RatingWidget, passing in the rating of the current
```

```
            item as a parameter
        RatingCell = RatingWidget(Rating = MyResult2[x][len(MyResult2[x]) - 2])
        self.CollectionTable.setCellWidget(x, (len(MyResult2[x]) - 2), RatingCell)

    # Set size of each column to the corresponding width in LargestArray
    for x in range(0, len(LargestArray)):
        self.CollectionTable.setColumnWidth(x, LargestArray[x] + 40)
```

Ascending or descending order of items is determined by alphabetically ordering the items in the first column of the SQL table

Table populated in ascending order:

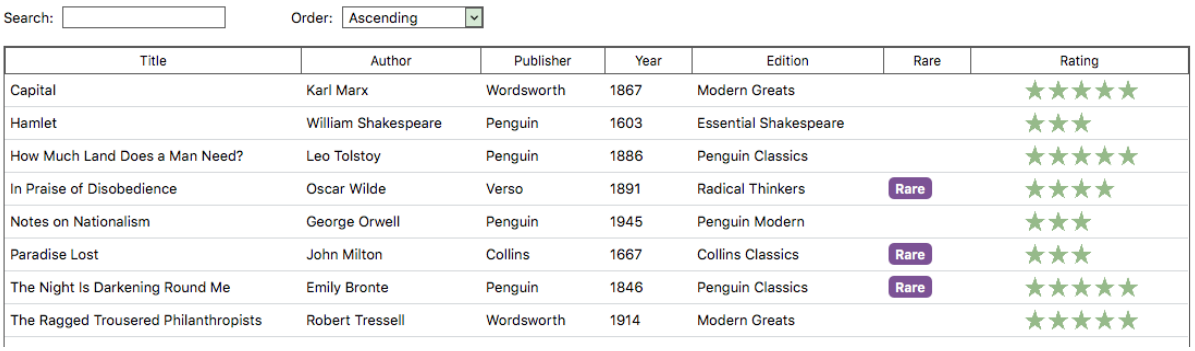


Table populated in descending order:

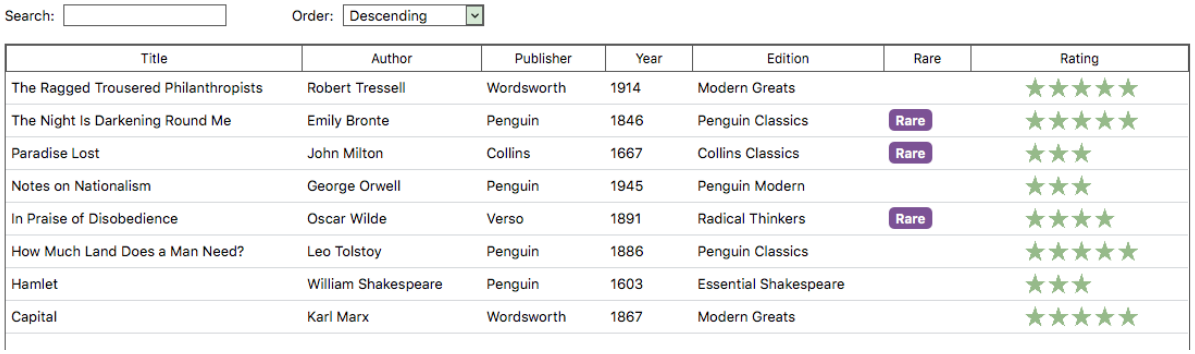
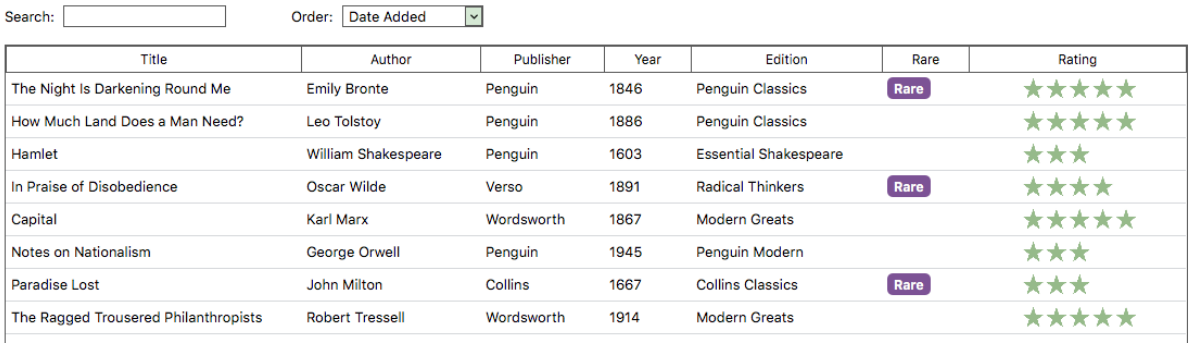


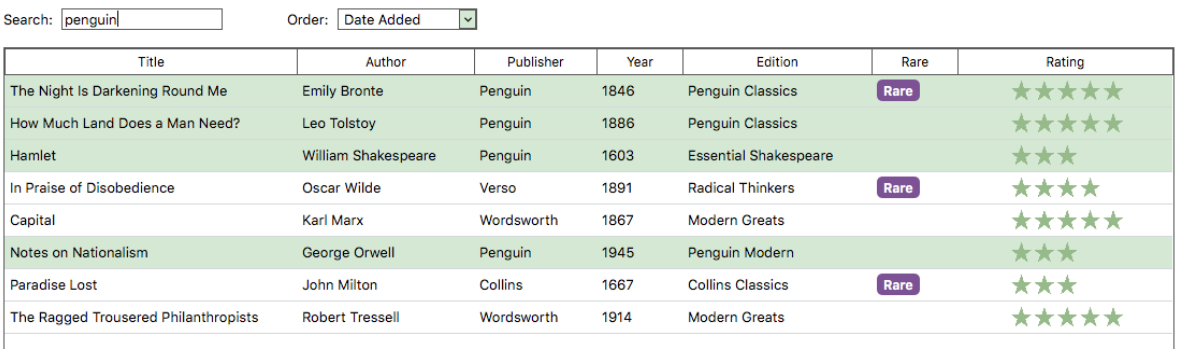
Table populated in chronological order:



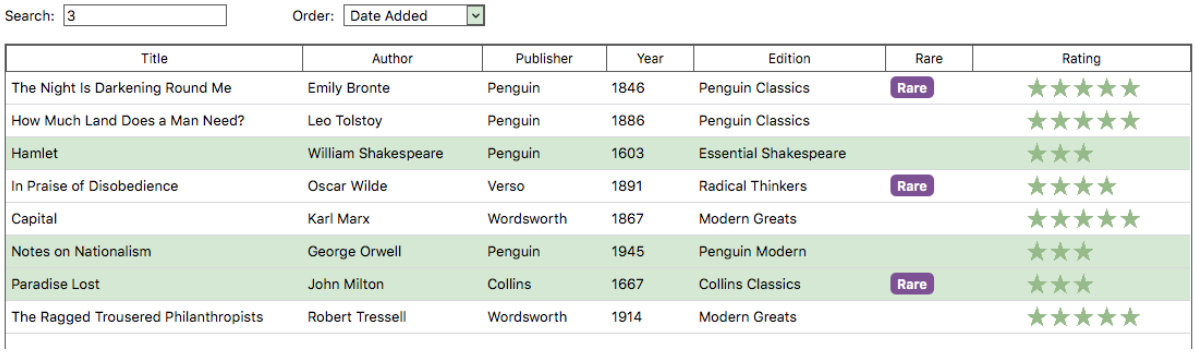
With the table now populated, I proceeded to program the simple search function. This function is executed when the text in the search bar text box changes. It takes the string in the search bar and queries the database, using the LIKE operator, in order to find any entries that match the search string. The program then highlights all the rows in the PyQt table which match with the results of the SQL query. The content of the search bar has to be validated using a presence check before the search is performed.

```
# This function is executed when the itemSelectionChanged signal is emitted
# It performs a simple search on all columns in the SQL table based on what the user
  has inputted in the search bar
def SimpleSearch(self):
    # Get search string from search bar
    SearchRequest = self.SearchBarTB.text()
    # SearchResult stores the rows of the table widget which contain the items that
      have fulfilled the terms of the search
    SearchResult = []
    # Presence check
    if SearchRequest == "":
        self.CollectionTable.clearSelection()
    else:
        # Clear all previous selections
        self.CollectionTable.clearSelection()
        self.CollectionTable.setSelectionMode(QAbstractItemView.MultiSelection)
        # Retrieve all column names from SQL table
        self.MyCursor.execute("SHOW COLUMNS FROM " + self.OpenCollectionTable)
        MyResult4 = self.MyCursor.fetchall()
        # Concatenate strings together to form a SQL query that will search the table
          for target items and return their primary key
        SQLStatement = 'SELECT PK_Table' + str(self.OpenCollectionID) + ' FROM ' +
        self.OpenCollectionTable + ' WHERE '
        # The function searches all columns in the table except the Thumbnail column
        for x in range(1, len(MyResult4) - 2):
            # Uses LIKE statements to find instances of the string inputted in the
              search bar in all fields of the database
            SQLStatement += MyResult4[x][0] + ' LIKE "%' + SearchRequest + '%" OR '
        SQLStatement += MyResult4[len(MyResult4) - 2][0] + ' LIKE "%' +
        SearchRequest + '%"'
        self.MyCursor.execute(SQLStatement)
        # Retrieve results of SQL query
        MyResult5 = list(set(self.MyCursor.fetchall()))
        # For each element in the result of the SQL query, find their corresponding
          row in the table widget by comparing their primary keys
        for z in range(0, self.CollectionTable.rowCount()):
            for x in range(0, len(MyResult5)):
                if self.CollectionTable.item(z, 0).text() == str(MyResult5[x][0]):
                    # Add each tableItem object to the list
                    SearchResult.append(self.CollectionTable.item(z, 0))

        # Highlight/Select each row stored in SearchResult to relay the results of the
          search to the user
        for z in range(0, len(SearchResult)):
                self.CollectionTable.selectRow(SearchResult[z].row())
        self.CollectionTable.setSelectionMode(QAbstractItemView.ExtendedSelection)
```

For example, if I were using the book collection (Table55) and I inputted the string “penguin” into the search bar, the function would create the SQL statement “`SELECT PK_Table FROM Table55 WHERE Title LIKE %’penguin’% OR Author LIKE %’penguin’% OR Publisher LIKE %’penguin’% OR Year LIKE %’penguin’% OR Edition LIKE %’penguin’% OR Rare LIKE %’penguin’% OR Rating LIKE %’penguin’%`”. This is what the search returns:



A search for the integer value 3 highlights all items with a rating of 3 stars



Title	Author	Publisher	Year	Edition	Rare	Rating
The Night Is Darkening Round Me	Emily Bronte	Penguin	1846	Penguin Classics	Rare	★★★★★
How Much Land Does a Man Need?	Leo Tolstoy	Penguin	1886	Penguin Classics		★★★★★
Hamlet	William Shakespeare	Penguin	1603	Essential Shakespeare		★★★
In Praise of Disobedience	Oscar Wilde	Verso	1891	Radical Thinkers	Rare	★★★★
Capital	Karl Marx	Wordsworth	1867	Modern Greats		★★★★★
Notes on Nationalism	George Orwell	Penguin	1945	Penguin Modern		★★★
Paradise Lost	John Milton	Collins	1667	Collins Classics	Rare	★★★
The Ragged Trousered Philanthropists	Robert Tressell	Wordsworth	1914	Modern Greats		★★★★★

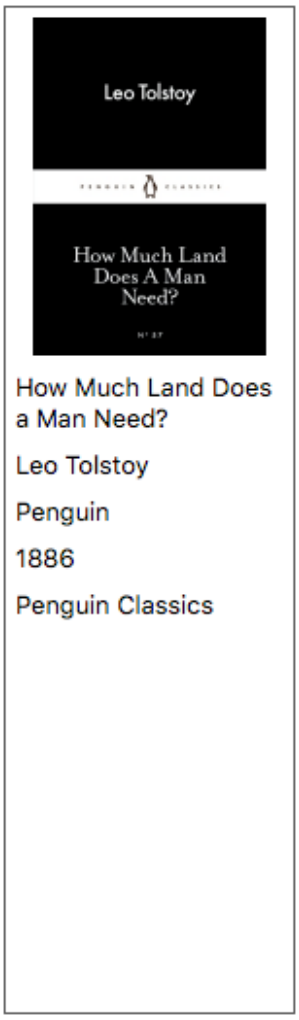
The simple search differs from the advanced search because it searches all columns for the search data whereas the advanced search should allow users to construct more precise queries.

The final method in this iteration that needed coding was the `DisplayImage` function which is executed when the user selects a row in the table. It displays the image and data of the selected item in the `ThumbnailGroupBox` on the left-hand side. Images are stored in the SQL table as Binary Large Objects, so the function has to convert the raw binary data into a `QImage` and then a `QPixmap`. I also decided, during development, that items marked as rare should have a small purple tag overlayed in the corner of their thumbnail. This feature would involve some image manipulation involving the `QPaintEvent`.

```
# This function is executed when the user selects an item in CollectionTable
# It displays the thumbnail for the selected item alongside the data stored about them
  in the SQL table
# The thumbnails of items which are marked as rare get a rare tag overlayed over their
  thumbnail
def DisplayImage(self):
    # Get currently selected row
    Row = self.CollectionTable.selectionModel().currentIndex()
    # Get the primary key of that row by reading the valus stored in the first
      (hidden) column
    PKValue = Row.sibling(Row.row(), 0).data()
    # Presence check
    if PKValue:
        # Retrieve the SQL entry which matches the currently selected row
        self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " WHERE
        PK_" + self.OpenCollectionTable + " = " + PKValue)
        MyResult3 = self.MyCursor.fetchall()[0]

        # If a binary large object has been stored in the Thumbnail column...
        if MyResult3[len(MyResult3) - 1]:
            # Load image as QImage from raw binary data
            ThumbnailImage = QImage.fromData(MyResult3[len(MyResult3) - 1])
            # Convert QImage into QPixmap
            ThumbnailPixmap = QPixmap.fromImage(ThumbnailImage)
            # Resize image
            ThumbnailPixmap = ThumbnailPixmap.scaledToWidth(120, Qt.SmoothTransform)
            # Check if item is tagged as Rare
            if MyResult3[len(MyResult3) - 3] == 1:
                # If image is rare, load overlay image
                OverlayPixmap = QPixmap("Resources/RareTag.png")
                # Initialise QPainter object
                Painter = QPainter(ThumbnailPixmap)
                # Draw rare tag over item thumbnail
                Painter.drawPixmap(0, 0, OverlayPixmap)
```

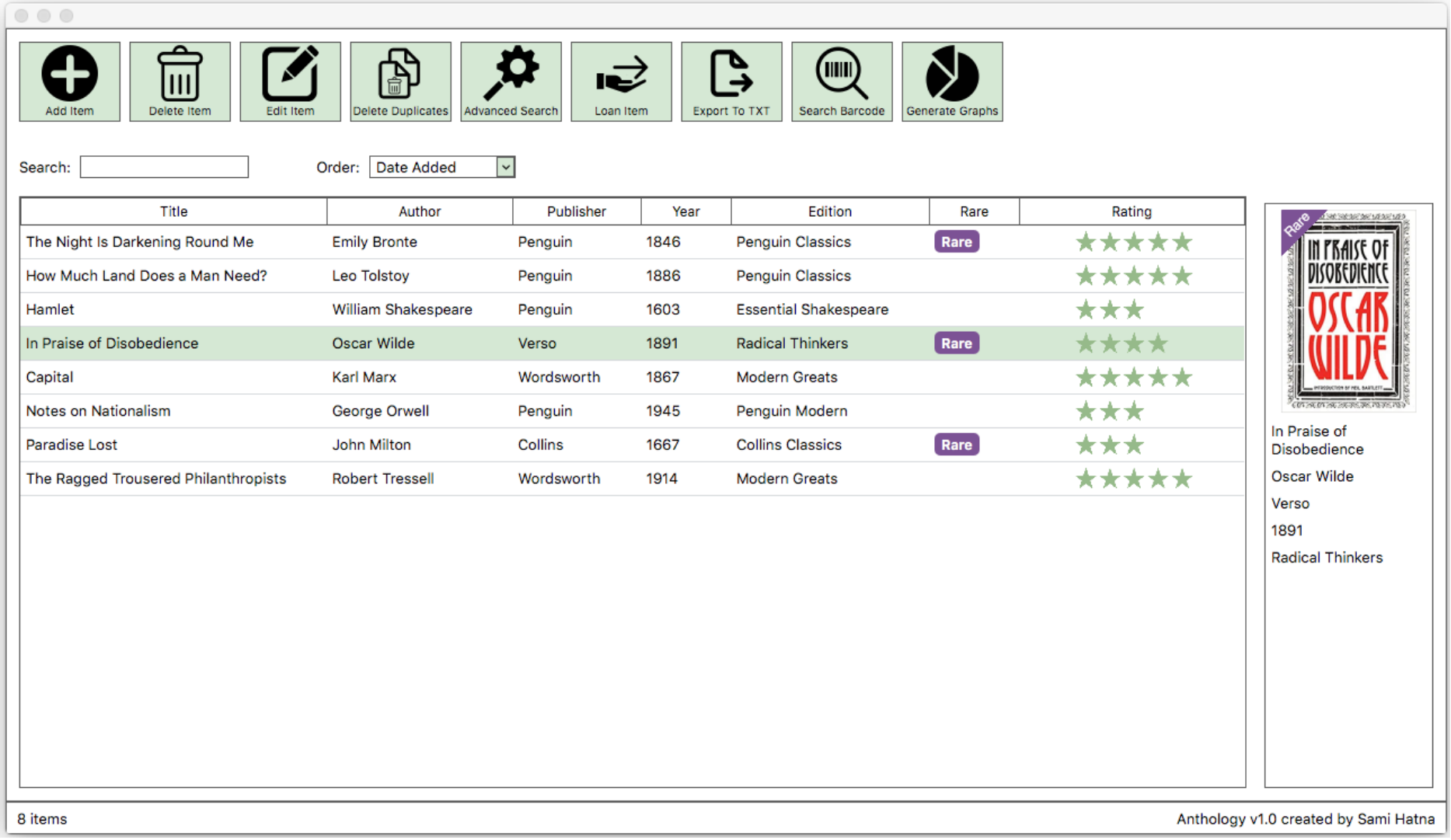
```
            # End paint event
            Painter.end()
        # If there is nothing in the Thumbnail column, display a generic 'No Image
          Available' image
        else:
            ThumbnailPixmap = QPixmap("Resources/NoImageAvailable.png")
            ThumbnailPixmap = ThumbnailPixmap.scaledToWidth(120, Qt.SmoothTransform)
        # Load pixmap into ThumbnailLabel
        self.ThumbnailImageLabel.setPixmap(ThumbnailPixmap)

        # Iterate through list of thumbnail labels and set each one's text to the
          corresponding data stored in the field for that item
        for x in range(0, len(self.ThumbnailLabelsArray)):
            self.ThumbnailLabelsArray[x].setText(str(MyResult3[x + 1]))
```

`ThumbnailGroupBox` now displays the following when an item is selected:

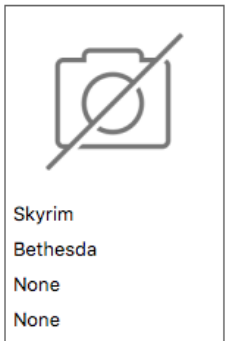


Rare items now have a special tag on their thumbnails:

I have inserted a screenshot of the full window on the next page to show where development is currently at.


Add Item
Delete Item
Edit Item
Delete Duplicates
Advanced Search
Loan Item
Export To TXT
Search Barcode
Generate Graphs
Search:
Order: Date Added
Title
Author
Publisher
Year
Edition
Rare
Rating
The Night Is Darkening Round Me
Emily Bronte
Penguin
1846
Penguin Classics
Rare
How Much Land Does a Man Need?
Leo Tolstoy
Penguin
1886
Penguin Classics
Hamlet
William Shakespeare
Penguin
1603
Essential Shakespeare
In Praise of Disobedience
Oscar Wilde
Verso
1891
Radical Thinkers
Rare
Capital
Karl Marx
Wordsworth
1867
Modern Greats
Notes on Nationalism
George Orwell
Penguin
1945
Penguin Modern
Paradise Lost
John Milton
Collins
1667
Collins Classics
Rare
The Ragged Trousered Philanthropists
Robert Tressell
Wordsworth
1914
Modern Greats
Rare
IN PRAISE OF DISOBEDIENCE
OSCAR WILDE
In Praise of Disobedience
Oscar Wilde
Verso
1891
Radical Thinkers
8 items
Anthology v1.0 created by Sami Hatna

Testing

Test	Expected Result	Actual Result
Select item without thumbnail	A generic placeholder image should be displayed in ThumbnailGroupBox	This test works as expected 
Null fields in certain columns	The function PopulateTable should just skip over these fields, leaving their cells blank	The initial loop I programmed for populating the table ran into logic errors when testing it on items with null fields. Rather than moving on, it would place the next field's contents in the empty cell, offsetting the position of all the data. I remedied this issue during development by reprogramming the loop to not ignore null fields
SQL injection attack on search bar text box	The text the user inputs into the search bar is directly used in a SQL query. This leaves the program vulnerable to injection attacks as hackers can input a SQL statement which will unknowingly run on the database	Initially, the program seemed to withstand the most common injection attacks. This is because I blocked any inputs containing suspicious strings such as "--". However, there were two forms of injection attack that I had not considered. The first is when hackers add the condition OR 1=1 to the end of SELECT WHERE statements. Because the condition 1=1 is always true, the query will return all the entries in a table, compromising the security of the database. The other injection attack I failed to account for was batched SQL statements. Batched statement attacks are when hackers add a semicolon on the end of a SQL statement to begin a new command. I remedied both of these issues by validating the user input to ensure it didn't contain any semicolons of equals signs.
PopulateTable() speed test	Populating the table with data from the database is one of the most intensive operations the program will have to perform as the program has to repopulate the whole table each time the function is executed, which involves lots of interfacing with the SQL database	I placed the code `Start = datetime.now()` at the start of the function and `End = datetime.now()` at the end of the function. Once the function had executed, I added the line `print(End - Start)` to get the time the function took to execute. This outputted **0.049768 seconds** on a collection consisting of 20 items. At this point in development, this seems to be a reasonable amount of time for the function to take. However, this time will grow as the size of the dataset grows so I will revisit this test when I have got some end-user feedback from the beta testing phase.

Integration testing involved testing opening the collection by clicking on a collection button and closing the current open collection. Each instance of `CollectionButton` in `CollectionWindow` has a method called `OpenCollection`. This method emits a signal which passes the primary key of the collection which is being opened to the `CollectionWindow` object. `CollectionWindow` then passes this primary key as an argument in the `OpenCollectionWindow` class, thus informing the object as to which collection it is actually opening. Closing a collection simply closes the current instance of `OpenCollectionWindow` and creates a new instance of `CollectionWindow`. Both functions work well, proving that the modules coded in iterations 4/5 and iteration 6 work in conjunction with each other.

Review

We are now at a point in development where users are able to open collections. This window is headed by a toolbar, a search bar and a combo box for selecting the order collections should be presented in. Items in the collection are presented in a table, with the selected item's thumbnail and data being displayed in a panel on the left hand side of the window. The actual implementation of this window is, in appearance, very similar to the prototype from the design stage. However, programmatically, the implementation of the window has been changed quite a bit from the pseudocode of the design stage. The primary change was the introduction of more comprehensive validation of user inputs. I also didn't account for the use of signals and slots in my pseudocode. Going forward, the next step in development is coding the core functions of adding, deleting and editing items.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- A table presenting items in a user's collection along with corresponding data (3)
- The selected item's thumbnail and information should be displayed in the left sidebar (4)
- A simple search function for quickly searching for an item (10)
- The ability to order collections in different ways (e.g., alphabetical, chronological, etc.) (13)
- The option to mark rare or niche items with a special tag (15)

Iteration 7; Adding Items

This iteration involves coding the window through which users will add items to collections. It is split into two phases: coding the front-end window and coding the back-end SQL operations. I began by creating an object inheriting from `QWidget` called `AddItemWindow`. `AddItemWindow` takes multiple arguments:

- The database cursor and connection objects
- The primary key of the currently logged in user
- The primary key in the Collections table of the currently open collection
- The table name of the currently open collection
- The button widget that was pressed to create this instance of `AddItemWindow` (`AddItemBTN`)

`AddItemBTN` is passed into the class as an argument because it is used to calculate the position the add item window is displayed at. It should display below the button that was pressed to initialize it.

This is the function which is executed when `AddItemBTN` is pressed (`ItemAddedSignal` will be discussed in greater detail further on):

```
def AddItemMethod(self):
    self.AddItemWindowInstance = AddItemWindow.AddItemWindow(
                                    MyCursor=self.MyCursor,
                                    MyDB=self.MyDB,
                                    ActiveUserID=self.ActiveUserID,
                                    OpenCollectionID=self.OpenCollectionID,
                                    OpenCollectionTable=self.OpenCollectionTable,
                                    widget = self.AddItemBTN
                                    )
    self.AddItemWindowInstance.ItemAddedSignal.connect(self.PopulateTable)
```

I wanted the window to have the following outline, with the arrow pointing to `AddItemBTN`:

To achieve this, I made the window inherit from `QWidget` rather than `QMainWindow`. This allowed me to access the paint event of the widget and set the `QPainter` to paint the window as this image rather than a generic rectangle. This is the declaration and init method of the window (note that the windows position is calculated using the global position of the argument widget):

```
# This class is the window which users add items to the collection from
# It inherits from QWidget rather than QMainWindow because I need to access its
  painEvent
class AddItemWindow(QWidget):
    # This signal is emitted when the user has successfully added an item to the
      collection
    # When the OpenCollection object receives this signal, it will execute the
      function PopulateTable()
    ItemAddedSignal = pyqtSignal(bool)
    # Takes the same arguments as OpenCollectionWindow except for widget which is the
      toolbar button which is pressed to open the window
    # Widget is an argument because it is used to determine the position of this new
```

```
      window on the screen
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID,
                 OpenCollectionTable, widget = None):
        super(AddItemWindow, self).__init__()

        # Reassign arguments as attributes
        self.ActiveUserID = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.OpenCollectionID = OpenCollectionID
        self.OpenCollectionTable = OpenCollectionTable
        # Boolean for keeping track of if the user has uploaded a thumbnail or not
        self.ContainsImage = False

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Initialise main window layout
        self.AddItemVBL1 = QVBoxLayout()
        self.setLayout(self.AddItemVBL1)

        self.InitUI()

        # Set this window so that it closes when its parent window is closed
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setAttribute(Qt.WA_TranslucentBackground)
        # Modal windows force the user to interact with it before they can go back to
          using the parent application
        self.setWindowModality(Qt.ApplicationModal)
        # Get rid of window frame and ensure this window always stays on top of other
          applicaton windows
        self.setWindowFlags(Qt.WindowFlags(
                                            Qt.FramelessWindowHint
                                            |Qt.WindowStaysOnTopHint)
                                            )

        # Determine where window should be positioned based on the widget which is
          passed in as an argument
        x = widget.mapToGlobal(QPoint(0, 0)).x()
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        # The window should appear on the screen just below the AddItemBTN which is
          clicked to create it
        self.move(x, y)

        # Show window
        self.show()
```

To create irregularly shaped windows with outlines like the image above, we have to repurpose the `QWidget`'s paint event. To do this, we redefine the function, create a `QPainter` instance and pass a `QPixmap` with the outline image loaded onto it into the `QPainter`. We then start the painter and it draws the window as the image.

```
# This event runs when the window is first initialized
# It draws the outline/background of the window as the image "AddItemOutline.png"
# This allows me to create irregularly shaped windows
def paintEvent(self, event):
    # Create painter
    Painter = QPainter()
    # Start painter, set painter to perform on AddItemWindow (self)
    Painter.begin(self)
```

```
    # Create pixmap to act as guide for painter
    Outline = QPixmap()
    Outline.load('Resources/AddItemOutline.png')
    # Paint window as Outline pixmap
    Painter.drawPixmap(QPoint(0, 0), Outline)
    # End paint event
    Painter.end()
```

Initially, the new paint event worked, but it was not drawing the image with the correct aspect ratio. I remedied this issue by resetting the window's `sizeHint`:

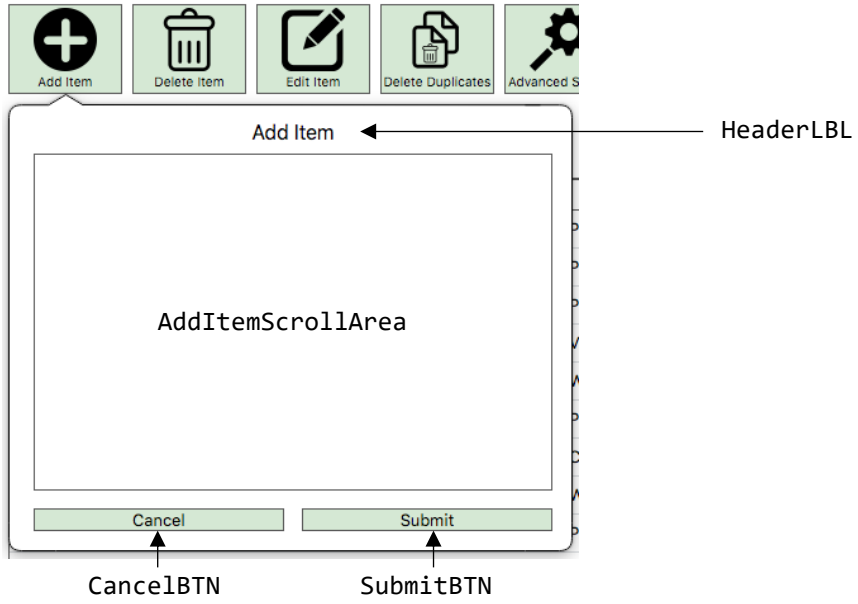
```
# Inbuilt PyQt function
# sizeHint is the preferred size of the widget
# It has to be set so that the painter event can paint the outline image according to
these dimensions
def sizeHint(self):
    return QSize(450, 365)
```

This is the current output when `AddItemBTN` is clicked. As you can see, the window has the same shape as the image above and the top of the window is positioned below the button which is clicked to create it.

The next method which needs coding populates the window with a header label, a scroll area and two action buttons for submitting the new item and cancelling the operation.

```
# Initialise and display window widgets
def InitUI(self):
    # Header label informs users exactly what the window's purpose is
    HeaderLBL = QLabel("Add Item")
    HeaderLBL.setStyleSheet("font-size: 16px;")
    HeaderLBL.setAlignment(Qt.AlignCenter)
    self.AddItemVBL1.addWidget(HeaderLBL)

    # The widgets which users use to input data about the item are presented in a
      scroll area
    self.AddItemScrollArea = QScrollArea()
    self.AddItemVBL1.addWidget(self.AddItemScrollArea)
    self.AddItemScrollArea.setWidgetResizable(True)
    # This is the widget the scroll area displays
    self.ScrollAreaWidget = QWidget()
```

```
    self.ScrollAreaWidget.setObjectName("BorderlessWidget")
    # This is the grid layout of the widget in the scroll area
    self.ScrollAreaLayout = QGridLayout()
    self.ScrollAreaLayout.setAlignment(Qt.AlignTop)
    self.ScrollAreaWidget.setLayout(self.ScrollAreaLayout)
    self.AddItemScrollArea.setWidget(self.ScrollAreaWidget)

    # Two buttons in this Horizontal layout, one for adding the new item to the
      collection, and one for cancelling the operation
    ActionBTNsHBL = QHBoxLayout()
    self.CancelBTN = QPushButton("Cancel")
    self.CancelBTN.clicked.connect(self.close)
    self.SubmitBTN = QPushButton("Submit")
    self.SubmitBTN.clicked.connect(self.AddItem)
    ActionBTNsHBL.addWidget(self.CancelBTN)
    ActionBTNsHBL.addWidget(self.SubmitBTN)
    self.AddItemVBL1.addLayout(ActionBTNsHBL)

    self.PopulateScrollArea()
```

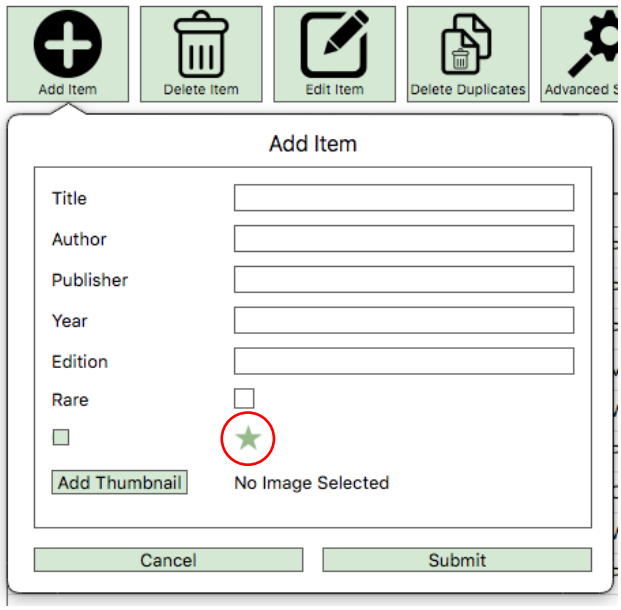


Next, I coded the function `PopulateScrollArea`. This function fills `AddItemScrollArea` with input fields which allow the user to enter the data for the new item. For each column in the collection's table, a text input box is created and added to the scroll area. Because there is a varying number of text input boxes depending on how many columns are in the collection, the text box objects will be stored in a list called `FieldTextBoxes`. They can then be accessed by calling their corresponding index in that list. The user will mark an item as rare using a checkbox. They will input the rating of the item using a slider. The user can upload a thumbnail by clicking a button which will open the system's file explorer and allow them to select their image of choice.

```
# This function populates the scroll area with input fields for entering the data
  about the new item
def PopulateScrollArea(self):
    # Get all column names and their datatypes from the current collectin's table
    self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
    MyResult1 = self.MyCursor.fetchall()
    # Because there will be a variable number of input boxes, they have to be stored
      in a list for future access
    self.FieldTextBoxes = []
```

```
        # Iterate through over each column, creating a new input box for each one
          (excluding primary key, rare, rating and thumbnail columns)
        for x in range(1, len(MyResult1) - 3):
            # Add a label to the grid layout displaying the column name
            self.ScrollAreaLayout.addWidget(QLabel(MyResult1[x][0]), x, 0)
            # Create a text input box for the current column
            Temp = QLineEdit()
            Temp.setFixedWidth(250)

            # If the datatype of the current column is integer, apply a QIntValidator to
              that text box
            if MyResult1[x][1] == b'int':
                # QValidators limit the input into a text box, in this case it limits the
                  input to only accepts integers
                Temp.setValidator(QIntValidator())

            # Add text box to grid layout
            self.ScrollAreaLayout.addWidget(Temp, x, 1)
            # Add text box to list
            self.FieldTextBoxes.append(Temp)

        # Add Rare label
        self.ScrollAreaLayout.addWidget(QLabel("Rare"), len(MyResult1) - 2, 0)
        # The user marks an item as rare by checking a check box
        self.RareCheckBox = QCheckBox()
        self.RareCheckBox.setFixedWidth(15)
        # Add check box to grid layout
        self.ScrollAreaLayout.addWidget(self.RareCheckBox, len(MyResult1) - 2, 1)

        # The user uses a slider to select a value from one to five for the item's rating
        self.RatingSlider = QSlider()
        self.RatingSlider.setOrientation(Qt.Horizontal)
        self.RatingSlider.setTickInterval(1)
        self.RatingSlider.setMinimum(1)
        self.RatingSlider.setMaximum(5)
        # When the value of the slider is changed, the signal valueChanged is emitted
        # This signal is triggers the function ChangeRatingImage, which changes the pixmap
          in RatingImageLBL to display the amount of star corresponding with the value of
          the slider
        self.RatingSlider.valueChanged.connect(self.ChangeRatingImage)
        self.ScrollAreaLayout.addWidget(self.RatingSlider, len(MyResult1) - 1, 0)

        # This is the label which displays the image of the rating the user has selected
        self.RatingImageLBL = QLabel()
        self.ChangeRatingImage()
        self.ScrollAreaLayout.addWidget(self.RatingImageLBL, len(MyResult1) - 1, 1)

        # The user presses this button to open their system's file explorer
        self.UploadImageBTN = QPushButton("Add Thumbnail")
        self.UploadImageBTN.clicked.connect(self.UploadImage)
        self.UploadImageBTN.setFixedWidth(100)
        self.ScrollAreaLayout.addWidget(self.UploadImageBTN, len(MyResult1), 0)

        # If the user chooses a thumbnail image, this label will be changed to show the
          name of the currently selected image
        self.UploadedImageLBL = QLabel("No Image Selected")
        self.UploadedImageLBL.setFixedWidth(250)
        self.ScrollAreaLayout.addWidget(self.UploadedImageLBL, len(MyResult1), 1)
```

This is what the populated window looks like:



In the above code, the `valueChanged` signal of `RatingSlider` is connected to the function `ChageRatingImage`. The image of a star circled in red is the widget `RatingImageLBL`. When the value of the slider changes, the function should change the image displayed so that the number of stars corresponds with the value of the slider.

```
# This function is executed when the valueChange signal is emitted
# It displays an image of 1 to 5 stars, dependent on the value of RatingSlider
def ChangeRatingImage(self):
    # Get value of slider
    SliderValue = self.RatingSlider.value()
    # Set label pixmap to corresponding image
    self.RatingImageLBL.setPixmap(QPixmap("Resources/" + str(SliderValue) +
    "stars.png").scaledToHeight(20,Qt.SmoothTransform))
```

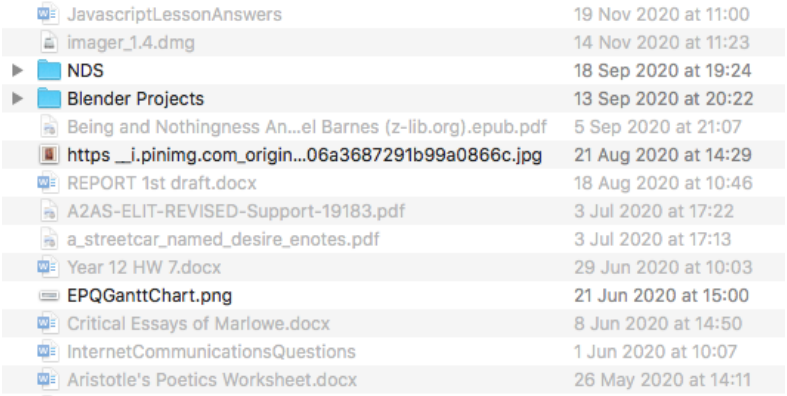
As the images below demonstrate, the picture displayed changes with the value of the slider

Another method that needed programming was the one executed when the user presses `UploadImageBTN`. This function opens a file dialog and allows the user to select a thumbnail image. When the user selects an image, the function should return its file path. The program than uses Python's in-built file handling capabilities to convert the image file into raw binary data, ready to be inserted into the Thumbnail column. The input file needs to be validated to ensure that it is an image file. The workaround I devised was to limit the user's choice of files in the file dialog to only jpg and png files, by passing the argument "Image files (*.jpg *.png)" into the constructer of the `QDialog`.

```
# This function is executed when the user presses UploadImageBTN
def UploadImage(self):
    # "Image files (*.jpg *.png) limits user selection to only Jpeg or png files
    # This function opens a file dialog and returns the path of the selected image
    self.FileName, _ = QFileDialog.getOpenFileName(self, "Open Image File",
```

```
                    os.path.abspath(os.sep), "Image files (*.jpg *.png)")
    # If the user selects a file...
    if self.FileName:
        # Change the label text to the name of the selected file
        self.UploadedImageLBL.setText("selected " + os.path.basename(self.FileName))
        # Set boolean value created in __init__ method to true
        self.ContainsImage = True
        # Convert selected image into raw binary data and store under variable
          self.BinaryData
        with open(self.FileName, "rb") as file:
            self.BinaryData = file.read()
```

As you can see below, the file dialog only allows users to open folders or select files with the extension .jpg or .png



When the user selects an image and it has been successfully converted into binary, the label `UploadedImageLBL` is changed to display the file name of the selected image.

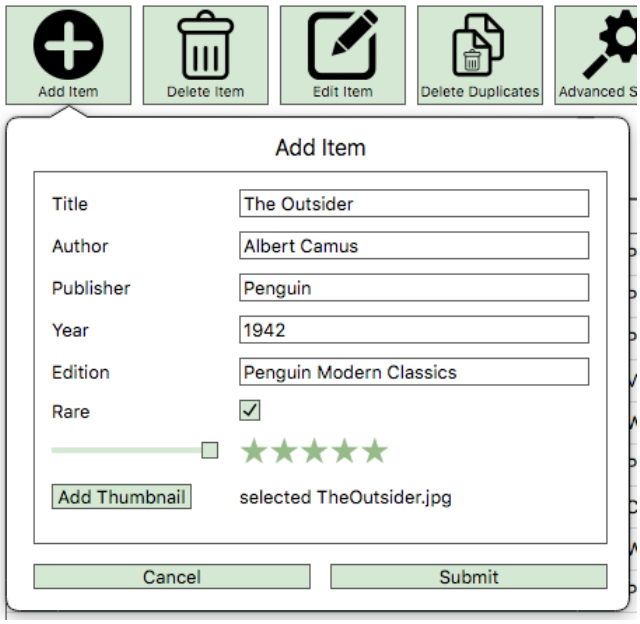
The crucial function of this iteration is the method which collects the user inputs and inserts them as a new entry into the database. The function begins by performing a presence check to ensure that all the input boxes have been filled in. If the presence check is failed, the empty text boxes will be passed into a new function which will handle the error. If the presence check is passed, the program proceeds to collect all the user inputs from the text boxes, the rare checkbox, the rating slider and the uploaded thumbnail (stored in `self.BinaryData`) and format them as a SQL statement. One way of formatting values into a SQL command is using the substitution character `%s` and then passing a tuple alongside the statement into the `cursor.execute` function. The program will then format the contents of the tuple into the command. For example, on the books collection used in previous examples, the SQL command would be "`INSERT INTO Table55 (Title, Author, Publisher, Year, Edition, Rare, Rating) VALUES (%s,%s,%s,%s,%s,%s,%s)`". Each `%s` would then have the user inputted value substituted in when the line `MyCursor.execute(SQLStatement,InsertionData)` is executed.

```
# This function performs a presence check on the input fields before the item is added
to the collection
def PresenceCheck(self):
    # First perform presence check to ensure that no input fields have been left empty
    EmptyTextBoxes = []
    FilledTextBoxes = []
    Error = False
    # Iterate over each text box and check if they are empty
    for x in range(0, len(self.FieldTextBoxes)):
```

```
        if self.FieldTextBoxes[x].text() == "":
            # Append all empty text boxes to this list
            EmptyTextBoxes.append(self.FieldTextBoxes[x])
            # Boolean indicates whether the program has encountered an empty text box
              yet
            Error = True
        else:
            # Append all filled text boxes to this list
            FilledTextBoxes.append(self.FieldTextBoxes[x])
    # If there are any empty text boxes, run this error handling function
    if Error:
        self.AddItemError(EmptyTextBoxes, FilledTextBoxes)
    else:
        return True

# This function is executed when the user presses SubmitBTN
# It collects all the user inputs and formats them into a SQL statement which it then
  executes
def AddItem(self):
    # If the presence check returns TRUE, add the item
    if self.PresenceCheck():
        # SQLStatement is a dynamically generated string which stores the SQL command
        SQLStatement = "INSERT INTO " + self.OpenCollectionTable + " ("
        # Get column names from database
        self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
        MyResult2 = self.MyCursor.fetchall()
        # Add each column name to the SQL statement
        # The loop starts from 1 because the primary key column should automatically
          increment itself
        for x in range(1, len(MyResult2) - 3):
            SQLStatement += "`" + MyResult2[x][0] + "`, "
        # If the user has uploaded a thumbnail, add the Thumbnail column to the list
          of columns in which we are inserting data
        if self.ContainsImage:
            SQLStatement += "Rare, Rating, Thumbnail) VALUES ("
            Count = len(MyResult2) - 1
        # If the user hasn't uploaded a thumbnail, there is no data to insert into
          Thumbnail column, so its is left out of the command
        else:
            SQLStatement += "Rare, Rating) VALUES ("
            Count = len(MyResult2) - 2
        # %s operator is used to substitute values from DataForInsertion tuple into
          statement when it is executed
        for x in range(1, Count):
            SQLStatement += "%s, "
        SQLStatement += "%s)"

        # DataForInsertion is the tuple which stores all the data we have retrieved
          from the user input fields
        DataForInsertion = ()
        # Get the contents of each text box in the window and add it to the tuple
        for x in range(0, len(self.FieldTextBoxes)):
            DataForInsertion += (self.FieldTextBoxes[x].text(), )
        # If user has checked the rare check box, set boolean value of Rare column to
          TRUE
        if self.RareCheckBox.isChecked():
            DataForInsertion += (True, )
        # Otherwise, the boolean value will be FALSE
        else:
            DataForInsertion += (False, )
        # Add the value of the slider for the Rating column
        DataForInsertion += (self.RatingSlider.value(), )
```

```
        # If the user has uploaded an image, add the binary data to the tuple
        if self.ContainsImage:
            DataForInsertion += (self.BinaryData, )
        # Execute the SQL command, substituting the contents of the tuple in for each
          of the %s operators
        self.MyCursor.execute(SQLStatement, DataForInsertion)
        # Save changes to database
        self.MyDB.commit()
        # Emit signal telling root window to repopulate CollectionTable
        self.ItemAddedSignal.emit(True)
        # Close AddItemWindow
        self.close()
```

Now, if I input the details for a book, it will be stored as a new item in the database.



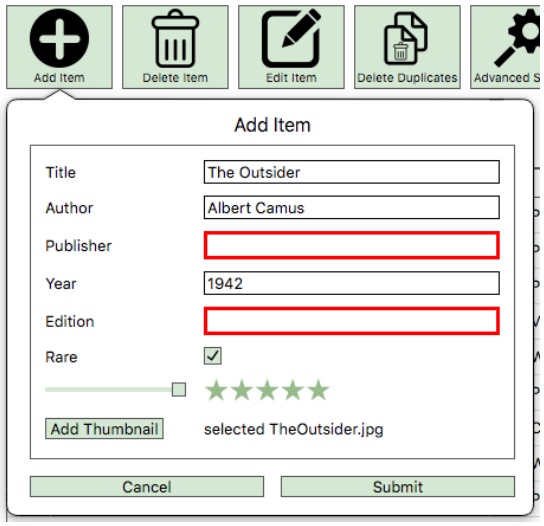
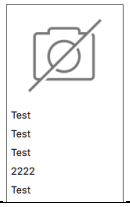
PK_Table55	Title	Author	Publis...	Year	Edition	Rare	Rating	Thumbnail
10	The Outsider	Albert Ca...	Penguin	1942	Penguin Modern Classics	1	5	BLOB

When the user presses submit, the signal `ItemAddedSignal` is emitted. This signal is bound to the function PopulateTable in the OpenCollection class, so when this signal is received, the table in the main window should be refreshed in order to show the new item.

The final method of this iteration is the function which is called if the user leaves an input field empty when they press the submit button. This function takes the lists `OffendingWidgets` and `NonOffendingWidgets` as parameters. The method is called in the `AddItem` method when the presence check is failed. It changes the border colour of every empty text box to red.

```
# This function is executed when the user submits an item but has left one of the
  input fields blank
# It highlights each empty text box with a red border
# It takes two parameters: a list of empty text boxes and a list of filled text boxes
def AddItemError(self, OffendingWidgets, NonOffendingWidgets):
    # Iterate through list of empty text boxes and set their border to red
    for TextBox in OffendingWidgets:
        TextBox.setStyleSheet("border: 1px solid #FF0000")
    # Iterate through list of filled text boxes and set their border to black
    for TextBox in NonOffendingWidgets:
        TextBox.setStyleSheet("border: 1px solid black")
```

Testing

Test	Expected Result	Actual Result
Leave input field blank when adding new item	The item shouldn't be added to the database and each empty textbox should be highlighted with a red border	This is what the window looks like when the user has left an input box empty: 
Add an item without uploading a thumbnail	The thumbnail column for this entry should be empty. When the item is selected in `CollectionTable`, a generic placeholder image should be displayed instead	The image below shows an item without a thumbnail, as you can see the thumbnail field is null  This is the image displayed when this item is selected 
SQL injection attack on input text boxes	The program should reject any suspicious inputs	I reused the code from the previous iteration for validating user inputs to prevent against SQL injections, so this time the program withstood attempted attacks
Upload corrupted image file as thumbnail	The program should display the same placeholder image it uses when an item doesn't have a thumbnail	Initially, this test crashed the program as it encountered a runtime error whilst trying to convert the raw binary data into a QPixmap. I remedied this issue by using a try/except clause to implement a failsafe procedure (backtracking)

Review

We now have a fully functioning system for adding items to a collection. The design for this window is the same as the prototype in the design stage, although I have changed the outline of the window by reimplementing the object's `PaintEvent`. The next iteration will involve writing the code that allows users to delete selected items.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- The option to add and delete items from collections (7)

Iteration 8; Deleting Items

This iteration focusses on 2 key components of the application: deleting selected items and deleting duplicate items. Both of these functions will be methods of the `OpenCollectionWindow` object.

The first method – `DeleteItem` – is executed when the toolbar button `DeleteItemBTN` is pressed. It works by gathering all the rows of the frontend table that the user has selected. The table's first column is hidden from the user because it contains the primary key of each item, so the program uses this first column to retrieve the primary key of each selected item. These primary keys are stored in a list, and then used to construct a SQL command which deletes those items from the database.

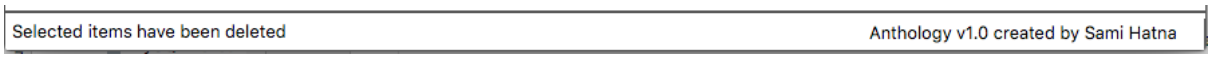
```
# This function is called when DeleteItemBTN is pressed
# It deletes the selected items in CollectionTable from the database
def DeleteItemMethod(self):
    # List stores all the rows in CollectionTable the user has selected
    SelectedRowsArray = []
    # List will store the primary keys of the items the user has selected
    SelectedRowsIDArray = []

    # Iterate through all the selected rows and add them to the list
    for x in self.CollectionTable.selectionModel().selectedRows():
        SelectedRowsArray.append(x.row())

    # Iterate through the array of selected rows and add their primary key to the
      second list
    for y in range(0, len(SelectedRowsArray)):
        # The primary key of each item is stored in the first column, which is hidden
        SelectedRowsIDArray.append(
                            self.CollectionTable.item(SelectedRowsArray[y], 0).text()
                                   )
    # If the user hasn't selected any items before they pressed the button, there is
      no point in proceeding
    if len(SelectedRowsIDArray) == 0:
        pass
    else:
        # Data has to be passed into a SQL command as a tuple
        SelectedRowsIDTuple = tuple(SelectedRowsIDArray)
        # Delete each item in the tuple from the database
        for z in range(0, len(SelectedRowsIDArray)):
            self.MyCursor.execute("DELETE FROM " + self.OpenCollectionTable + " WHERE
            PK_" + self.OpenCollectionTable + " = %s" % (SelectedRowsIDTuple[z]))
        # Save changes to database
        self.MyDB.commit()
        # Refresh table to get rid of deleted items
        self.PopulateTable()
        # Display temporary message to user informing them that operation was
          successful
        self.StatusBar.showMessage("Selected items have been deleted", 3000)
```

The method `DeleteDuplicates` uses the condition `COUNT(*) > 1` to select every item in the SQL table where there is more than one entry with the exact same data stored in the specified columns. It then deletes every duplicate item, leaving one of each set of duplicates in the table (e.g., if you had three duplicate items, two would be deleted and one would be left over)

```
# This function is executed when DeleteDuplicatesBTN is pressed
# It deletes all duplicate items from the database
def DeleteDuplicatesMethod(self):
    # Get columns from open collection table
    self.MyCursor.execute("SHOW COLUMNS FROM " + self.OpenCollectionTable)
```

```
    MyResult6 = self.MyCursor.fetchall()

    # Construct a SQL command which will select all the duplicate entries in the table
      using the COUNT(*) command
    SQLStatement = "SELECT "
    for x in range(1, len(MyResult6) - 4):
        SQLStatement += MyResult6[x][0] + ", "
    SQLStatement +=MyResult6[len(MyResult6) - 4][0]+" FROM " +self.OpenCollectionTable
    + " GROUP BY "
    for x in range(1, len(MyResult6) - 4):
        SQLStatement += MyResult6[x][0] + ", "
    SQLStatement += MyResult6[len(MyResult6) - 4][0] + " HAVING COUNT(*) > 1"
    self.MyCursor.execute(SQLStatement)
    MyResult7 = self.MyCursor.fetchall()

    # Iterate through duplicate entries
    for x in range(0, len(MyResult7)):
        # Construct a SQL command which returns the primary key of each duplicate
          record
        SQLStatement2 = "SELECT PK_" + self.OpenCollectionTable + " FROM " +
        self.OpenCollectionTable + " WHERE "
        # For loop starts at 1 because the first instance of the duplicate entry isn't
          deleted
        for y in range(1, len(MyResult6) - 4):
            SQLStatement2 += str(MyResult6[y][0]) + " = '" + str(MyResult7[x][y - 1])
            + "' AND "
        SQLStatement2 += str(MyResult6[len(MyResult6) - 4][0]) + " = '" +
        str(MyResult7[x][len(MyResult7[x]) - 1]) + "' ORDER BY PK_" +
        self.OpenCollectionTable + " DESC"
        self.MyCursor.execute(SQLStatement2)
        # MyResult8 stores all the duplicate entry primary keys
        MyResult8 = self.MyCursor.fetchall()

        # For each duplicate record (there may be more than one duplicate) delete it
          from the SQL table
        for z in range(0, len(MyResult8) - 1):
            self.MyCursor.execute("DELETE FROM " + self.OpenCollectionTable + " WHERE
            PK_" + self.OpenCollectionTable + " = " + str(MyResult8[z][0]))
            # Save changes to database
            self.MyDB.commit()

    # Refresh table to get rid of deleted items
    self.PopulateTable()
    # Display temporary message to user informing them that operation was successful
    self.StatusBar.showMessage("Duplicate items have been deleted", 3000)
```

For usability purposes, a message is displayed in the status bar when both deletion functions are complete, informing the user that they have been successfully carried out.



Testing

To test the delete items function, I inserted several items into the book collection to be deleted

PK_Table55	Title	Author	Publisher	Year	Edition	Rare	Rating	Thumbnail
33	ForDeletion1	ForDeletion1	ForDeletion1	1	ForDeletion1	0	1	NULL
34	ForDeletion2	ForDeletion2	ForDeletion2	2	ForDeletion2	0	1	NULL
35	ForDeletion3	ForDeletion3	ForDeletion3	3	ForDeletion3	0	1	NULL
36	ForDeletion4	ForDeletion4	ForDeletion4	4	ForDeletion4	0	1	NULL

The first test was to try pressing the delete button without having selected any items. Because I had implemented a presence check during development, the program did not come into any issues during this test.

To make sure the actual deletion process works, I selected the items in the frontend table and pressed the delete button. These are the SQL commands that were executed:

```
DELETE FROM Table55 WHERE PK_Table55 = 33
DELETE FROM Table55 WHERE PK_Table55 = 34
DELETE FROM Table55 WHERE PK_Table55 = 35
DELETE FROM Table55 WHERE PK_Table55 = 36
```

Before deletion:

Capital	Karl Marx	Wordsworth	1867	Modern Greats
ForDeletion1	ForDeletion1	ForDeletion1	1	ForDeletion1
ForDeletion2	ForDeletion2	ForDeletion2	2	ForDeletion2
ForDeletion3	ForDeletion3	ForDeletion3	3	ForDeletion3
ForDeletion4	ForDeletion4	ForDeletion4	4	ForDeletion4
Hamlet	William Shakespeare	Penguin	1603	Essential Shakespeare

After deletion:

Title	Author	Publisher	Year	Edition	Rare	Rating
Capital	Karl Marx	Wordsworth	1867	Modern Greats		
Hamlet	William Shakespeare	Penguin	1603	Essential Shakespeare		
How Much Land Does a Man Need?	Leo Tolstoy	Penguin	1886	Penguin Classics		
In Praise of Disobedience	Oscar Wilde	Verso	1891	Radical Thinkers	Rare	
Like a Thief in Broad Daylight	Slavoj Zizek	Allen Lane	2018	First Edition		
Notes on Nationalism	George Orwell	Penguin	1945	Penguin Modern		
Paradise Lost	John Milton	Collins	1667	Collins Classics	Rare	
Protagoras and Meno	Plato	Penguin	380	Penguin Classics	Rare	
The Night Is Darkening Round Me	Emily Bronte	Penguin	1846	Penguin Classics	Rare	
The Outsider	Albert Camus	Penguin	1942	Penguin Modern Classics	Rare	
The Ragged Trousered Philanthropists	Robert Tressell	Wordsworth	1914	Modern Greats		

As you can see, the delete function has successfully deleted the selected items from the collection.

The test data I used for the delete duplicate items function was a set of items, some of which were identical, others of which were similar but had slight differences from the rest.

PK_Table55	Title	Author	Publisher	Year	Edition	Rare	Rating	Thumbnail
37	Test	Test	Test	1	Test	0	1	NULL
38	Test	Test	Test	1	Test	0	1	NULL
39	Test	Test	Test	1	Test	0	1	NULL
40	Test	Test	Test	1	Test	0	1	NULL
41	Test	TestTest	Test	1	Test	0	1	NULL
42	Test	Test	Test	2	Test	0	1	NULL

The fields highlighted in red are areas where the items slightly differ from the rest of the duplicate items. The program should not delete these items as they are not exact duplicates.

Before deletion:

Protagoras and Meno	Plato	Penguin	380	Penguin Classics	Rare	★★★
Test	Test	Test	1	Test		★
Test	Test	Test	1	Test		★
Test	Test	Test	1	Test		★
Test	Test	Test	1	Test		★
Test	TestTest	Test	1	Test		★
Test	Test	Test	2	Test		★

The SQL query the program constructs:

```
SELECT Title, Author, Publisher, Year, Edition FROM Table55 GROUP BY Title, Author, Publisher, Year, Edition HAVING COUNT(*) > 1
```

After Deletion:

Protagoras and Meno	Plato	Penguin	380	Penguin Classics	Rare	★★★
Test	Test	Test	1	Test		★
Test	TestTest	Test	1	Test		★
Test	Test	Test	2	Test		★

This test was successful as the actual duplicate items were deleted whilst the near duplicates were not.

Review

With this iteration complete, users can now delete selected items from collections and delete any duplicate items. I had not initially intended to include a delete duplicates option, so I did not consider it in the design stage, but I think it's a useful addition to the application which will help with managing large collections. The next iteration will be the system for editing items already added to the collection.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- The option to add and delete items from collections (7)
- The option to delete duplicate items (8)

Iteration 9; Editing Items

The functionality of this iteration is very similar to that of the system for adding items coded in iteration 7. As a result, the `EditItemWindow` object will inherit from the `AddItemWindow` class. The key difference between the two objects is that `EditItemWindow` takes the argument `ItemsForEditing`. This will allow me to pass the item the user has selected in the frontend table into the class, thus telling the program which item needs editing. Below is a method of `OpenCollectionWindow` which is executed when `EditItemBTN` is pressed. It fetches the selected item and creates an instance of `EditItemWindow`.

```
# This function is executed when the user presses EditItemBTN
# It retrieves the selected row from CollectionTable and passes it into a new instance
  of EditItemWindow as an argument
def EditItemMethod(self):
    # Stores the selected row
    SelectedRowsArray = []
    # Stores the id corresponding to each selected row
    SelectedRowsIDArray = []
    # Iterate through selected rows in CollectionTable and append them to the list
    for x in self.CollectionTable.selectionModel().selectedRows():
        SelectedRowsArray.append(x.row())
    # Iterate through the contents of SelectedRowsArray and retrieve the corresponding
      primary key by getting the contents of the first (hidden) column
    for y in range(0, len(SelectedRowsArray)):
        SelectedRowsIDArray.append(
                            self.CollectionTable.item(SelectedRowsArray[y], 0).text()
                                   )
    # If the user hasn't selected an item don't continue
    if len(SelectedRowsIDArray) == 0:
        pass
    else:
        # Create EditItemWindow instance
        # EditItemWindow takes the same parameters of AddItemWindow as well as
          ItemsForEditing which in this case is the first value in SelectedRowsIDArray
        self.EditItemWindowInstance = EditItemWindow.EditItemWindow(
                                        MyCursor=self.MyCursor,
                                        MyDB=self.MyDB,
                                        ActiveUserID=self.ActiveUserID,
                                        OpenCollectionID=self.OpenCollectionID,
                                        OpenCollectionTable=self.OpenCollectionTable,
                                        widget = self.EditItemBTN,
                                        ItemsForEditing = SelectedRowsIDArray[0]
                                               )
        # Connect signal to slot
        self.EditItemWindowInstance.ItemAddedSignal.connect(self.ItemAddedSlot)
```

`EditItemWindow` inherits from `AddItemWindow`. The program calls `super()` on it to execute the `init` function of its parent. Once this has been done and the window is initialised as an `AddItemWindow`, the header label of the window is changed and the submit button is connected to the EditItem method rather than the `AddItem` method. The function `PopulateForEditing` is then called. The purpose of this function is to fetch the data for the selected item from the SQL database and then fill the input fields with that data. The user can then edit the item's data and save the edits by pressing the submit button.

Initializing object:

```
# This object inherits from the AddItemWindow class I programmed in Iteration 7
# This subclass takes the new argument ItemsForEditing which is the primary key of the
  item the user has selected in CollectionTable
```

```
# Its appearance is the same as the add item window, but each input field is populated
  with the item's data, for the user to edit
class EditItemWindow(AddItemWindow.AddItemWindow):
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID,
                 OpenCollectionTable, ItemsForEditing, widget = None):
        # super calls the __init__ method of the parent object
        super(EditItemWindow, self).__init__(
                                    MyCursor = MyCursor,
                                    ActiveUserID = ActiveUserID,
                                    MyDB = MyDB,
                                    OpenCollectionID = OpenCollectionID,
                                    OpenCollectionTable = OpenCollectionTable,
                                    widget = widget
                                                )

        # Reassign arguments as attributes
        self.ItemsForEditing = ItemsForEditing

        self.PopulateForEditing()
        # In the init method of the parent object, SubmitBTN was connected to the
          function AddItem()
        # For editing items, this button has to be disconnected from AddItem() and
          connected to EditItem()
        self.SubmitBTN.clicked.disconnect()
        self.SubmitBTN.clicked.connect(self.EditItem)
        # Change window header
        self.HeaderLBL.setText("Edit Item")
```

`PopulateForEditing` is called once the window has been initialized:

```
# This function fills input fields with the selected item's data
# The user can then edit the item's data before pressing submit to save their changes
  to the database
def PopulateForEditing(self):
    # Get data from DB for selected item using its primary key
    self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " WHERE PK_" +
    self.OpenCollectionTable + " = " + self.ItemsForEditing)
    MyResult3 = self.MyCursor.fetchall()[0]
    # Iterate through each field (excluding the primary key, Rating, Rare and
      Thumbnail columns) and set the contents of each text input to the field's
      contents
    for x in range(1, len(MyResult3) - 3):
        self.FieldTextBoxes[x - 1].setText(str(MyResult3[x]))
    # Tick the rare checkbox if the item is rare
    if MyResult3[len(MyResult3) - 3]:
        self.RareCheckBox.setChecked(True)
    # Set the rating slider's value to the integer stored in the Rating column
    self.RatingSlider.setValue(MyResult3[len(MyResult3) - 2])
    # Change text of label showing which image has been selected
    # This object uses the UploadImage method of the parent object
    self.UploadedImageLBL.setText("Change Image")
```

`EditItem` is executed when the user presses `EditItemBTN`. It works in a similar way to `AddItem`, but it uses an `UPDATE` statement rather than `INSERT`:

```
# This method is called when the user presses SubmitBTN
# It fetches the edited data for the item and updates the corresponding entry in the
  SQL DB to save the changes
def EditItem(self):
    # Proceed if the presence check returns TRUE
    if self.PresenceCheck():
        # Get column names
        self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
```

```
        MyResult4 = self.MyCursor.fetchall()
        # Construct UPDATE statement
        SQLStatement = "UPDATE " + self.OpenCollectionTable + " SET "
        # Loop through list of column names and add to SQL statement
        for x in range(1, len(MyResult4) - 3):
            # %s operator is used for string substitution
            SQLStatement += "`" + MyResult4[x][0] + "` = %s, "
        SQLStatement += "Rare = %s, Rating = %s"
        # If the image has been changed, add that to the UPDATE statement
        if self.ContainsImage:
            SQLStatement += ", Thumbnail = %s"
        # Limit SQL operation to selected item using its primary key
        SQLStatement += " WHERE PK_" + self.OpenCollectionTable + " = " +
        self.ItemsForEditing

        # DataForInsertion is the tuple which stores all the data retrieved from the
          user input fields
        DataForInsertion = ()
        # Get the contents of each text box in the window and add it to the tuple
        for x in range(0, len(self.FieldTextBoxes)):
            DataForInsertion += (self.FieldTextBoxes[x].text(), )

        # Add the value of the slider for the Rating column
        DataForInsertion += (self.RatingSlider.value(), )
        # If the user has uploaded an image, add the binary data to the tuple
        if self.ContainsImage:
            DataForInsertion += (self.BinaryData, )

        # Execute the SQL command, substituting the contents of the tuple in for each
          of the %s operators
        self.MyCursor.execute(SQLStatement, DataForInsertion)
        # Save changes to database
        self.MyDB.commit()
        # Emit signal telling root window to repopulate CollectionTable
        self.ItemAddedSignal.emit(False)
        # Close EditItemWindow
        self.close()
```

The final stage in this iteration is updating the slot which is connected to `ItemAddedSignal`.

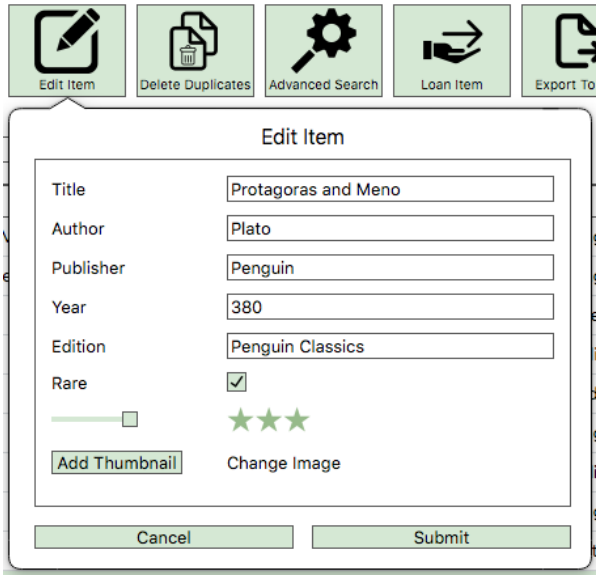
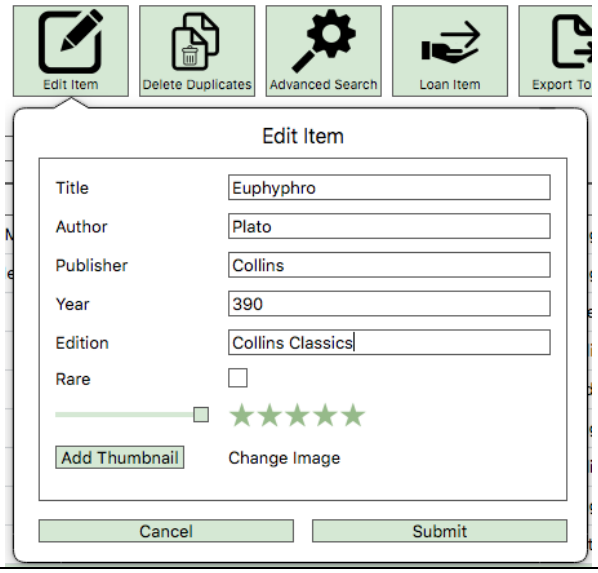
```
# This function is executed when the signal ItemAddedSignal is emitted
def ItemAddedSlot(self, Message):
    # Message is a boolean value passed into ItemAddedSignal
    # The value of Message is passed to this slot when the signal is emitted
    # Message is set to False if the window is an edit item window, so the program
      shows the according message in the status bar
    if not Message:
        self.StatusBar.showMessage("Item has been edited", 3000)
    # Message is set to True if the window is an add item window
    else:
        self.StatusBar.showMessage("New item has been added to collection", 3000)
    # Refresh front end table
    self.PopulateTable()
```

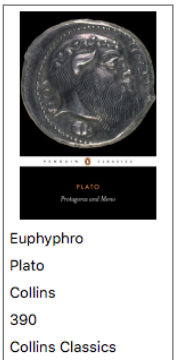
The slot will now display a different message, depending on whether the user is adding a new item or editing an existing one.

Testing

I have the following item in the book collection to test on:

PK_Table55	Title	Author	Publisher	Year	Edition	Rare	Rating	Thumbnail
13	Protagoras and Meno	Plato	Penguin	380	Penguin Classics	1	3	BLOB

Test	Expected Result	Actual Result
Press `EditItemBTN` without having selected an item	Nothing should happen as the program does not know which item needs editing	This test works as expected
Press `EditItemBTN` with multiple items selected	The program should use the most recently selected item for editing	This test works as expected
Edit an item	The edit item window should be opened. The input fields should be populated with the data for the selected item and the user should be able to change the data for that item. When the user presses submit, the changes should be made in the database and the frontend should be refreshed to display those changes	This is what the window displays when it is first opened:  I changed the title, publisher, year, edition and rating of the item and unchecked the rare property 

		When we press the submit button, these are the changes applied to the database: PK_Table55 | Title | Author | Publisher | Year | Edition | Rare | Rating | Thumbnail 13 | Euphyphro | Plato | Collins | 390 | Collins Classics | 0 | 5 | BLOB And the frontend successfully refreshes to display the changes made to the item:  Euphyphro Plato Collins 390 Collins Classics
Change an item's thumbnail	The binary data stored in the database should be changed to that of the newly uploaded image. The frontend should be refreshed to display the new thumbnail	This method reuses the `UploadImage` method from the window's parent class, so it works as expected
Leave an input field blank	The changes should not be applied to the database and the empty fields should be highlighted with a red border	I was initially programming this presence check from scratch before I realised that I could reuse the method from the parent class to save time and lines of code

Integration testing was simply a case of ensuring that the `EditItemWindow` object was instantiated when `EditItemBTN` was pressed. The signals and slots going between the main window and the edit item window all worked flawlessly, so the iteration has passed integration testing.

Review

During the design stage I had planned to implement the editing system as an entirely new object. However, after observing the similar functionality and appearances of the add item and edit item windows, I decided to implement `EditItemWindow` as an inheritor of `AddItemWindow`. I think this was a positive deviation from the design stage as it is a neater implementation and saved a lot of development time. Users can now add, delete and edit items in their collections which means that I have finished programming the core functionalities of the application. The next stage of development is the collection filtering/sorting panel.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- The option to edit the data for items already stored in a collection (9)

End-User Feedback

With the core functions of the program successfully implemented, I felt it was appropriate to arrange another round of end-user feedback on the program so far. I decided to get Sarah to test the application this time as she was concerned with the clarity of my interface design and the application's usability. This is the email she sent to me after trying out the application:

> *Dear Sami,*
> *I really like what you've done. I think you've done a good job of making the app easy to use and I love how each window has a clear header and all the buttons are appropriately labelled. I also like the little tags for rare items, it helps make them stand out. I came across one bug when editing items. Sometimes, when I would edit an item, the program would apply the change, but it would also add a duplicate item to the collection. Apart from this, I have no other concerns. I look forward to seeing the finished product!*
> *From Sarah*

The bug Sarah came across was resolved by adding the line `self.SubmitBTN.clicked.disconnect()` to the `init` function of `EditItemWindow`. The bug arose because each time `super()` was called on `EditItemWindow`, the `init` function of `AddItemWindow` would be called which connects the submit button to the add item function. This meant that when the submit button of `EditItemWindow` was pressed, the methods `AddItem` and `EditItem` were both called as the button was bound to both functions. By disconnecting the clicked signal for that button, we are ensuring that only the `EditItem` method is run when the button is pressed.

Iteration 10; Filtering Collections

During this iteration, I will be coding the panel on the left-hand side of the main window which displays a breakdown of the current collection into different categories which will then allow the user to filter their collection. This panel will contain a tree widget. There will be a parent item in this tree for each column name/property within the collection. E.g., for the books collection there would be 7 parent widgets: Title, Author, Publisher, Year, Edition, Rare and Rating. Each of these parent widgets will then have children, one for each type of entry under that parent column. For example, if the books collection contained three books published by Penguin, four books published by Wordsworth and one book published by Collins, this is how the tree diagram should be structured:

```
Publisher
  |--Penguin (3)
  |--Wordsworth (4)
  |--Collins (1)
```

When the user selects a child item, the corresponding rows in the front-end PyQt table should be highlighted. E.g., if the user selects Penguin in the above example, every item in the collection published by Penguin should be highlighted. I began the iteration by adding the following code to the `InitUI` method of the `OpenCollection` class:

```
# Create groupbox for displaying sorting TreeWidget
self.SortingGroupBox = QGroupBox()
self.SortingGroupBox.setFixedWidth(150)
# Set layout for group box
self.SortingLayout = QVBoxLayout()
self.SortingLayout.setContentsMargins(0, 0, 0, 0)
# Initialise tree widget
# The tree widget will have parent items for each column/property within the collection
# These parents will then have child items for each category within that column
self.SortingTree = QTreeWidget()
# Style tree widget
self.SortingTree.setStyleSheet("QTreeWidget { selection-color: black; }")
self.SortingTree.verticalScrollBar().setStyleSheet("QScrollBar { border-right: 0px; border-
top: 0px; border-bottom: 0px; } QScrollBar::handle { border-top: 0px; border-bottom: 0px;
}")
self.SortingTree.setObjectName("BorderlessWidget")
# Hide tree widget headers
self.SortingTree.setHeaderHidden(True)
self.SortingTree.setAttribute(Qt.WA_MacShowFocusRect, 0)
# Connect the signal emitted when the selected item in the tree widget changes to a slot
self.SortingTree.itemSelectionChanged.connect(self.SortingTreeClicked)
# Add tree widget to layout
self.SortingLayout.addWidget(self.SortingTree)
self.SortingGroupBox.setLayout(self.SortingLayout)
# Add group box to main window layout
self.OpenCollectionHBL1.addWidget(self.SortingGroupBox)
# The sorting panel is hidden on startup, the user can show it through the view menu
self.SortingGroupBox.hide()
```

This code initialises a group box and the tree widget which will display the filtering options. This panel is added to the window's horizontal layout, so it displays to the left of `CollectionTable`. With the widgets and their layouts initialised, I then had to populate the tree widget by adding the following code to the `PopulateTable` method:

```
# If the sorting tree widget is not hidden, populate it
if not self.SortingTree.isHidden():
    # 3D list containing each parent and its corresponding children
    self.SortingTreeKey = []
    # Clear the tree's contents before repopulating it
    self.SortingTree.clear()
    # Iterate through the list containing each column in the table
```

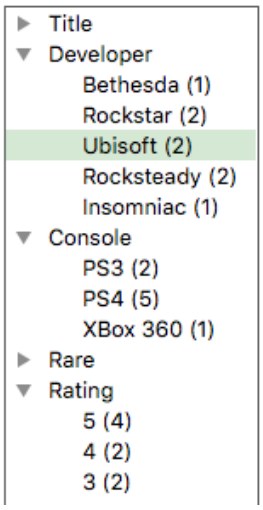
```
    for x in range (1, len(MyResult1) - 1):
        # Create a parent item for each column in the table (excluding the thumbnail and
          primary key columns) and add them to the tree widget
        Temp1 = QTreeWidgetItem(self.SortingTree, [MyResult1[x][0]])
        # Add the parent to the list
        self.SortingTreeKey.append([Temp1])
        # SQL query which returns each entry under the specified column along with the
          amount of times it occurs in the table
        self.MyCursor.execute("SELECT " + str(MyResult1[x][0]) + ", COUNT(*) FROM " +
        self.OpenCollectionTable + " GROUP BY " + str(MyResult1[x][0]))
        MyResult3 = self.MyCursor.fetchall()
        # Iterate through the results of the query, creating a child item for each entry
          and adding it to the tree widget
        for y in range(0, len(MyResult3)):
            # Create child item and add to tree widget, the string passed in is the name
              with the number of time it occurs appended to the end
            Temp2 = QTreeWidgetItem(self.SortingTreeKey[x - 1][0], [str(MyResult3[y][0]) +
            " (" + str(MyResult3[y][1]) + ")"])
            Temp2.setData(0, Qt.UserRole, MyResult3[y][0])
            # Add child to 3D list in the same index as its parent
            self.SortingTreeKey[x - 1].append(Temp2)
```

This algorithm begins by creating a parent item for each column/property in the collection. It does this using the SQL command `SHOW FIELDS FROM [table name]` which returns the name of each column in the table. The program excludes the primary key and thumbnail columns.

It then uses the `GROUP BY` clause to count the number of times a value appears in each column. The results of this query are then used to create child widgets under each parent. Because the widgets are algorithmically generated, they all have to be stored in the list `SortingTreeKey` for future reference. `SortingTreeKey` is a 3D array. Each index is structured in the following way:

```
[parent widget, child widget 1, child widget 2, child widget 3, ….]
```

An index in the following form is created for each parent widget. This is the result in my video game, book and movie collections:

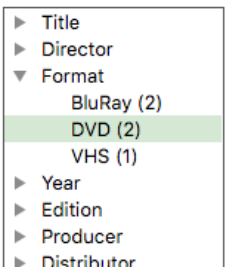


The application can now break each collection down into its component parts and display them in a tree widget. The next step is to add functionality to the tree widget by applying filters when a child

item is selected. In the `InitUI` method, the tree widget's `itemSelectionChanged` signal was connected to the following method which highlights items in the front-end table based on the selected item in the tree widget:

```
# This slot is executed whenever a new item is selected in SortingTree
# It highlights all the items which correspond with the user's selection
def SortingTreeClicked(self):
    # Get the current selected item
    SelectedItem = self.SortingTree.currentItem()
    # If the currently selected item is a parent item do not proceed
    if SelectedItem.parent() is None:
        pass
    # If the selected item is a child item, proceed
    else:
        # Get the child item's parent
        SelectedItemParent = SelectedItem.parent()
        # Select the primary key of all entries where the contents of the parent column is
          the same as the selected child
        self.MyCursor.execute("SELECT PK_" + self.OpenCollectionTable + " FROM " +
        self.OpenCollectionTable + " WHERE " + str(SelectedItemParent.text(0)) + " = '" +
        str(SelectedItem.data(0, Qt.UserRole)) + "'")
        # Reassign query result as list, not tuple
        MyResult1 = list(set(self.MyCursor.fetchall()))

        # List for storing all the rows in the frontend table which correspond with the
          results of the SQL query
        SearchResult = []
        # Clear selection
        self.CollectionTable.clearSelection()
        # Set the table's selection mode so that multiple rows can be selected at once
        self.CollectionTable.setSelectionMode(QAbstractItemView.MultiSelection)

        # Iterate through all the rows in the frontend table and add each row which
          corresponds with the SQL query results to the list
        for z in range(0, self.CollectionTable.rowCount()):
            for x in range(0, len(MyResult1)):
                if self.CollectionTable.item(z, 0).text() == str(MyResult1[x][0]):
                    SearchResult.append(self.CollectionTable.item(z, 0))

        # Iterate through the contents of SearchResult and highlight the rows
        for z in range(0, len(SearchResult)):
                self.CollectionTable.selectRow(SearchResult[z].row())

        # Return table's selection mode to default
        self.CollectionTable.setSelectionMode(QAbstractItemView.ExtendedSelection)
```

The method works by querying the database for the primary key of all the entries into the table where the column sharing the name of the parent of the selected item is equal to the selected item. For example, if I selected the item Penguin under the parent Publishers, the query would be "`SELECT PK_Table55 FROM Table55 WHERE Publisher = 'Penguin'`". The results of this query are then used to highlight all the rows in the table which correspond with those results.

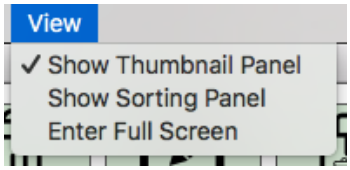


Title	Director	Format	Year	Edition	Produ
The Shining	Stanley Kubrick	BluRay	1980	Horror Classics	Robert Frye
Inception	Christopher Nolan	DVD	2010	Standard	Emma Thon
Clueless	Amy Heckerling	VHS	1995	10th Anniversary	Scott Rudin
Baby Driver	Edgar Wright	DVD	2017	Standard	Tim Bevan
Barry Lyndon	Stanley Kubrick	BluRay	1975	First Edition	Bernard Wil

As you can see in the above image, when I select “DVD” under the “Format” header in my video game collection, the program filters all the items in the collection so that only the films I have on DVD are highlighted.

The final stage of this iteration was creating the “View” menu which displays in the application’s menu bar alongside the file menu. The menu has two options, one for toggling whether the sorting panel should be displayed and hidden and one for showing/hiding the thumbnail panel. I began by creating the menu, creating the actions and adding them to the menu bar.

```
# Create view menu and add to menu bar
self.ViewMenu = self.MainMenu.addMenu(" &View")
# The view menu will have two actions: one for show/hiding the thumbnail panel and one for
showing/hiding the sorting panel
self.ShowThumbnailAction = QAction("Show Thumbnail Panel")
# setCheckable displays a tick next to menu actions when they are active
self.ShowThumbnailAction.setCheckable(True)
self.ShowThumbnailAction.setChecked(True)
# Connect action to method
self.ShowThumbnailAction.triggered.connect(self.ShowThumbnail)
# Add action to menu
self.ViewMenu.addAction(self.ShowThumbnailAction)
# Create action for showing/hiding the sorting panel
self.ShowSortingAction = QAction("Show Sorting Panel")
self.ShowSortingAction.setCheckable(True)
self.ShowSortingAction.setChecked(False)
# Connect action to slot
self.ShowSortingAction.triggered.connect(self.ShowSorting)
# Add action to menu
self.ViewMenu.addAction(self.ShowSortingAction)
```



This is the method executed when `ShowThumbnailAction` is triggered:

```
def ShowThumbnail(self):
    if self.ShowThumbnailAction.isChecked():
        self.ThumbnailGroupBox.show()
    else:
        self.ThumbnailGroupBox.hide()
```

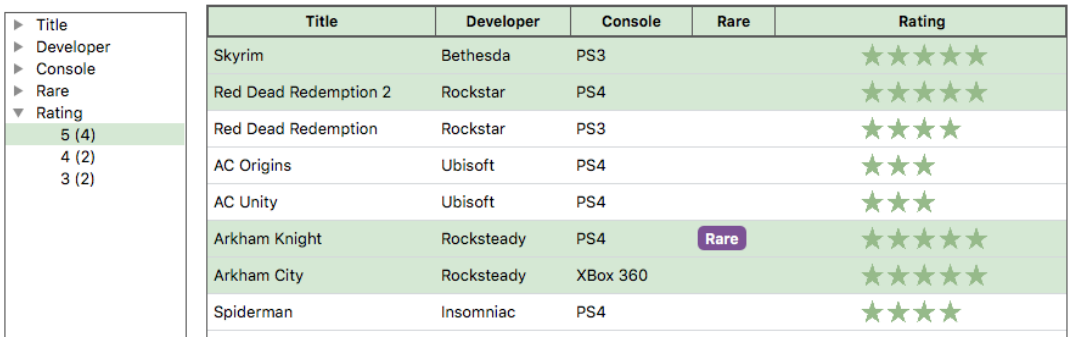
And this is executed when `ShowSortingAction` is triggered:

```
def ShowSorting(self):
    if self.ShowSortingAction.isChecked():
        self.SortingGroupBox.show()
    else:
        self.SortingGroupBox.hide()
```

The user can now hide or show these two panels using the view menu.

Testing

I will be testing this iteration using a video game collection with the columns Title, Developer, Console, Rare and Rating. Below is a series of screenshots of the application correctly filtering collections based on the selected item in the tree widget:

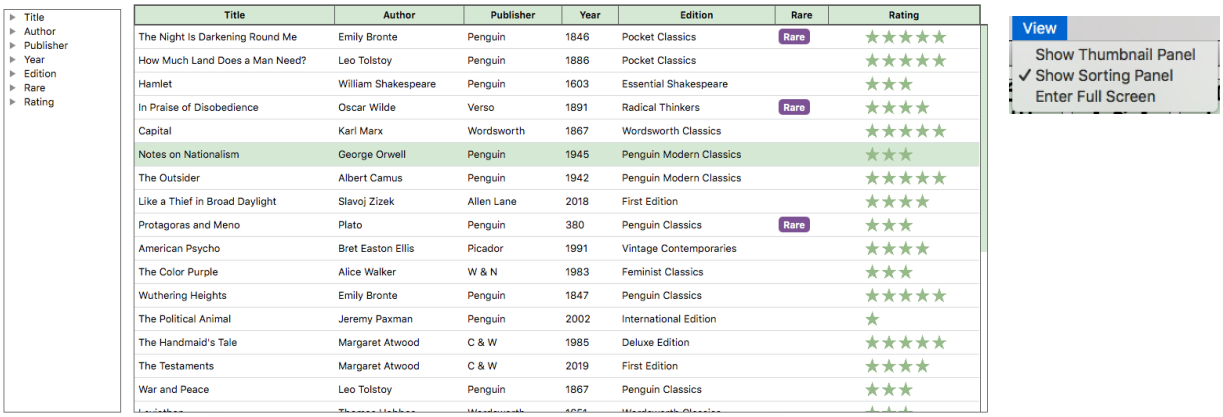


Testing the View menu

Window when the sorting panel is hidden:

Title	Author	Publisher	Year	Edition	Rare	Rating
The Night Is Darkening Round Me	Emily Bronte	Penguin	1846	Pocket Classics	Rare	★★★★★
How Much Land Does a Man Need?	Leo Tolstoy	Penguin	1886	Pocket Classics		★★★★★
Hamlet	William Shakespeare	Penguin	1603	Essential Shakespeare		★★★
In Praise of Disobedience	Oscar Wilde	Verso	1891	Radical Thinkers	Rare	★★★★
Capital	Karl Marx	Wordsworth	1867	Wordsworth Classics		★★★★★
Notes on Nationalism	George Orwell	Penguin	1945	Penguin Modern Classics		★★★
The Outsider	Albert Camus	Penguin	1942	Penguin Modern Classics		★★★★★
Like a Thief in Broad Daylight	Slavoj Zizek	Allen Lane	2018	First Edition		★★★★
Protagoras and Meno	Plato	Penguin	380	Penguin Classics	Rare	★★★
American Psycho	Bret Easton Ellis	Picador	1991	Vintage Contemporaries		★★★★
The Color Purple	Alice Walker	W & N	1983	Feminist Classics		★★★
Wuthering Heights	Emily Bronte	Penguin	1847	Penguin Classics		★★★★★
The Political Animal	Jeremy Paxman	Penguin	2002	International Edition		★
The Handmaid's Tale	Margaret Atwood	C & W	1985	Deluxe Edition		★★★★★
The Testaments	Margaret Atwood	C & W	2019	First Edition		★★★★
War and Peace	Leo Tolstoy	Penguin	1867	Penguin Classics		★★★

Window when the thumbnail panel is hidden:

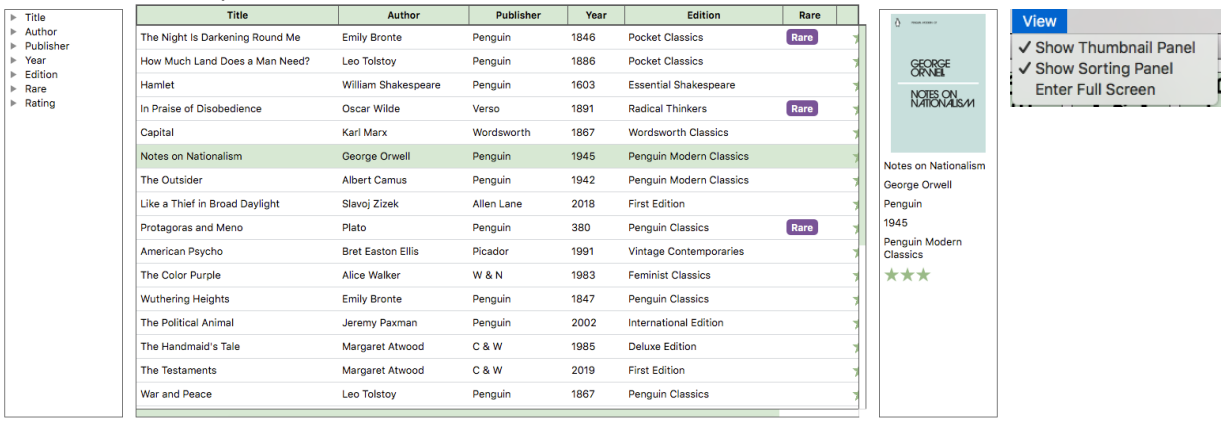


Title	Author	Publisher	Year	Edition	Rare	Rating
The Night Is Darkening Round Me	Emily Bronte	Penguin	1846	Pocket Classics	Rare	★★★★★
How Much Land Does a Man Need?	Leo Tolstoy	Penguin	1886	Pocket Classics		★★★★★
Hamlet	William Shakespeare	Penguin	1603	Essential Shakespeare		★★★
In Praise of Disobedience	Oscar Wilde	Verso	1891	Radical Thinkers	Rare	★★★★
Capital	Karl Marx	Wordsworth	1867	Wordsworth Classics		★★★★★
Notes on Nationalism	George Orwell	Penguin	1945	Penguin Modern Classics		★★★
The Outsider	Albert Camus	Penguin	1942	Penguin Modern Classics		★★★★★
Like a Thief in Broad Daylight	Slavoj Zizek	Allen Lane	2018	First Edition		★★★★
Protagoras and Meno	Plato	Penguin	380	Penguin Classics	Rare	★★★
American Psycho	Bret Easton Ellis	Picador	1991	Vintage Contemporaries		★★★★
The Color Purple	Alice Walker	W & N	1983	Feminist Classics		★★★
Wuthering Heights	Emily Bronte	Penguin	1847	Penguin Classics		★★★★★
The Political Animal	Jeremy Paxman	Penguin	2002	International Edition		★
The Handmaid's Tale	Margaret Atwood	C & W	1985	Deluxe Edition		★★★★★
The Testaments	Margaret Atwood	C & W	2019	First Edition		★★★★
War and Peace	Leo Tolstoy	Penguin	1867	Penguin Classics		★★★

Window when both panels are hidden:

Title	Author	Publisher	Year	Edition	Rare	Rating
The Night Is Darkening Round Me	Emily Bronte	Penguin	1846	Pocket Classics	Rare	★★★★★
How Much Land Does a Man Need?	Leo Tolstoy	Penguin	1886	Pocket Classics		★★★★★
Hamlet	William Shakespeare	Penguin	1603	Essential Shakespeare		★★★
In Praise of Disobedience	Oscar Wilde	Verso	1891	Radical Thinkers	Rare	★★★★
Capital	Karl Marx	Wordsworth	1867	Wordsworth Classics		★★★★★
Notes on Nationalism	George Orwell	Penguin	1945	Penguin Modern Classics		★★★
The Outsider	Albert Camus	Penguin	1942	Penguin Modern Classics		★★★★★
Like a Thief in Broad Daylight	Slavoj Zizek	Allen Lane	2018	First Edition		★★★★
Protagoras and Meno	Plato	Penguin	380	Penguin Classics	Rare	★★★
American Psycho	Bret Easton Ellis	Picador	1991	Vintage Contemporaries		★★★★
The Color Purple	Alice Walker	W & N	1983	Feminist Classics		★★★
Wuthering Heights	Emily Bronte	Penguin	1847	Penguin Classics		★★★★★
The Political Animal	Jeremy Paxman	Penguin	2002	International Edition		★
The Handmaid's Tale	Margaret Atwood	C & W	1985	Deluxe Edition		★★★★★
The Testaments	Margaret Atwood	C & W	2019	First Edition		★★★★
War and Peace	Leo Tolstoy	Penguin	1867	Penguin Classics		★★★

Window when no panels are hidden:



Title	Author	Publisher	Year	Edition	Rare
The Night Is Darkening Round Me	Emily Bronte	Penguin	1846	Pocket Classics	Rare
How Much Land Does a Man Need?	Leo Tolstoy	Penguin	1886	Pocket Classics	
Hamlet	William Shakespeare	Penguin	1603	Essential Shakespeare	
In Praise of Disobedience	Oscar Wilde	Verso	1891	Radical Thinkers	Rare
Capital	Karl Marx	Wordsworth	1867	Wordsworth Classics	
Notes on Nationalism	George Orwell	Penguin	1945	Penguin Modern Classics	
The Outsider	Albert Camus	Penguin	1942	Penguin Modern Classics	
Like a Thief in Broad Daylight	Slavoj Zizek	Allen Lane	2018	First Edition	
Protagoras and Meno	Plato	Penguin	380	Penguin Classics	Rare
American Psycho	Bret Easton Ellis	Picador	1991	Vintage Contemporaries	
The Color Purple	Alice Walker	W & N	1983	Feminist Classics	
Wuthering Heights	Emily Bronte	Penguin	1847	Penguin Classics	
The Political Animal	Jeremy Paxman	Penguin	2002	International Edition	
The Handmaid's Tale	Margaret Atwood	C & W	1985	Deluxe Edition	
The Testaments	Margaret Atwood	C & W	2019	First Edition	
War and Peace	Leo Tolstoy	Penguin	1867	Penguin Classics	

The only change I made during testing was adding the condition:

```
if not self.SortingTree.isHidden()
```

to the code populating the tree widget in the `PopulateTable` method. I added this if statement so that the application does not waste time populating the sorting panel if it has not been set to visible in the View menu. This helps improve the efficiency of the program by cutting out unneccessary steps.

Review

During this iteration I implemented a sorting/filtering panel which displays in the main open collection window. It breaks the collection down into its properties and under each property displays the entries for that property and the number of times they occur. When the user selects an item in this tree widget, the collection is filtered to only highlight items in the main table which belong to the selected group. I also created menu actions for showing or hiding the thumbnail and filtering panels. Testing during this iteration involved ensuring that the View menu and filtering functions work flawlessly. The implementation of this iteration didn't change from what I had devised during the design stage. The next stage in development will be the advanced search feature.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- The option to filter/sort collections (12)

Iteration 11; Advanced Search

During this iteration, I will program the advanced search feature which will allow users to construct precise queries as opposed to the generalised searches conducted by the simple search function. The advanced search will be implemented as a query builder, with users being able to add conditions for their search query and set certain filters for search results. When approaching this iteration, I realised that the UI mock-up I created in the design stage, whilst being usable, was too primitive to support the complex queries users will most likely want to construct, so I began this iteration conceiving of a new method for building queries. The system I ended up settling on was a system of condition widgets which the user could add to a scroll area in order to construct a query. For each condition, the user would input the string/integer they wanted to search for; select the column they would like to search for that data within; choose whether they would like a partial or exact match to the search data and select a logical operator to precede their search condition (AND/OR).

The advanced search is initialised when `AdvancedSearchBTN` in the toolbar is clicked. It is similar to `AddItemWindow` in that it is displayed below the button which is clicked to initialise it and it uses a similar image as the outline/background.

```
# This is the object which contains the advanced search window
# It inherits from QWidget and takes the same arguments as AddItemWindow
class AdvancedSearch(QWidget):
    # This signal is emitted when a successful search has been carried out
    SearchCompleteSignal = pyqtSignal(list)
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID,
                 OpenCollectionTable, widget = None):
        super(AdvancedSearch, self).__init__()

        # Set arguments to attributes
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.ActiveUserID = ActiveUserID
        self.OpenCollectionID = OpenCollectionID
        self.OpenCollectionTable = OpenCollectionTable

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Initialise main layout
        self.AdvancedSearchVBL1 = QVBoxLayout()
        self.setLayout(self.AdvancedSearchVBL1)

        # Get columns from SQL table
        self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
        MyResult1 = self.MyCursor.fetchall()
        # ConditionList stores the column names
        self.ConditionList = []
        # ConditionDataTypeList stores each columns data type
        self.ConditionDataTypeList = []
        for x in range(1, len(MyResult1) - 1):
            self.ConditionList.append(MyResult1[x][0])
            self.ConditionDataTypeList.append(MyResult1[x][1])
        # Remove the rare column from the list of conditions
        self.ConditionList.remove("Rare")
        # Remove its data type from the list of datat types
        self.ConditionDataTypeList.remove(b'tinyint(1)')

        # This list stored each instance of ConditionWidget for future reference
        self.ConditionWidgetsList = []
```

```
        self.InitUI()

        # Set the window to close when the parent window closes
        self.setAttribute(Qt.WA_QuitOnClose, False)
        # Give the window a transparent background so the outline image can be used as
          the window background
        self.setAttribute(Qt.WA_TranslucentBackground)
        # Make window modal
        self.setWindowModality(Qt.ApplicationModal)
        # Make window frameless and keep it on top of all other windows
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint |
        Qt.WindowStaysOnTopHint))

        # Determine where window should be positioned based on the widget which is
          passed in as an argument
        x = widget.mapToGlobal(QPoint(0, 0)).x()
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        # The window should appear on the screen just below the AddItemBTN which is
          clicked to create it
        self.move(x, y)

        # Show window
        self.show()
```

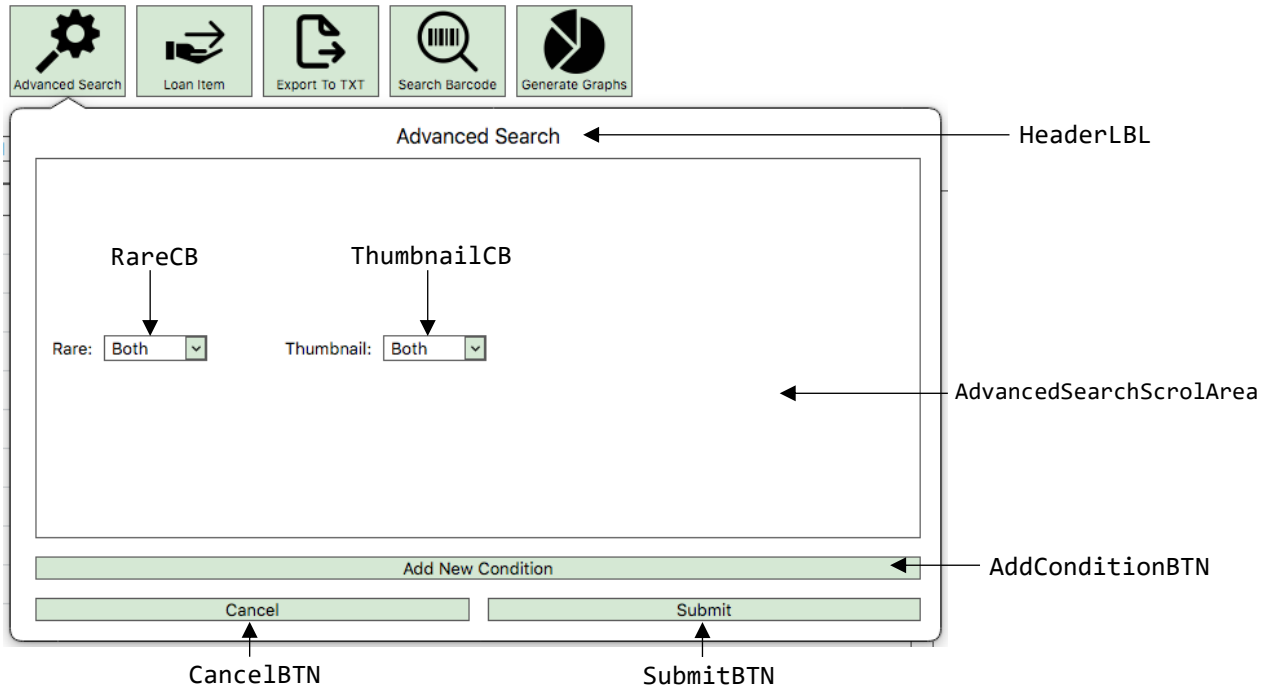
`ConditionList` is an important variable because it stores the names of each of the columns in the collection's SQL table (excluding the primary key). The list is used to populate the combobox of each condition widget with the fields the user can relegate their search to.

This window uses the same repurposed `PaintEvent` as `AddItemWindow` but uses a slightly larger image as the outline. With the window initialised and drawn with the appropriate outline, I progressed to coding the `InitUI` method. This method populates the window with a header, a scroll area for displaying and interacting with condition widgets, a button for adding new widgets and Cancel and Submit buttons.

```
# This method initialises widgets and arranges them in layouts
def InitUI(self):
    # Header label informs the user of the purpose of the window
    HeaderLBL = QLabel("Advanced Search")
    HeaderLBL.setStyleSheet("font-size: 16px;")
    HeaderLBL.setAlignment(Qt.AlignCenter)
    self.AdvancedSearchVBL1.addWidget(HeaderLBL)

    # Users will add conditions to this scroll area
    self.AdvancedSearchScrollArea = QScrollArea()
    # Add scroll area to main layout
    self.AdvancedSearchVBL1.addWidget(self.AdvancedSearchScrollArea)
    self.AdvancedSearchScrollArea.setWidgetResizable(True)
    # Widget which displays within the scroll area
    self.ScrollAreaWidget = QWidget()
    self.ScrollAreaWidget.setObjectName("BorderlessWidget")
    # Initialise layout of ScrollAreaWidget
    self.ScrollAreaLayout = QVBoxLayout()
    self.ScrollAreaLayout.setAlignment(Qt.AlignTop)
    self.OuterLayout = QVBoxLayout()
    self.OuterLayout.addLayout(self.ScrollAreaLayout)
    self.ScrollAreaWidget.setLayout(self.OuterLayout)
    self.AdvancedSearchScrollArea.setWidget(self.ScrollAreaWidget)

    self.CheckBoxLayouts = QHBoxLayout()
    self.OuterLayout.addLayout(self.CheckBoxLayouts)
    # This combobox allows users to filter query results based on their rarity
    self.CheckBoxLayouts.addWidget(QLabel("Rare:"))
```

```
        self.RareCB = QComboBox()
        self.RareCB.addItems(["Both", "True", "False"])
        self.RareCB.setFixedWidth(80)
        self.CheckBoxLayouts.addWidget(self.RareCB)
        self.CheckBoxLayouts.addSpacing(50)
        self.CheckBoxLayouts.addWidget(QLabel("Thumbnail:"))
        # This combobox allows users to filter query results based on whether they have a
          thumbnail or not
        self.ThumbnailCB = QComboBox()
        self.ThumbnailCB.addItems(["Both", "True", "False"])
        self.ThumbnailCB.setFixedWidth(80)
        self.CheckBoxLayouts.addWidget(self.ThumbnailCB)
        self.CheckBoxLayouts.addStretch()

        # This label displays when the advanced search returns no results
        self.ErrorLBL = QLabel("Your search returned no results")
        self.ErrorLBL.setAlignment(Qt.AlignCenter)
        self.ErrorLBL.setStyleSheet("color: red; ")
        self.AdvancedSearchVBL1.addWidget(self.ErrorLBL)
        # The label is initially hidden
        self.ErrorLBL.setHidden(True)

        # This is the button the user presses when they want to add a new condition
        self.AddConditionBTN = QPushButton("Add New Condition")
        # Bind button to function
        self.AddConditionBTN.clicked.connect(self.AddCondition)
        self.AdvancedSearchVBL1.addWidget(self.AddConditionBTN)

        ActionBTNsHBL = QHBoxLayout()
        # CancelBTN closes the advanced search window when it is clicked
        self.CancelBTN = QPushButton("Cancel")
        self.CancelBTN.clicked.connect(self.close)
        # SubmitBTN carries out the advanced search
        self.SubmitBTN = QPushButton("Submit")
        self.SubmitBTN.clicked.connect(self.ConductSearch)
        ActionBTNsHBL.addWidget(self.CancelBTN)
        ActionBTNsHBL.addWidget(self.SubmitBTN)
        self.AdvancedSearchVBL1.addLayout(ActionBTNsHBL)
```

This is what the window currently looks like:



`RareCB` is a combobox with the options “Both”, “True” and “False”. It is used to filter search results once the search has been conducted. If True is selected, the program will only search for rare items, if False is selected, only non-rare items will be returned, if Both is selected, then both rare and non-rare items will be returned. The same applies for `ThumbnailCB`, which filters results based on whether they have a thumbnail or not.

Once the window was initialised and populated, I progressed onto creating the `ConditionWidget` object. Whenever the user presses `AddConditionBTN`, one of these widgets is added to the scroll area. These widgets form the building blocks of the query. The object takes the arguments Conditions, `DataTypes` and First. Conditions and `DataTypes` are lists of the columns in the table and the data type of each column respectively. First is a Boolean value used to indicate whether the widget is the first condition widget or not. This is required because the first condition widget should not have an operator combobox and the user shouldn’t be able to delete it as each query needs at least one condition.

```
# This class inherits from QWidget
# It is the widget which allows users to input search terms, select the column to be
  searched, selects the comparison operator and choose AND or OR
class ConditionWidget(QWidget):
    # This signal is emitted when the user chooses to delete the widget through its
      context menu
    DeleteConditionSignal = pyqtSignal(QObject)
    def __init__(self, Conditions, DataTypes, First):
        super(ConditionWidget, self).__init__()

        # Set arguments to attributes
        # Conditions is a list of all the columns in the collection's SQL table
        self.Conditions = Conditions
        # DataTypes is a list of each column's data type
        self.DataTypes = DataTypes
        # First is a boolean value conveying if the widget is the first condition
          widget or not
        self.First = First

        self.setObjectName("BorderlessWidget")

        # Initialise and set widget layout
        self.ConditionWidgetLayout = QHBoxLayout()
        self.setLayout(self.ConditionWidgetLayout)

        # If the widget isn't the first widget, it will contain an OperatorCB for
          selecting the condition's logical operator
        if not First:
            self.OperatorCB = QComboBox()
            self.OperatorCB.addItems(["or", "and"])
            self.OperatorCB.setFixedWidth(70)
            self.ConditionWidgetLayout.addWidget(self.OperatorCB)

        # This text box is for inputting the search term for this widget
        self.ConditionTB = QLineEdit()
        self.ConditionWidgetLayout.addWidget(self.ConditionTB)

        self.ConditionWidgetLayout.addWidget(QLabel("in"))

        # This combobox contains the name of each column in the table as an option to
          choose from
        # It is used to select the column the user would like to query
        self.ConditionCB = QComboBox()
        self.ConditionCB.addItems(self.Conditions)
```

```
        self.ConditionCB.currentTextChanged.connect(self.CurrentTextChangedSlot)
        self.ConditionCB.setFixedWidth(130)
        self.ConditionWidgetLayout.addWidget(self.ConditionCB)

        # The user has a choice between exact matches using the = operator, or partial
          matches using the LIKE operator
        self.MatchTypeCB = QComboBox()
        self.MatchTypeCB.addItems(["Exact Match", "Partial Match"])
        self.MatchTypeCB.setFixedWidth(130)
        self.ConditionWidgetLayout.addWidget(self.MatchTypeCB)
```

The first method belonging to `ConditionWidget` is the slot executed when the user selects a new item in the condition combobox. The method checks if the newly selected field is of the integer data type and, if it is, it applies a `QIntValidator` to the text box so that the user can only input integers.

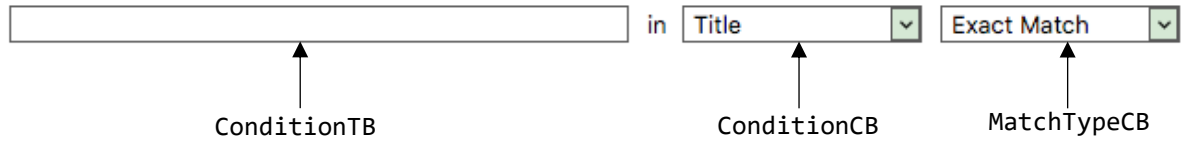
```
# This method is executed when the user selects another item in the combobox
def CurrentTextChangedSlot(self):
    # If the selected item's data type is integer, a validator is set for the input
      box to ensure only integers are inputted
    if self.DataTypes[self.ConditionCB.currentIndex()] == b'int':
        self.ConditionTB.setText("")
        self.ConditionTB.setValidator(QIntValidator())
    # If the datatype is anything else, the validator is not needed
    else:
        self.ConditionTB.setValidator(None)
```

The final method for `ConditionWidget` is the `contextMenuEvent` executed when the user right clicks on the widget. If the Boolean argument First is set to True, the context menu won't show because the first widget can't be deleted, otherwise a context menu is displayed with the option to delete that widget.

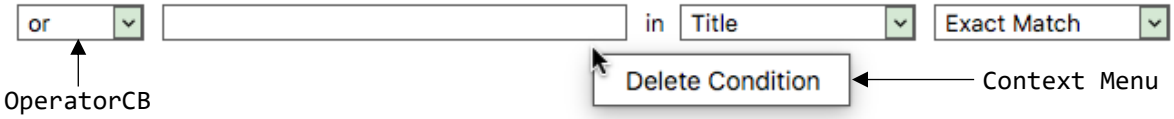
```
# This event occurs when the user right-clicks on the widget
# It displays a contextMenu with the option to delete the condition widget
def contextMenuEvent(self, event):
    # If the widget is the first one, it can't be deleted as each search query needs
      at least one condition to be successful
    if self.First:
        pass
    # Every other widget can be deleted
    else:
        # Initialise context menu
        ConditionWidgetContextMenu = QMenu(self)
        # Add action to context menu
        DeleteAction = ConditionWidgetContextMenu.addAction("Delete Condition")
        Action = ConditionWidgetContextMenu.exec_(self.mapToGlobal(event.pos()))
        # If the user selects the delete option from the context menu...
        if Action == DeleteAction:
            # Emit the delete signal which is received by the AdvancedSearch parent
              class
            self.DeleteConditionSignal.emit(self)
            # Setting the widget's parent to None deletes it
            self.setParent(None)
```

```
# This slot is executed when DeleteConditionSignal is emitted
# It removes the deleted object from the list of condition widgets
def DeleteConditionSlot(self, Sender):
    self.ConditionWidgetsList.remove(Sender)
```

`ConditionWidget` when First is set to True:



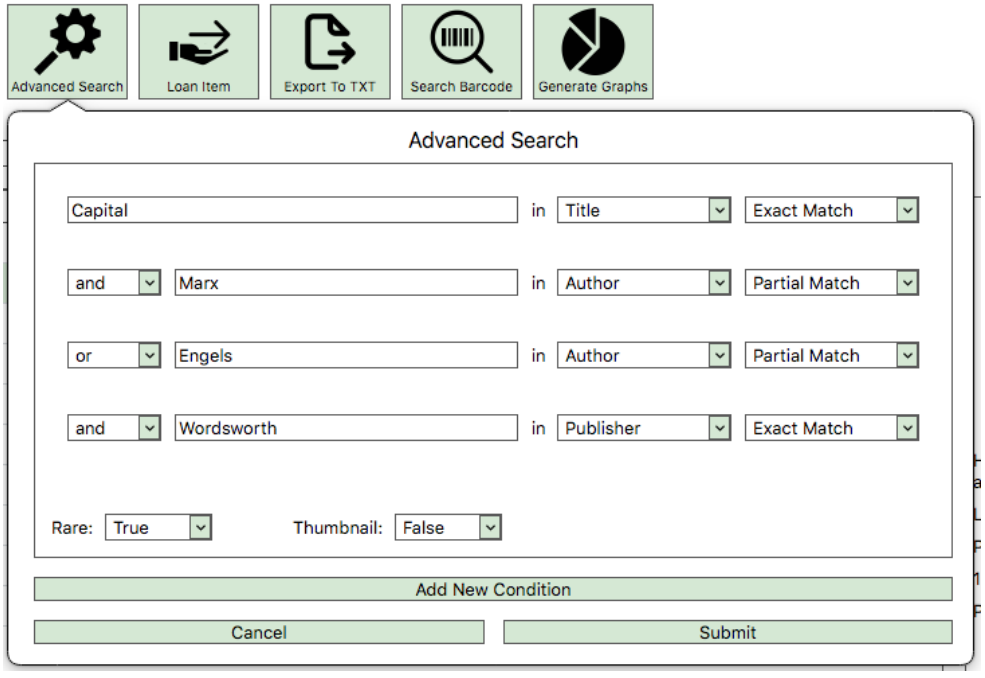
`ConditionWidget` when First is set to False:



When the user clicks `AddConditionBTN`, this is the function which adds the new `ConditionWidget` to the scroll area:

```
# This method is executed when the user clicks AddConditionBTN
def AddCondition(self):
    # Create an instance of ConditionWidget
    # Set arguments Conditions and DataTypes to the lists of column names and data
      types created in the constructor
    ConditionWidgetInstance = ConditionWidget(Conditions = self.ConditionList,
    DataTypes = self.ConditionDataTypeList, First = False)

    # Add instance to list for future reference
    self.ConditionWidgetsList.append(ConditionWidgetInstance)
    # Bind signal emitted when the condition is deleted to the slot
    self.ConditionWidgetsList[len(self.ConditionWidgetsList) - 1]
    .DeleteConditionSignal.connect(self.DeleteConditionSlot)

    # Add widget to scroll area
    self.ScrollAreaLayout.addWidget(ConditionWidgetInstance)
```

The user is now able to construct complex queries by stringing instances of `ConditionWidget` together:



The next stage is to program the window to actually search the collection based on the user's parameters. This function will be implemented as a method of the AdvancedSearch object. Each instance of `ConditionWidget` is stored in a list called `ConditionWidgetsList`. The search method iterates through each instance stored in that list. It uses the user inputs in the text boxes and comboboxes to construct a SQL query. This query is then executed. When the results of the query have been retrieved, they are filtered based on the options the user has selected in `RareCB` and `TumbnailCB`. A signal is then emitted to the parent class, relaying the search results ready for them to be displayed to the user.

```
# This method is executed when the user presses SubmitBTN
# It retrieves the user's inputs and constructs a SQL query from them
def ConductSearch(self):
    # SQLStatement stores the string of the SQL query which will be executed
    SQLStatement = "SELECT * FROM " + self.OpenCollectionTable + " WHERE "
    # Iterate through each condition widget, retrieving the user's inputs and adding
      the new conditions to the query
    for widget in self.ConditionWidgetsList:
        # If the widget is the first widget in the scroll area, there won't be any OR
          or AND statement coming after it
        if widget.First:
            SQLStatement += "`" + widget.ConditionCB.currentText() + "`"
        else:
            # Append the user's choice of operator, then the choice of column to
              search, then the search text to the string
            SQLStatement += widget.OperatorCB.currentText() + " `" +
            widget.ConditionCB.currentText() + "`"
        # Use the appropriate comparison operator based on the user's choice in the
          MatchType combobox
        if widget.MatchTypeCB.currentText() == "Exact Match":
            SQLStatement += " = '" + widget.ConditionTB.text() + "' "
        else:
            SQLStatement += " LIKE '%" + widget.ConditionTB.text() + "%' "
    # Execute SQL query
    self.MyCursor.execute(SQLStatement)
    MyResult2 = self.MyCursor.fetchall()

    # Filter the query results based on the user's rarity filter
    for result in MyResult2:
        # If the user wants results that are both rare and not rare, do nothing to the
          results
        if self.RareCB.currentText() == "Both":
            continue
        # If the user only wants rare results, delete all non-rare results
        elif self.RareCB.currentText() == "True":
            if not result[len(result) - 3]:
                MyResult2.remove(result)
        # If the user only wants non-rare results, delete all rare results
        else:
            if result[len(result) - 3]:
                MyResult2.remove(result)

    # Filter the query results based on the user's thumbnail
    for result in MyResult2:
        # If the user wants results that both have a thumbnail and don't, do nothing
          to the results
        if self.ThumbnailCB.currentText() == "Both":
            continue
        # If the user only wants results with a thumbnail, delete all results that
          don't have a thumbnail
        elif self.ThumbnailCB.currentText() == "True":
```

```
            if result[len(result) - 1] is None:
                MyResult2.remove(result)
        # If the user only wants results without a thumbnail, delete all results that
          do have a thumbnail
        else:
            if result[len(result) - 1] is not None:
                MyResult2.remove(result)

    # If the SQL query returns no results, display the error message
    if len(MyResult2) == 0:
        self.ErrorLBL.show()
        QTimer.singleShot(3000, self.ErrorLBL.hide)
    # If the search returns results, emit the corresponding signal
    else:
        # The signal takes the result of the query as a parameter
        self.SearchCompleteSignal.emit(MyResult2)
        self.close()
```

As you can see, the logic of this function mostly works through string manipulation in order to procedurally generate the SQL query. During development, I streamlined the code in several areas in order to reduce the number of queries being made to the SQL database, which should hopefully speed up its execution. There are several instances of validation carried out on the search result: the first two checks validate the search results based on the user's choice of item in `RareCB` and `ThumbnailCB`, the third check is a presence check which displays an error message to the user if their query has returned no results.

The signal emitted when the search is complete is received by the parent instance of `OpenCollectionWindow` and is bound to the following slot:

```
# This function is executed when the advanced search is complete
def SearchCompleteSlot(self, SearchItems):
    # This method of highlighting search results is similar to the one used for the
      simple search, with a few minor adjustments made
    SearchResult = []
    self.CollectionTable.clearSelection()
    # Set the selection colour for the table to light red in order to differentiate
      search results from normal selection
    self.CollectionTable.setStyleSheet("QWidget::item:selected {background: #FF6961}")
    self.CollectionTable.setSelectionMode(QAbstractItemView.MultiSelection)
    for z in range(0, self.CollectionTable.rowCount()):
        for x in range(0, len(SearchItems)):
            if self.CollectionTable.item(z, 0).text() == str(SearchItems[x][0]):
                SearchResult.append(self.CollectionTable.item(z, 0))
    for z in range(0, len(SearchResult)):
        self.CollectionTable.selectRow(SearchResult[z].row())
    self.CollectionTable.setSelectionMode(QAbstractItemView.ExtendedSelection)
    # Connect the signal emitted when the table is clicked to the slot
    self.CollectionTable.clicked.connect(self.CollectionTableClickedSlot)
    # Show message to user in status bar
    self.StatusBar.showMessage("Search results are highlighted in red", 3000)
```

The slot is similar to the simple search method in that it uses the same method for highlighting search results in the PyQt table. The only difference is that search results are highlighted in red to differentiate them.

An issue that I had to amend was that once I changed the table's selection colour to red, the selection colour would not revert to green when the user tries to select a new item. I remedied the issue by connecting the table's clicked signal to the following slot:

```
# This slot is executed when the table is clicked after an advanced search is
  completed
# It sets the selection colour in the table back to pastel green instead of red
def CollectionTableClickedSlot(self):
    self.CollectionTable.setStyleSheet("QWidget::item:selected {background: #D5E8D4}")
    self.CollectionTable.clicked.disconnect()
```

Testing

I will be testing this iteration using a collection with five similar, but different, items:

Leviathan	1	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	2	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	3	Thomas Hobbes	Wordsworth	1651	Modern Greats		★★★★★
Leviathan	4	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★
Leviathan	5	Thomas Hobbes	Wordsworth	1680	Modern Mediocrity	Rare	★★★★★

Item 1 is rare and has a thumbnail

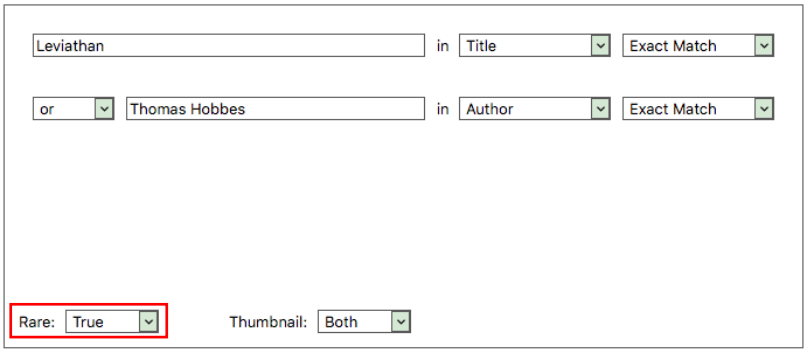
Item 2 is the same as item 1, but it doesn't have a thumbnail

Item 3 doesn't have a thumbnail and isn't rare

Item 4 is the same as item 1 (it has a thumbnail), but it has a different rating

Item 5 is similar to item 1, but the year and edition are different

Test 1: Determining whether the rare filter works or not. The following query should only return items which are rare:



The SQL query the program constructs:

```
SELECT * FROM Table55 WHERE `Title` = 'Leviathan' OR `Author` = 'Thomas Hobbes'
```

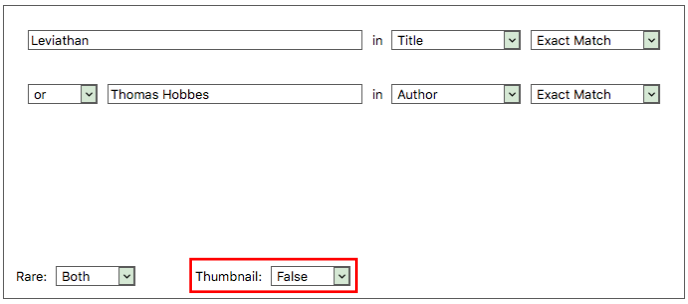
The query doesn't actually filter results based on rarity, this happens later on in the Python frontend

Query result:

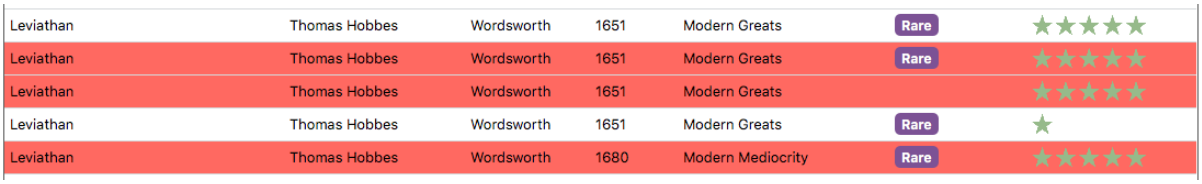
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats		★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★
Leviathan	Thomas Hobbes	Wordsworth	1680	Modern Mediocrity	Rare	★★★★★

As you can see above, the search has returned all of the query results except for those which aren't marked as rare.

Test 2: Testing the thumbnail filter. This time, the query should only return items which don't have a thumbnail:

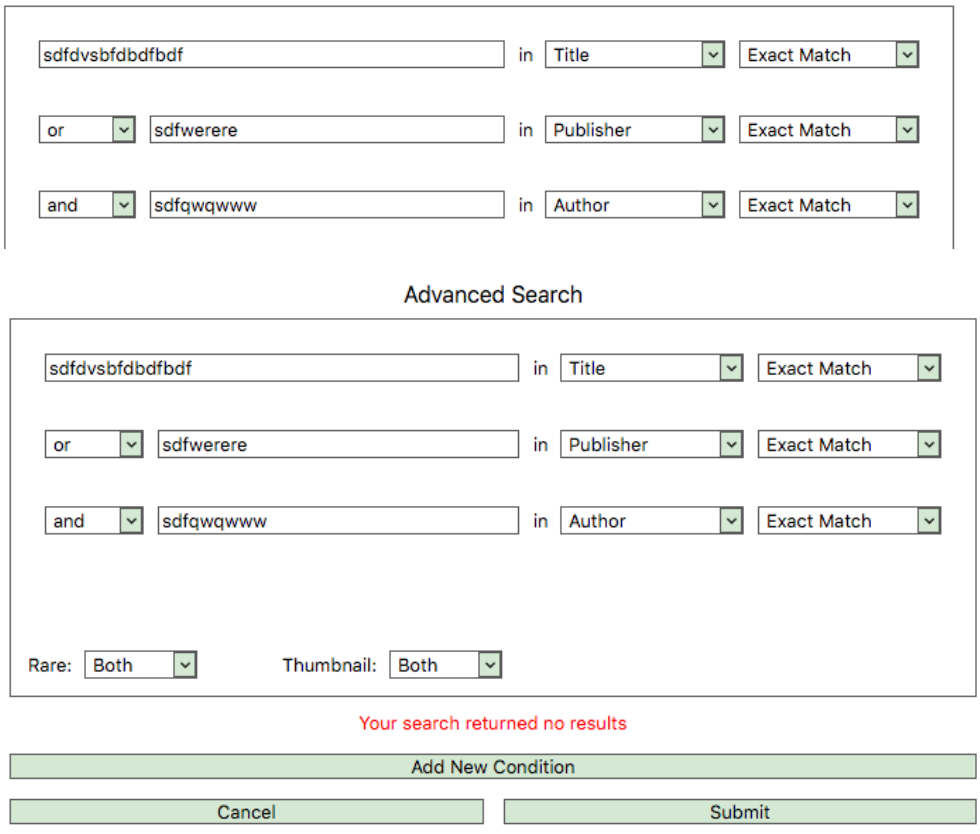


Query result:



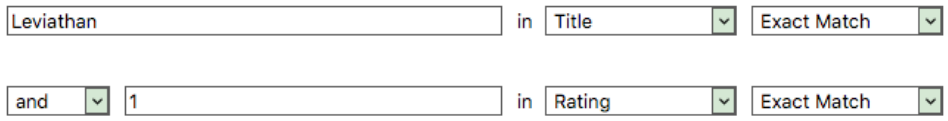
Items 2, 3 and 5 are selected, all of which don't have thumbnails

Test 3: Constructing a query which returns no results. This test should return an error message if no results are found.



The error message is displayed at the bottom of the window, giving the user the opportunity to change their query.

Test 4: A query which only selects item 4 using the and operator:




```
SELECT * FROM Table55 WHERE `Title` = 'Leviathan' AND `Rating` = '1'
```

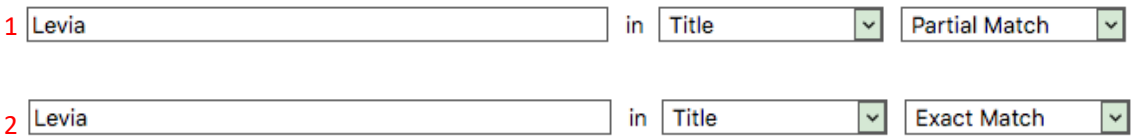
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats		★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★
Leviathan	Thomas Hobbes	Wordsworth	1680	Modern Mediocrity	Rare	★★★★★

Test 5: A query which selects all of the test items except for item 5:


```
SELECT * FROM Table55 WHERE `Title` = 'Leviathan' AND `Author` = 'Thomas Hobbes' AND
`Publisher` = 'Wordsworth' AND `Year` = '1651' AND `Edition` = 'Modern Greats'
```

Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats		★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★
Leviathan	Thomas Hobbes	Wordsworth	1680	Modern Mediocrity	Rare	★★★★★

Test 6: testing the partial match option



Query 1 should select all the test items
Query 2 should select none of the items

Query 1 SQL code:

```
SELECT * FROM Table55 WHERE `Title` LIKE '%Levia%'
```

Query 1 result:

Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats		★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★
Leviathan	Thomas Hobbes	Wordsworth	1680	Modern Mediocrity	Rare	★★★★★

Query 2 SQL code:

```
SELECT * FROM Table55 WHERE `Title` = 'Levia'
```

Query 2 result:

Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats		★★★★★
Leviathan	Thomas Hobbes	Wordsworth	1651	Modern Greats	Rare	★
Leviathan	Thomas Hobbes	Wordsworth	1680	Modern Mediocrity	Rare	★★★★★

Both tests worked as expected. The only changes that needed to be made during testing was placing the column names between backticks (`). This is to prevent against a bug which I ran into when testing. The bug occurs when users use reserved SQL identifiers such as Name and Date as column names. Putting the column names in the backticks allows the DBMS to differentiate between key words and column names. This iteration passed integration testing because it was simply a case of creating an instance of `GraphWindow` in `OpenCollection`.

Review

During this iteration I developed the advanced search feature which allows users to construct precise queries to search their collection with. This function will be most useful for users with massive collections who will need a quick and precise way to search them. The actual implementation of this feature is drastically different to the method I devised during the design stage. I think this system of using condition widgets to build queries is much more modular and affords the user more freedom in constructing their queries. I will now progress onto coding the function which generates and displays graphs based on a user's collection.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- An advanced search function for creating more complicated queries (11)

Iteration 12; Generating Graphs Based on Collections

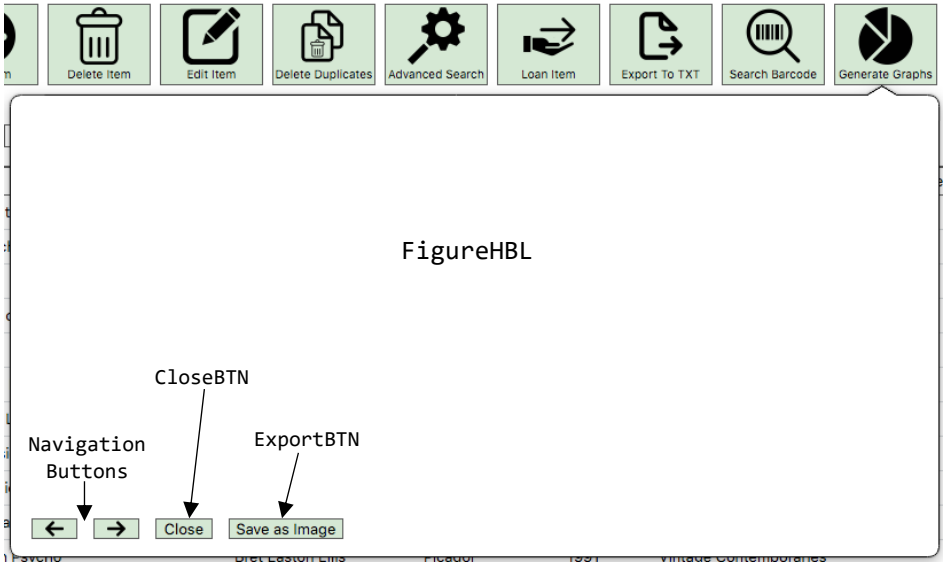
During this iteration I need to code a function which takes data about a collection and presents it to the user in graphical form. My initial plan for implementing this feature in the design stage involved displaying the graphs in a panel below the table of items. The user could then toggle whether the graph pane was hidden or shown by pressing the button in the toolbar. I decided to change this design to instead present the graphs in their own window as I didn't want to take space away from the main list of items which is the centrepiece of the window. The graph window is implemented as a `QWidget` and uses the same `PaintEvent` as `AddItemWindow` (although the outline for this window is much larger). The object also takes the same arguments as `AddItemWindow`. The layout of the window is composed of a horizontal layout for displaying the graph and its legend alongside each other. Below this are left and right navigation buttons, a button for closing the window and a button for exporting the graph as a png. Inserted below is the constructor of the object which initialises all these widgets and layouts and also creates a list of all the columns in the collection's SQL table. This list is used as a list of all the types of graphs the application will create. The thumbnail column is excluded from this list and an extra type of graph called "collection size" is added which will be explained later on.

```
# This object is the window which displays the graphs generated from the user's collection
# It uses matplotlib to plot the graphs
class GraphWidget(QWidget):
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID, OpenCollectionTable,
    widget = None):
        super(GraphWidget, self).__init__()
```

```
        # Reassign arguments as attributes
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.ActiveUserID = ActiveUserID
        self.OpenCollectionID = OpenCollectionID
        self.OpenCollectionTable = OpenCollectionTable

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window attributes (same as AddItemWindow)
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setAttribute(Qt.WA_TranslucentBackground)
        self.setWindowModality(Qt.ApplicationModal)
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint|Qt.WindowStaysOnTopHint))

        # Initialise main window layout
        self.GraphWidgetVBL1 = QVBoxLayout()
        self.setLayout(self.GraphWidgetVBL1)

        # Retrieve column names from table
        self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
        self.GraphTypes = self.MyCursor.fetchall()
        # Remove thumbnails from list
        self.GraphTypes.pop(len(self.GraphTypes) - 1)
        # Add Collection Size category to list
        self.GraphTypes.append(("Collection Size", ))
        # Index keeps track of which graph is currently being displayed
        self.GraphTypesIndex = 1

        # Horizontal layout for displaying the graph/chart and its legend
        self.FigureHBL = QHBoxLayout()

        self.FigureWidget = QWidget()
        self.FigureWidget.setObjectName("BorderlessWidget")
        self.FigureWidgetLayout = QVBoxLayout()
        self.FigureWidgetLayout.setContentsMargins(0, 0, 0, 0)
        self.FigureWidget.setLayout(self.FigureWidgetLayout)
        # Create MatPlotLib figure for displaying the graph/chart
        self.Figure = Figure()
        # Create MatPlotLib canvas and pass figure in as argument
        self.Canvas = FigureCanvas(self.Figure)
        # Add canvas to layout
        self.FigureWidgetLayout.addWidget(self.Canvas)
        self.FigureHBL.addWidget(self.FigureWidget)
        self.GraphWidgetVBL1.addLayout(self.FigureHBL)

        # Create MatPlotLib figure for displaying the legend
        self.LegendFigure = Figure()
        # Create MatPlotLib canvas and pass figure in as argument
        self.LegendCanvas = FigureCanvas(self.LegendFigure)
        self.FigureHBL.addWidget(self.LegendCanvas)

        # Horizontal box layout for displaying the navigation and action buttons
        self.NavigationHBL = QHBoxLayout()
        # BackBTN is used to navigate to the previous graph
        self.BackBTN = QPushButton()
        self.BackBTN.setIcon(QIcon("Resources/LeftArrow2.png"))
        self.BackBTN.clicked.connect(self.PreviousGraph)
        self.BackBTN.setFixedWidth(40)
        # ForwardBTN is used to naviagte to the next graph
        self.ForwardBTN = QPushButton()
        self.ForwardBTN.setIcon(QIcon("Resources/RightArrow2.png"))
        self.ForwardBTN.clicked.connect(self.NextGraph)
        self.ForwardBTN.setFixedWidth(40)
```

```
        # CloseBTN closes the window
        self.CloseBTN = QPushButton("Close")
        self.CloseBTN.clicked.connect(self.close)
        self.CloseBTN.setFixedWidth(50)
        # ExportBTN is used to export the graph as a png
        self.ExportBTN = QPushButton("Save as Image")
        self.ExportBTN.clicked.connect(self.ExportGraph)
        self.ExportBTN.setFixedWidth(100)
        # Label displays message to user when graph is successfully exported as image
        self.ExportStatusLBL=QLabel("Successfully Exported graph as 'AnthologyExport.png'")
        self.ExportStatusLBL.hide()
        # Add buttons to layout
        self.NavigationHBL.addWidget(self.BackBTN)
        self.NavigationHBL.addWidget(self.ForwardBTN)
        self.NavigationHBL.addWidget(self.CloseBTN)
        self.NavigationHBL.addWidget(self.ExportBTN)
        self.NavigationHBL.addWidget(self.ExportStatusLBL)
        self.GraphWidgetVBL1.addLayout(self.NavigationHBL)
        # Push buttons to the left hand side of the layout
        self.NavigationHBL.addStretch()

        # Execute ChooseGraph method to display the first graph
        self.ChooseGraph(self.GraphTypes[self.GraphTypesIndex][0])

        # Position window below GenerateGraphsBTN using widget argument
        x = widget.mapToGlobal(QPoint(0, 0)).x() - 720
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        self.move(x, y)

        # Show window
        self.show()
```

This is what the window looks like at this stage in development:



Before I could actually devise the system for creating and plotting the graphs, I created a system for keeping track of the size of a collection over time. This data will then be used to plot a line chart depicting the size of the collection over time. I did this by creating the following SQL table:

```
CREATE TABLE Sizes
  (
     PK_Sizes             INTEGER(500) NOT NULL AUTO_INCREMENT PRIMARY KEY,
     TimeRecorded         DATETIME,
     Magnitude            INTEGER(500),
     FK_Collections_Users INTEGER(500)
  )
```

Each time an item is added to or deleted from a collection, an entry is made into this table with the size of the collection and the date it was recorded. `FK_Collections_Users` is the foreign key linking entries into this table with entries into the Collections table. The data stored in this table is used to plot the collection size line chart in the graph window. To record size data, I created a new method in `OpenCollectionWindow`. This method is then called at the end of the `AddItem`, `DeleteItem` and `DeleteDuplicates` methods.

```
# Executed when the size of the collection changes (i.e. when items are added or
  deleted)
def UpdateSizes(self):
    # Get the size of the collection
    self.MyCursor.execute("SELECT COUNT(*) FROM " + self.OpenCollectionTable)
    MyResult9 = self.MyCursor.fetchall()
    # Store the size in the database alongside the date that size was recorded at
    self.MyCursor.execute("INSERT INTO Sizes (TimeRecorded, Magnitude,
    FK_Collections_Sizes) VALUES ('" + str(datetime.now()) + "', " +
    str(MyResult9[0][0]) + ", " + str(self.OpenCollectionID) + ")")
    # Save changes to database
    self.MyDB.commit()
```

I also had to update `DeleteCollectionSlot` in `CollectionWindow` so that when a collection is deleted, all the corresponding entries into the Sizes table are also deleted.

The sizes table will now be populated with entries for the size of each collection when it changes:

PK_Sizes	TimeRecorded	Magnitude	FK_Collections_Sizes
1	2020-12-10 18:07:53	0	55
2	2020-10-22 09:22:10	0	57
5	2020-12-11 18:07:53	27	55
6	2020-12-11 18:08:10	28	55
7	2020-12-11 18:08:28	29	55
8	2020-12-11 18:08:38	30	55
9	2020-12-11 18:08:48	28	55
10	2020-12-11 18:08:53	27	55

With this system of recording collection sizes set up, I progressed onto creating and displaying the graphs. The first step was a simple method executed when a new graph needed to be drawn which would determine whether the graph was a pie chart or a line chart. The method takes the parameter Type which is passed in as the data stored in `GraphTypes` at the index `GraphTypesIndex`.

```
# This method determines whether the program needs to draw a line graph or a pie chart
def ChooseGraph(self, Type):
    if Type == "Collection Size":
        self.DrawLineGraph()
    else:
        self.DrawPieChart(Type)
```

The graphs are created using MatPlotLib. The first graph function I programmed was for generating pie charts. The method takes the parameter Type which is passed in from the `ChooseGraph` method. It works by grouping data in a certain column and then calculating the percentage of the total items each group constitutes. These percentages are then passed into the `Axis.pie()` method to draw the chart. The first job for the method to carry out is therefore to calculate these values and collect them in a list:

```
# This function plots a pie chart and creates a legend
# It takes the parameter Type, which is the collection field that it is plotting
def DrawPieChart(self, Type):
    self.FigureWidget.setFixedWidth(380)
    # Clear figure of previous graph
```

```
    self.Figure.clear()
    # Group values stored under the current column and return the number in each group
    self.MyCursor.execute("SELECT " + Type + ", COUNT(*) AS c FROM " +
    self.OpenCollectionTable + " GROUP BY " + Type)
    MyResult1 = self.MyCursor.fetchall()
    # Total is the total number of items being plotted (used to calculate percentages)
    self.Total = 0
    # GraphLabels is a list containing each category in the pie chart
    self.GraphLabels = []
    # Percentages is a list containing the percentage of each category
    self.Percentages = []
    # Count number of items by incrementing Total for each value in MyResult1
    for x in range(0, len(MyResult1)):
        self.Total += MyResult1[x][1]
    # Iterate through MyResult1, calculating the percentage of each category and
      adding category labels to the list
    for y in range(0, len(MyResult1)):
        # Calculate category's percentage of the Total
        self.Percentages.append((MyResult1[y][1] / self.Total) * 100)
        # If the category name is longer than 17 characters, cut it down and add an
          ellipses to the end
        if len(str(MyResult1[y][0])) > 17:
            Label = str(MyResult1[y][0][:17]) + "..."
        else:
            Label = str(MyResult1[y][0])
        # If we are plotting the rarity, we have to replace 1 and 0 with True and
          False for our legend
        if Type == "Rare":
            if MyResult1[y][0]:
                Label = "True"
            else:
                Label = "False"
        # Add the label to the list, percentages are round to the nearest whole number
        self.GraphLabels.append(Label + " (" + str(round(self.Percentages[y],
        int(floor(log10(abs(self.Percentages[y])))))) + "%)")
```

With the percentages and names of each category stored in respective lists, the method goes on to create an Axis object. This object is an attribute of the Figure and is the entity used to actually draw the graph.

```
# Create axis object for plotting values
self.Axis = self.Figure.add_axes((0, 0, 1, 1))
# Set the background colour of the figure to white
self.Figure.set_facecolor("#FFFFFF")
# Plot pie chart using values stored in Percentages list
self.patches, texts = self.Axis.pie(self.Percentages, startangle = 90)
# Set the graph's aspect ration to "equal" to display the pie chart as a perfect
circle rather than an ellipse
self.Axis.axis("equal")
```

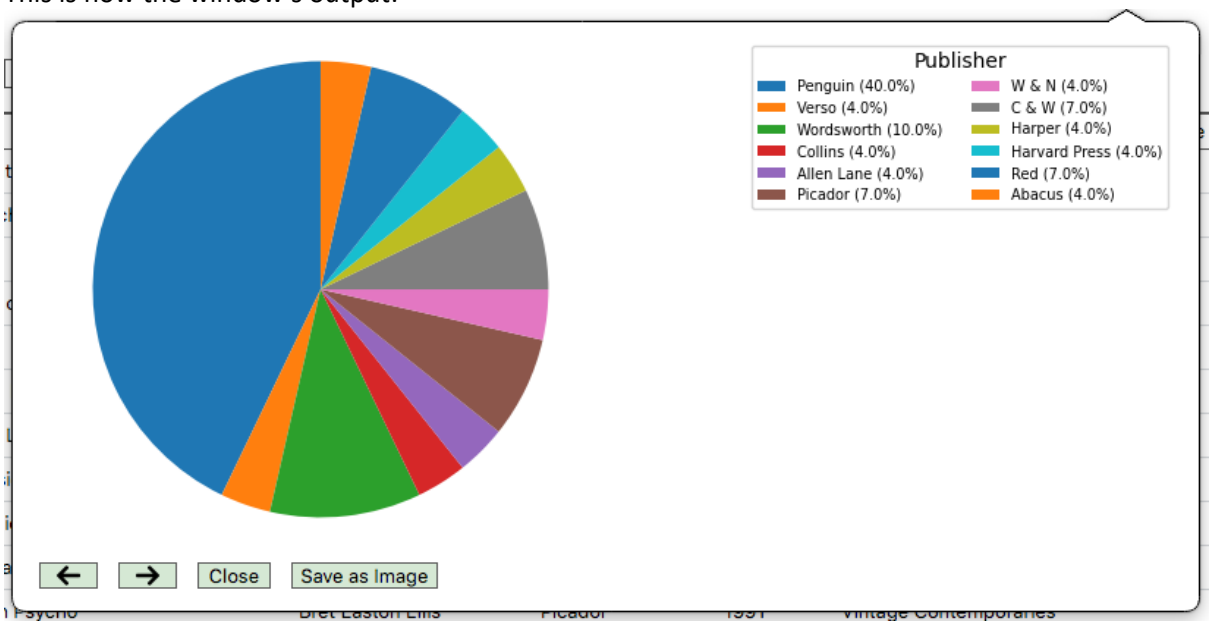
Next, the legend needs to be created. It has its own canvas and figure, and it uses the `GraphLabels` list to get the name of each category.

```
# Clear figure of previous legend
self.LegendFigure.clear()
self.LegendFigure.tight_layout()
# Create legend
self.Legend = self.LegendFigure.legend(self.patches, self.GraphLabels, prop = {"size":
7}, ncol = 2)
# Set legend title to Type parameter
self.Legend.set_title(Type, prop = {"size": 10})
```

Finally, each canvas has to be refreshed in order to display the new graph.

```
# Call draw() method on both canvases to display new graphs
self.Canvas.draw()
self.LegendCanvas.draw()
self.LegendCanvas.show()
```

This is now the window's output:



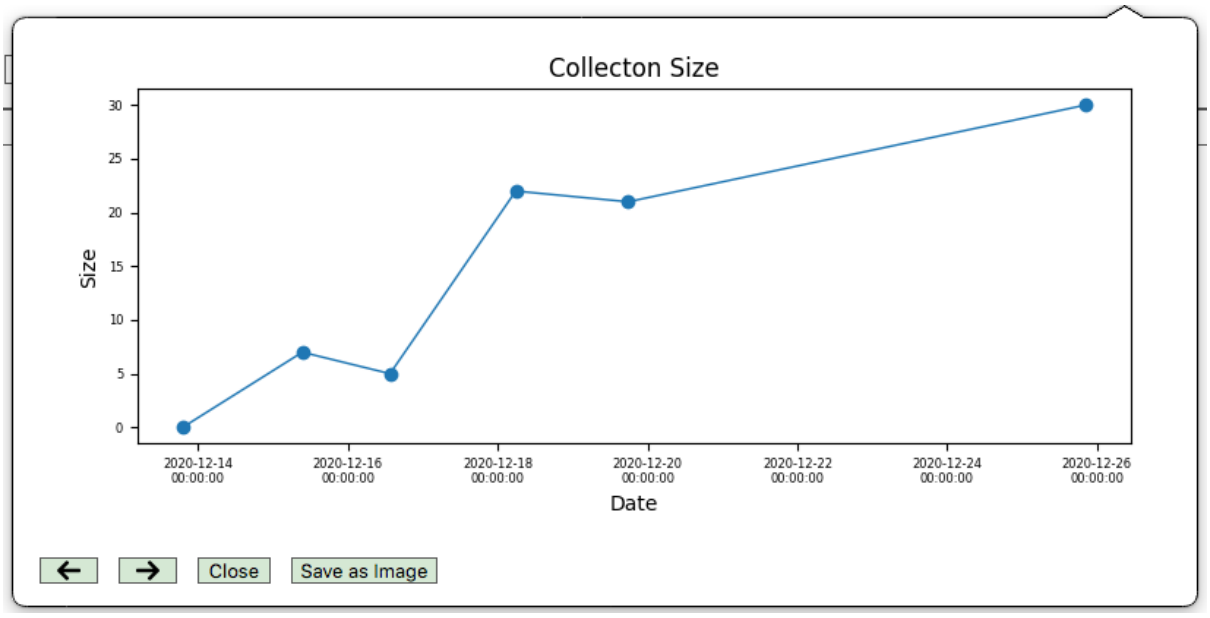
As you can see, the program has taken all the values stored in the Publisher column of the book collection, arranged them into groups of the same publisher, plotted them in a pie chart and created a key for that chart.

With the pie chart function working, the next step was to create the line chart plotting the collection's size. Plotting line graphs works in a similar manner to pie charts, although a key is not required. The method of collecting data is also different: the size and time data are retrieved by collecting all entries into the Sizes table where the foreign key matches with the primary key of that collection in the Collections table. Because the dates are stored in the table as DateTime objects, a formatter will have to be tied to the x axis of the graph to determine how these dates will be displayed.

```
# The method plots a line chart
# It is called when the user wants to view the collection sizes chart
def DrawLineGraph(self):
    self.FigureWidget.setFixedWidth(750)
    # The line chart doesn't need a legend so its canvas is hidden
    self.LegendCanvas.hide()
    # Clear figure of previous graph
    self.Figure.clear()
    # Get all entries into the sizes table where the foreign key corresponds with the
      primary key of the currently open collection
    self.MyCursor.execute("SELECT * FROM Sizes WHERE FK_Collections_Sizes = " +
    str(self.OpenCollectionID))
    MyResult2 = self.MyCursor.fetchall()
    # Dates is a list storing the date when each size was recorded
    self.Dates = []
    # Magnitudes stores the size of the collection at each date
    self.Magnitudes = []
    # Iterate through the results of SQL query and append them to the appropriate list
    for x in range(0, len(MyResult2)):
```

```
            self.Dates.append(MyResult2[x][1])
            self.Magnitudes.append(MyResult2[x][2])
        # Convert MySQL date values to python-compatible datetime objects
        self.Dates = [datetime.strptime(str(d), "%Y-%m-%d %H:%M:%S") for d in self.Dates]
        # Convert python datetime objects to MatPlotLib compatible values
        self.Dates = matplotlibdates.date2num(self.Dates)
        # Create a formatter for the x axis to correctly display the time values
        self.DatesFormat = matplotlibdates.DateFormatter("%Y-%m-%d\n%H:%M:%S")
        # Create axis object for plotting values
        self.Axis = self.Figure.add_subplot()
        self.Figure.set_facecolor("#FFFFFF")
        self.Figure.tight_layout()
        # Set the formatter of the x axis to the newly created formatter
        self.Axis.xaxis.set_major_formatter(self.DatesFormat)
        # Plot dates against sizes as a line chart
        self.Axis.plot(self.Dates, self.Magnitudes, marker = "o", linewidth = 1)
        # Set the graph's title
        self.Axis.set_title("Collecton Size", size = 12)
        # Set the x axis label
        self.Axis.set_xlabel("Date", size = 10)
        # Set the y axis label
        self.Axis.set_ylabel("Size", size = 10)
        self.Axis.tick_params(axis = "both", labelsize = 6)
        self.Figure.tight_layout()

        # Call draw() method on both canvases to display new graphs
        self.LegendFigure.clear()
        self.Canvas.draw()
        self.LegendCanvas.draw()
```

Initially, there were issues with the spacing of the graph. This was remedied by using `add_subplot` instead of `add_axes` when creating the Axis object. This is the output when the size graph is displayed:



Now that the graph is able to successfully generate graphs for different collections based on the data stored under each column, I needed to make the navigation buttons functional to allow users to navigate between different graphs. This was simply a case of writing methods that would change the value of `GraphTypesIndex` and then call `ChooseGraph`.

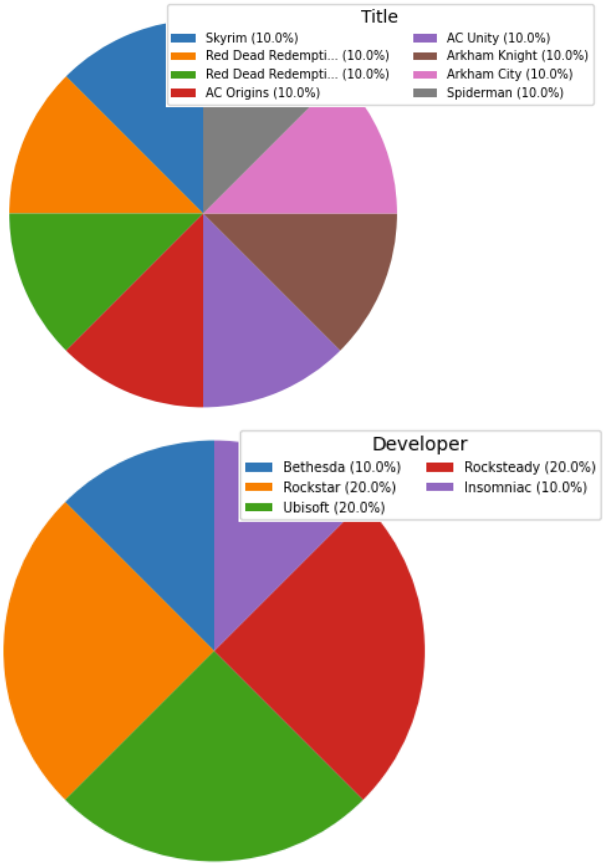
```
# Executed when ForwardBTN is clicked
def NextGraph(self):
    # If the index is at the end of the list, set the index to the beginning of the
      list
    if self.GraphTypesIndex >= len(self.GraphTypes) - 1:
        self.GraphTypesIndex = 1
    # Otherwise, increment the index by 1
    else:
        self.GraphTypesIndex += 1
    # Draw the new graph
    self.ChooseGraph(self.GraphTypes[self.GraphTypesIndex][0])

# Executed when BackBTN is clicked
def PreviousGraph(self):
    # If the index is at the start of the list, set the index to the end of the list
    if self.GraphTypesIndex <= 1:
        self.GraphTypesIndex = len(self.GraphTypes) - 1
    # Otherwise, decrement the index by 1
    else:
        self.GraphTypesIndex -= 1
    # Draw the new graph
    self.ChooseGraph(self.GraphTypes[self.GraphTypesIndex][0])
```

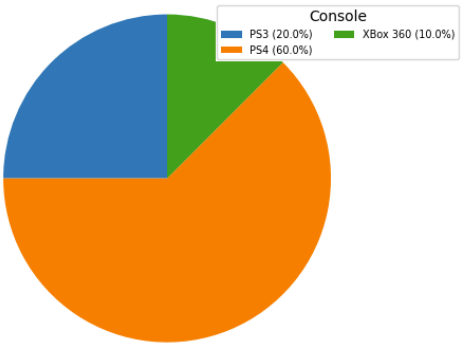
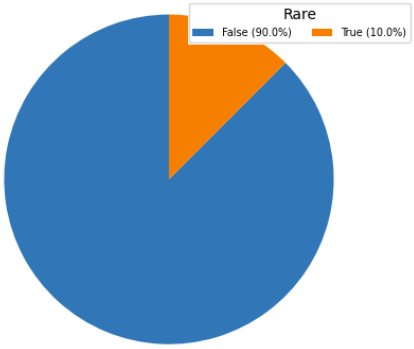
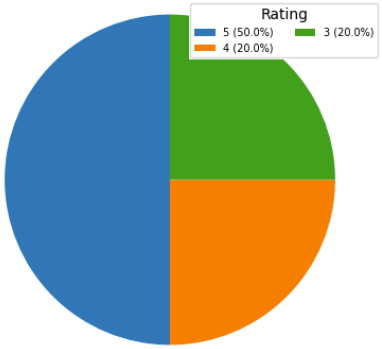
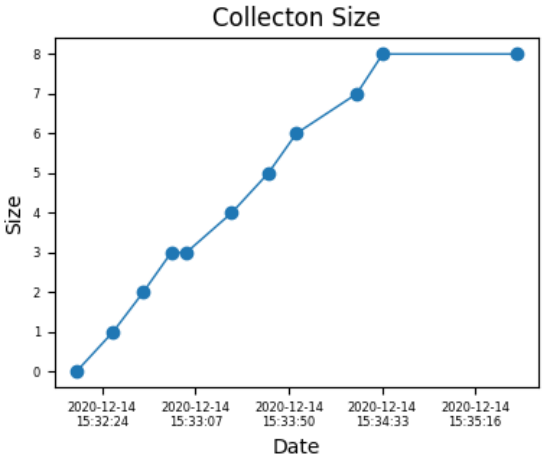
The final method in this iteration is for exporting a graph as a png. This function is executed when the user presses `ExportBTN`. It takes the MatPlotLib canvas and converts it into a `QImage`. The program then opens a file dialog for the user to select the directory to save the image to. The `QImage` is then saved to that directory and a message is displayed in the window informing the user that the collection has been successfully exported. When the user exports a pie chart, two images are created, one for the chart itself and one for the legend. To combine these two images into a single image for exporting, a `QPainter` has to be used.

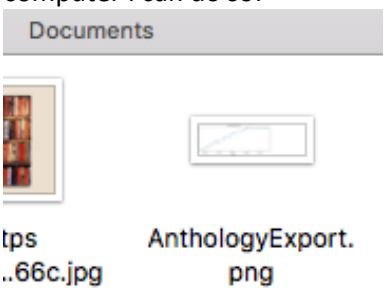
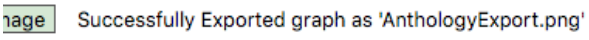
```
# Executed when the user clicks ExportBTN
# It exports the graph currently being displayed as a png
def ExportGraph(self):
    # Open file dialog and allow user to select a directory to export to
    # The file path of this directory is stored under the variable FilePath
    self.FilePath = QFileDialog.getExistingDirectory(None, "Select Export Directory",
                    os.path.abspath(os.sep))

    # If the graph being exported is a line chart, the legend is not required
    if self.GraphTypes[self.GraphTypesIndex] == "Collection Size":
        GraphSize = self.Canvas.size()
        # Convert the matPlotLib canvas into a PyQt QImage
        GraphImage = QImage(self.Canvas.buffer_rgba(), GraphSize.width(),
        GraphSize.height(), QImage.Format_RGBX8888)
        # Save the image to the user-specified directory with the name
          AnthologyExport.png
        GraphImage.save(self.FilePath + "/AnthologyExport.png")

    # If the graph is a pie chart and has an index, QPainter is used to place the two
      images together and export them as a single image
    else:
        # Get dimensions of pie chart canvas
        GraphSize = self.Canvas.size()
        # Convert graph convas into QImage
        GraphImage = QImage(self.Canvas.buffer_rgba(), GraphSize.width(),
        GraphSize.height(), QImage.Format_RGBX8888)
        # Get dimensions of legend canvas
        LegendSize = self.LegendCanvas.size()
        # Convert legend canvas into QImage
        LegendImage = QImage(self.LegendCanvas.buffer_rgba(), LegendSize.width(),
```

```
            LegendSize.height(), QImage.Format_RGBX8888)
            # Create a new QPixmap for the export
            # The width of the new image is the sum of the width of the graph image and
              the legend image
            FinalExport = QPixmap(QSize(GraphImage.width() + LegendImage.width(),
            GraphImage.height() + 5))
            # Start painter, set painter to perform on FinalExport
            Painter = QPainter(FinalExport)
            # Draw the graph image starting at point (0, 0)
            Painter.drawImage(QPoint(0, 0), GraphImage)
            # Draw the legend image alongside the graph image
            Painter.drawImage(QPoint(GraphImage.width(), 0), LegendImage)
            # End painter
            Painter.end()
            # Save image
            FinalExport.save(self.FilePath + "/AnthologyExport.png")

        # Display message to user informing them that the image has been successfully
          imported
        self.ExportStatusLBL.show()
        # Set message to disappear after 3 seconds
        QTimer.singleShot(3000, self.ExportStatusLBL.hide)
```

Testing

Test	Expected Result	Actual Result
Generating pie charts	A pie chart should be created for each column in the collection, breaking down the distribution of types of item within that collection. A legend should also be created. The user should be able to navigate between these graphs using the navigational buttons.	For a video game collection with the columns title, developer and console, these are the pie charts produced: 

		   As you can see, a graph has been produced for each property of the collection and a key has also been produced for each graph. I also tested this function with a multitude of other collections, all of which contained different properties, amounts of data and types of data.
Generating line charts mapping out a collection's size	A line chart should be displayed plotting the size of the collection against the date the size was recorded. The values plotted should come from the Sizes table	For the same video game collection, the following graph was produced: 

Test navigation buttons	When clicked, these buttons should display the previous or next graphs	The program did not initially pass this test because of some logic errors I had made when determining the value of `GraphTypesIndex`. After this bug was dealt with, the test worked as expected
Generate a graph for a collection with no items	The program should display a blank graph canvas as there is no input data to derive a graph from	During testing, doing this would cause the program to crash. This issue was remedied by adding a presence check to the start of the graph drawing methods to ensure that there is actually some data to draw from
Exporting graphs as images	A file dialog should be opened allowing the user to select a directory to save the image to. The program should then convert the graph canvas into a png and save it to that directory	The code successfully saves the graph as a png to the user-specified directory. The only issue encountered was that I initially used `QFileDialog.getOpenFileName` to open the file explorer. The issue with this method was that it allowed the user to select files instead of only directories. Therefore, if the user selected a file instead of a directory, the program would overwrite that file. This bug was remedied by using `QFileDialog.getExistingDirectory` which limits the user to only select directories. Now if I want to export a graph to the Documents folder on my computer I can do so:  The following message is show after a successful export: 

Just like the previous iteration, integration testing was simply a case of ensuring that the `GraphWidget` object was properly instantiated in `OpenCollection`

Review

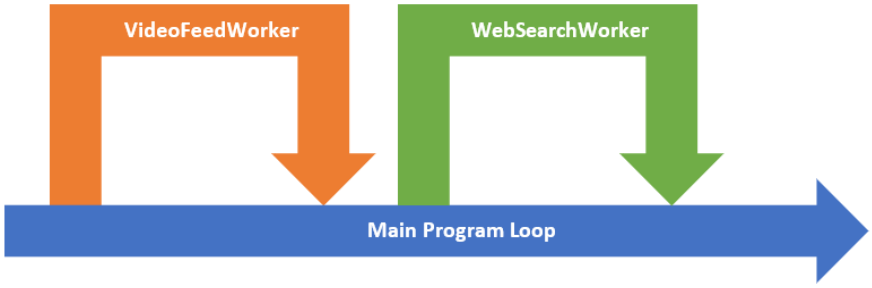
In this iteration, I developed the code which creates graphs based on data from a user's collection, presents them in a window allowing them to navigate between different graphs, and allows users to export graphs as a png to a directory of their choice. I deviated from the UI prototype created in the design stage by presenting the graphs in their own window rather than a panel underneath the list of items. I did this because I didn't want to have to cramp UI elements into a small space and confuse users by having several different panes on the same window at once. Testing of this iteration involved generating different graphs from different collections and ensuring that the export function worked. The next stage in development is the barcode searching feature.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

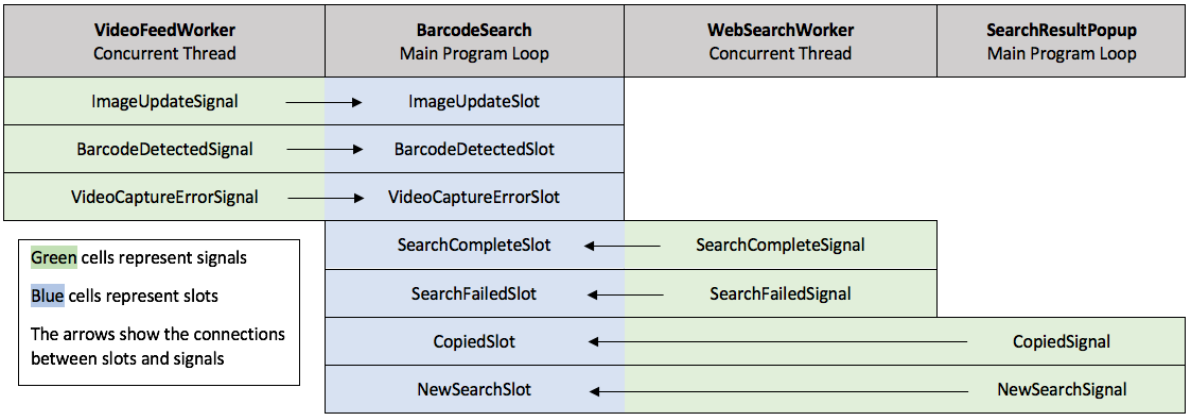
- The ability to generate graphical breakdowns of collections (18)
- The option to export graphs as images (19)

Iteration 13; Barcode Search

During this iteration, I will have to make use of concurrent threads to handle outputting a live video feed to the PyQt window and search online webpages for any detected barcodes. The program will require two threads to run alongside the main program loop: `VideoFeedWorker` and `WebSearchWorker`. Each thread will use signals to relay the results of processing to the main program loop. The diagram below outlines how these threads will run alongside the main loop:



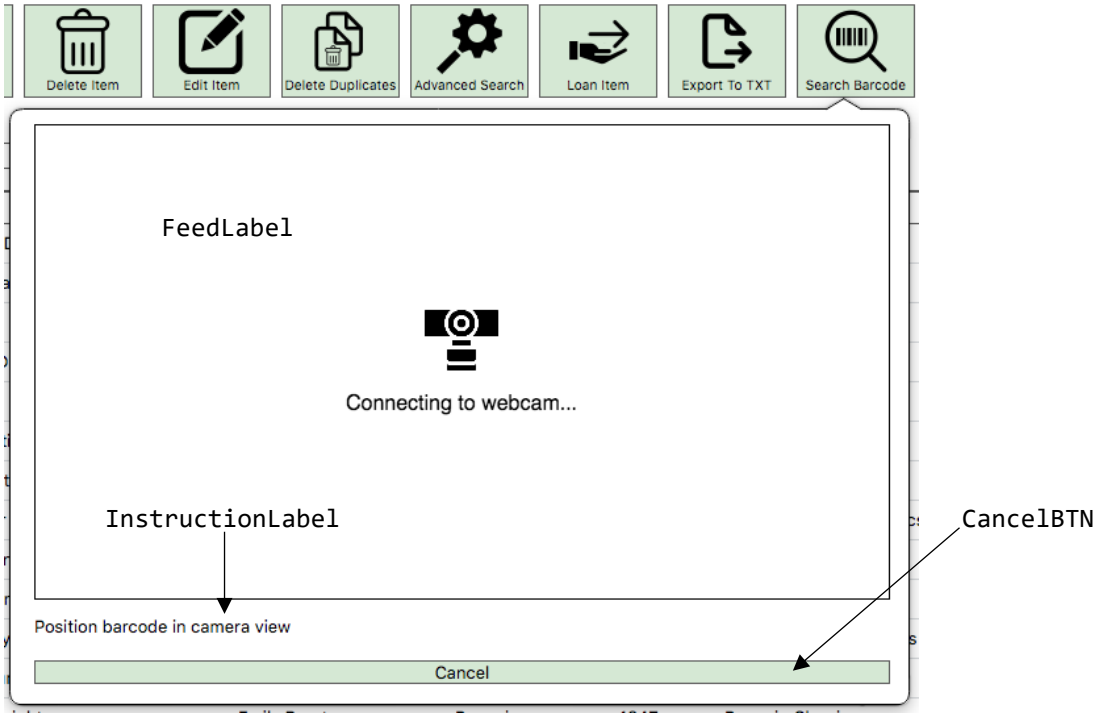
As you can see, `VideoFeedWorker` is created when the barcode search window is initialised. When it detects a barcode in the frame, it will send a signal back to the main program loop. The main program loop will then start the `WebSearchWorker` thread, passing in the detected barcode as an argument. When `WebSearchWorker` has finished processing, it sends a signal back to the main program loop telling it to output the result of the search to the user. Below is a summary of all the signals each thread/object emits and the corresponding slots they are connected to:



`SearchResultPopup` is a popup window used to display the results of the barcode search to the user. The first step in this iteration was to write the constructor of the `BarcodeSearch` object. It initialises all the widgets in the window and starts the `VideoFeedWorker` thread.

```
# This class contains the window which will display the live video feed, an
  instruction label and a cancel button
# The two concurrent threads are initialised in this window
# The only argument it takes is the button widget clicked to create it
class BarcodeSearch(QWidget):
    def __init__(self, widget = None):
        super(BarcodeSearch, self).__init__()

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window attributes
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setAttribute(Qt.WA_TranslucentBackground)
        self.setWindowModality(Qt.ApplicationModal)
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint|Qt.WindowStaysOnTopHint))
```

```
        # Initialise main layout
        self.BarcodeSearchVBL1 = QVBoxLayout()
        self.setLayout(self.BarcodeSearchVBL1)

        # This label displays each frame captured by the webcam
        self.FeedLabel = QLabel()
        # Set the image displayed by the window to a generic placeholder image whilst
          the webcam feed is being set up
        self.FeedLabel.setPixmap(QPixmap("Resources/WebCamConnecting.png"))
        self.BarcodeSearchVBL1.addWidget(self.FeedLabel)

        # This label will be updated to relay messages to the user
        # Currently, it is telling the user how to scan a barcode using the webcam
        self.InstructionLabel = QLabel("Position barcode in camera view")
        self.BarcodeSearchVBL1.addWidget(self.InstructionLabel)

        # The user presses this button when they want to close the window and stop the
          video feed
        self.CancelBTN = QPushButton("Cancel")
        # The button's clicked signal is connected to the CancelFeed method
        self.CancelBTN.clicked.connect(self.CancelFeed)
        self.BarcodeSearchVBL1.addWidget(self.CancelBTN)

        # Initialise Thread ----------------------------------------------------
        self.Worker1 = VideoFeedWorker()
        # Connect the object's signals to slots/methods in this class
        self.Worker1.ImageUpdateSignal.connect(self.ImageUpdateSlot)
        self.Worker1.BarcodeDetectedSignal.connect(self.BarcodeDetectedSlot)
        # Start the thread
        self.Worker1.start()
        # ----------------------------------------------------------------------

        # Determine where window should be positioned based on the widget which is
          passed in as an argument
        x = widget.mapToGlobal(QPoint(0, 0)).x() - 600
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        # The window should appear on the screen just below the button which is
          clicked to create it
        self.move(x, y)

        self.show()
```



Initially, I did not include the placeholder image in `FeedLabel`, instead a blank space would be displayed while the webcam connection was being set up. However, from a usability perspective, adding the placeholder image helps inform the user as to why there is a delay in initialising the webcam video feed. With the window object and its widgets/layouts initialised, I moved on to programming the thread which captures each frame from the webcam and returns it to the parent class to be displayed in `FeedLabel`. Every thread in PyQt inherits from the QThread class and they all must have the standard methods `run()` and `stop()`. `VideoFeedWorker` works using a while loop. For each iteration of the loop, a frame is captured, flipped, scaled ,and converted into a format compatible with PyQt. That frame is then emitted as a parameter of `ImageUpdateSignal`. The thread checks each frame for a barcode using the pyzbar module. If a barcode is detected in the frame, a red rectangle is drawn around it and the `BarcodeDetectedSignal` is emitted with the barcode data passed in as a parameter.

```
# This object is the thread which handles displaying the openCV video feed on the user's
  screen
# It inherits from QThread
class VideoFeedWorker(QThread):
    # Signal for when a new frame has been captured
    # it takes the captured frame image as a parameter
    ImageUpdateSignal = pyqtSignal(QImage)
    # Signal for when a barcode is detected in the frame
    # It takes the upc of the barcode as a parameter
    BarcodeDetectedSignal = pyqtSignal(str)
    # Signal for when there is an error capturing frames
    VideoCaptureErrorSignal = pyqtSignal()
    # Method executed when start() is called on the object
    # Initialises the thread loop
    def run(self):
        # Counter keeps track of how many frames have been recorded
        Counter = 0
        # Boolean value used to signal whether the while loop capturing each frame should
          continue executing or not
        # This value is set to False when the user presses the cancel button
        self.ThreadActive = True
        # Initialise connection with webcam, the 0 parameter tells the program to select
          the first webcam connection if the computer has multiple camera inputs
        Capture = cv2.VideoCapture(0)
        # This while loop captures each frame, processes it, scans it for barcodes and
          sends it back to the main program loop to be displayed in FeedLabel
        # It keeps iterating until boolean is set to False
        # Because it is executing in a concurrent thread, the while loop won't freeze the
          main program loop
        while self.ThreadActive:
            # Caputre frame
            Ret, Frame = Capture.read()
            # If the frame is successfully captured...
            if Ret:
                # Convert frame to object
                Image = cv2.cvtColor(Frame, cv2.COLOR_BGR2RGB)
                # Flip image (OpenCV doesn't automatically mirror images)
                FlippedImage = cv2.flip(Image, 1)
                # The program scans for barcodes every 5 frames
                if Counter % 5 == 0:
                    # Use pyzbar to scan frame for barcodes
                    Barcodes = pyzbar.decode(Frame)
                    # If a barcode is present
                    if Barcodes:
                        # Get the dimensions of the barcode
                        x, y, w, h = Barcodes[0].rect
                        # Draw a red rectangle around the barcode
                        RectImage = cv2.rectangle(FlippedImage, (x, y), (x + w, y + h),
```

```
                        (255, 0, 0), 5)
                        # Convert the frame to a format that is compatible with PyQt
                        ConvertToQtFormat = QImage(RectImage.data, RectImage.shape[1],
                        RectImage.shape[0], QImage.Format_RGB888)
                        # Scale image
                        FinalImage = ConvertToQtFormat.scaled(640, 480, Qt.KeepAspectRatio)
                        # Emit signal with the captured frame passed in as a parameter
                        # When the parent object receives its signal, it will set the image
                          displayed in FeedLabel to the frame emitted with the signal
                        self.ImageUpdateSignal.emit(FinalImage)
                        # Emit signal with the upc of the scanned barcode passed in as a
                          parameter
                        self.BarcodeDetectedSignal.emit(Barcodes[0].data.decode("utf-8"))
                        # Stop the video feed thread
                        self.stop()
                    # If a barcode isn't detected in the frame...
                    else:
                        # Convert frame to PyQt compatible format
                        ConvertToQtFormat = QImage(FlippedImage.data,FlippedImage.shape[1],
                        FlippedImage.shape[0], QImage.Format_RGB888)
                        # Scale image
                        FinalImage = ConvertToQtFormat.scaled(640, 480, Qt.KeepAspectRatio)
                        # Emit signal
                        self.ImageUpdateSignal.emit(FinalImage)
                # This is the code executed when the number of frames isn't a multiple of 5
                else:
                    # Convert frame to PyQt compatible format
                    ConvertToQtFormat = QImage(FlippedImage.data, FlippedImage.shape[1],
                    FlippedImage.shape[0], QImage.Format_RGB888)
                    # Scale image
                    FinalImage = ConvertToQtFormat.scaled(640, 480, Qt.KeepAspectRatio)
                    # Emit signal
                    self.ImageUpdateSignal.emit(FinalImage)
                # Increment counter
                Counter += 1
            # if there is an error capturing the frame, emit the error signal
            else:
                self.VideoCaptureErrorSignal.emit()
```

The stop method of the thread stops video from being captured and kills the thread. It is executed when a barcode is detected and when the cancel button is pressed. This method is required because, without it, the program would continue recording the video even after the window was closed, which affects the software's performance and raises privacy concerns.

```
# This method stops the thread from executing
def stop(self):
    self.ThreadActive = False
    self.quit()
```

`ImageUpdateSignal` is connected to the slot `ImageUpdateSlot` in the `BarcodeSearch` class. This slot updates `FeedLabel` to display the new frame.

```
# This method is executed when VideoFeedWorker emits ImageUpdateSignal
# It sets the image displayed in FeedLabel to the image emitted by the thread
def ImageUpdateSlot(self, Frame):
    self.FeedLabel.setPixmap(QPixmap.fromImage(Frame))
```

The program now outputs a live video feed from the user's webcam:

When the user positions a barcode in the frame, it is highlighted with a red barcode, the video feed is stopped and `BarcodeDetectedSignal` is emitted:

`BarcodeDetectedSignal` is connected to `BarcodeDetectedSlot`. This changes the text of `InstructionLabel` to inform the user that the barcode is being searched for and initialises the `WebSearchWorker` thread.

```
# This method is executed when VideoFeedWorker emits BarcodeDetectedSignal
# It initialises the second QThread object which performs the online barcode search
def BarcodeDetectedSlot(self, Barcode):
    # Change the text of InstructionLabel to inform the user that the barcode is being
      searched for
    self.InstructionLabel.setText("Searching Barcode...")
    # Initialise new thread, passing in the detected barcode as an argument
    self.Worker2 = WebSearchWorker(Barcode = Barcode)
    # Connect the object's signals to slots/methods in the parent class
    self.Worker2.SearchCompleteSignal.connect(self.SearchCompleteSlot)
    self.Worker2.SearchFailedSignal.connect(self.SearchFailedSlot)
```

```
    # Start the thread
    self.Worker2.start()
```

If there is an error connecting with the webcam and establishing the video feed, an error message is displayed:

```
# This method is executed when there is an error establishing a connection with the webcam
  (e.g. the user has no webcam or the webcam is broken)
def VideoCaptureErrorSlot(self):
    # Display error message
    self.InstructionLabel.setText("There was an issue capturing video”)
    self.FeedLabel.setPixmap(QPixmap(“Resources/WebcamError.png”))
    self.InstructionLabel.setStyleSheet("color: red")
```

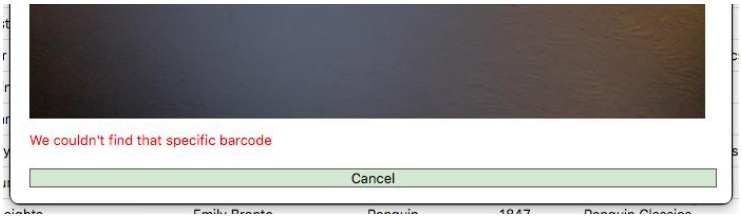
`WebSearchWorker` is another concurrent thread. It is a webscraping function which searches two online UPC databases for the barcode detected in the frame (which is passed in as an argument of the object). It uses the HTTP library requests to make connections with webpages and the HTML parsing library BeautifulSoup to parse webpages and find the necessary data. The first website the program queries, upcscavenger.com, stores the item name under the HTML class “us1881050501” and the second website, upcitemdb.com uses the class “num” in a li tag. The HTML parser can be directed towards the item name using these classes. If a result is found, `SearchCompleteSignal` is emitted, passing the search result in as a parameter. If no result can be found, `SearchFailedSignal` is emitted.

```
# This thread handles the webscraping barcode search
# Searches for the barcode in two different web databases: upcscavenger.com and
  upcitemdb.com
# It also inherits from QThread
class WebSearchWorker(QThread):
    # Signal for when the search has been successfully completed
    # It takes the string result of the search as a parameter
    SearchCompleteSignal = pyqtSignal(str)
    # Signal for when the search fails to find a result
    SearchFailedSignal = pyqtSignal()

    # Start thread
    def run(self):
        # requests.get makes a connection with the website
        # Format the barcode into the url to search for it in their database
        Response = requests.get("http://www.upcscavenger.com/barcode/" + self.Barcode +
        "/#page=barcode")
        # Create a parser object, passing in the HTML file from the requests.get()
        Soup = BeautifulSoup(Response.text, "html.parser")
        # Parse the file for the specific class which contains the result text, then strip
          any line breaks from the result
        Result = Soup.find(class_ = "us2329735521 us2445479844").find(class_ =
        "us1881050501").get_text().replace("\n", "")
        # If the search on the first website does not return a result, try searching the
          second website
        if Result == self.Barcode:
            try:
                # Connect with second website
                Response2 = requests.get("https://www.upcitemdb.com/upc/" + self.Barcode)
                # Create second parser object
                Soup2 = BeautifulSoup(Response2.text, "html.parser")
                # Locate HTML class
                Result2 = Soup2.find(class_ = "num").find("li").get_text()
                # Emit signal
                self.SearchCompleteSignal.emit(Result2)
            # If the second search also doesn't return a result, emit SearchFailedSignal
            except:
                self.SearchFailedSignal.emit()
        # If the first search is successful, emit SearchCompleteSignal, passing in the
```

```
            search result as a parameter
        else:
            self.SearchCompleteSignal.emit(Result)
```

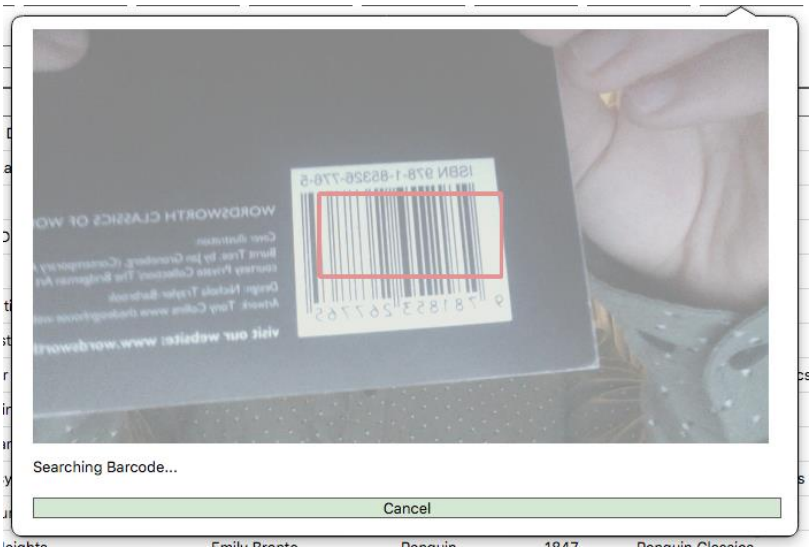
SearchFailedSignal is connected to a slot which displays an error message in InstructionLabel for 6 seconds. It then restarts the video feed, allowing the user to scan a new barcode.

```
# This method is executed when WebSearchWorker emits SearchFailedSignal
def SearchFailedSlot(self):
    # Sets the text in InstructionLabel to an error message
    self.InstructionLabel.setText("We couldn't find that specific barcode")
    self.InstructionLabel.setStyleSheet("color: red")
    QTimer.singleShot(6000, self.RevertInstructionLabel)
    # Restart video feed
    self.FeedLabel.setPixmap(QPixmap("Resources/WebCamConnecting2.png"))
    self.Worker1.start()
```



SearchCompleteSignal is connected to SearchCompleteSlot. This method changes the saturation of the image displayed in FeedLabel to grey it out. It then creates an instance of SearchResultPopup to output the result of the barcode search to the user.

```
# This method is executed when WebSearchWorker emits SearchCompleteSignal
# It relays the results of the barcode search to the user
def SearchCompleteSlot(self, Result):
    # Create image effect object and apply to FeedLabel
    Effect = QGraphicsColorizeEffect(self.FeedLabel)
    # This effect will change the saturation of the image displayed in the label
    Effect.setStrength(0.7)
    Effect.setColor(QColor("silver"))
    self.FeedLabel.setGraphicsEffect(Effect)
    # Create an instance of the popup, passing the results of the barcode search and
      FeedLabel object in as arguments
    self.SearchResultPopupInstance = SearchResultPopup(Result = Result, FeedLabel =
    self.FeedLabel)
    # Connect signals to slots
    self.SearchResultPopupInstance.CopiedSignal.connect(self.CopiedSlot)
    self.SearchResultPopupInstance.NewSearchSignal.connect(self.NewSearchSlot)
```



`SearchResultPopup` is a frameless window. It consists of a label displaying the search result, a button for copying the search result to the clipboard and a button for scanning a new barcode. During development, I found that some search results were too long to be displayed fully in the window, so I placed the label in a scroll area so that long search results will be scrollable. The position of this window is calculated so that it displays in the middle of `FeedLabel` on top of the greyed out image.

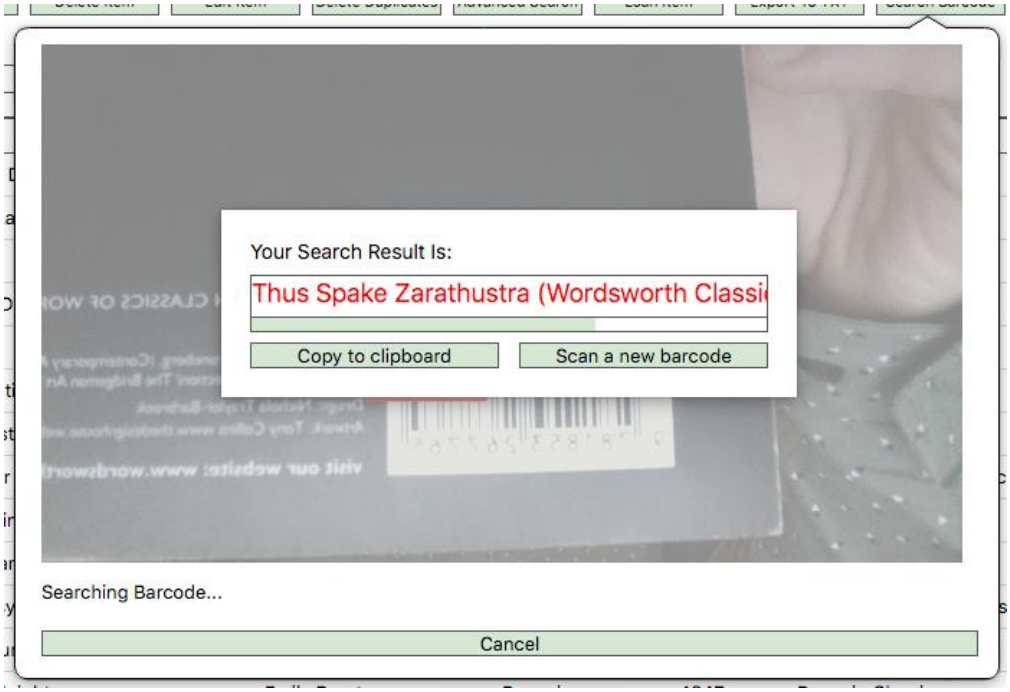
```
# This object is the window which relays the results of the barcode search to the user
# It takes the result of the search and the FeedLabel object as arguments
class SearchResultPopup(QWidget):
    # Signal for when the search result has been copied to the clipboard
    CopiedSignal = pyqtSignal()
    # Signal for when the user wants to scan a new barcode
    NewSearchSignal = pyqtSignal()
    def __init__(self, Result, FeedLabel):
        super(SearchResultPopup, self).__init__()

        # Assign arguments as attributes
        self.Result = Result
        self.FeedLabel = FeedLabel

        # Create clipboard object
        self.Clipboard = QApplication.clipboard()
        self.Clipboard.clear(mode=self.Clipboard.Clipboard)

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window attributes
        self.setWindowModality(Qt.ApplicationModal)
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint|Qt.WindowStaysOnTopHint))

        # Initialise main layout
        self.VBL = QVBoxLayout()
        self.setLayout(self.VBL)

        InfoLabel = QLabel("Your Search Result Is:")
        InfoLabel.setStyleSheet("font-size: 14px;")
        self.VBL.addWidget(InfoLabel)

        # Label for displaying the result of the search
        ResultLabel = QLabel(self.Result)
        ResultLabel.setStyleSheet("font-size: 18px; color: red; padding: 5px;")
        # The label will be displayed in a scroll area in case the label is longer than the
          width of the window
        self.ResultScrollArea = QScrollArea()
        self.ResultScrollArea.setWidget(ResultLabel)
        self.ResultScrollArea.setStyleSheet("QScrollBar:horizontal { border-bottom: 0px;
        border-left: 0px; border-right: 0px; } QScrollBar::handle { border-left: 0px;
        border-right: 0px; }")
        self.ResultScrollArea.setFixedHeight(50)
        self.VBL.addWidget(self.ResultScrollArea)

        self.ActionBTNHBL = QHBoxLayout()
        # Button for copying the search result to the clipboard
        self.CopyClickboardBTN = QPushButton("Copy to clipboard")
        self.CopyClickboardBTN.clicked.connect(self.CopyToClipboard)
        self.ActionBTNHBL.addWidget(self.CopyClickboardBTN)
        # Button for starting a new barcode search
        self.ScanAgainBTN = QPushButton("Scan a new barcode")
        self.ScanAgainBTN.clicked.connect(self.StartNewSearch)
        self.ActionBTNHBL.addWidget(self.ScanAgainBTN)
        self.VBL.addLayout(self.ActionBTNHBL)
```

```
        self.setFixedSize(400, 140)

        # Position widget in the middle of FeedLabel
        x = self.FeedLabel.mapToGlobal(QPoint(0, 0)).x() + (self.FeedLabel.width() / 2) -
        (self.width() / 2)
        y = self.FeedLabel.mapToGlobal(QPoint(0, 0)).y() + (self.FeedLabel.height() / 2) -
        (self.height() / 2)
        self.move(x, y)

        self.show()
```

```
# When the user wants to start a new barcode search, the appropriate signal is emitted and
the popup window is closed
def StartNewSearch(self):
    self.NewSearchSignal.emit()
    self.close()
```

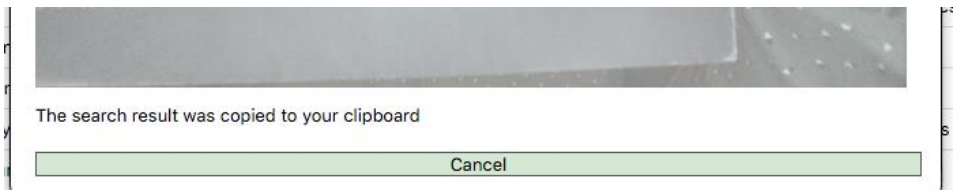
```
# When the user wants to copy the result, it is copied to their clipboard and the
appropriate signal is emitted
def CopyToClipboard(self):
    self.Clipboard.setText(self.Result)
    self.CopiedSignal.emit()
```

Below is the output when a barcode search has been successfully performed:



When `CopyClipboardBTN` is pressed, the slot `CopiedSlot` displays a message in `InstructionLabel`:

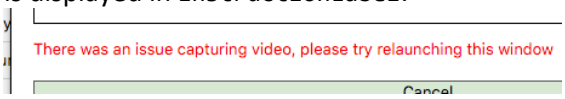
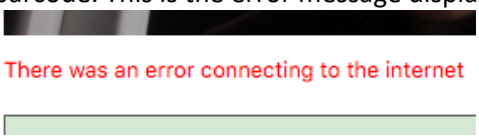
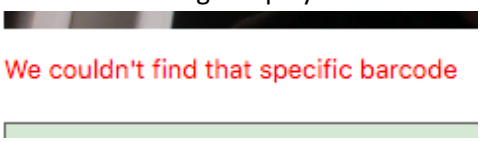
```
# This method is executed when SearchResultPopupInstance emits CopiedSignal
# It changes the text in InstructionLabel to inform the user that they have successfully
  copied the search results to their clipboard
def CopiedSlot(self):
    self.InstructionLabel.setText("The search result was copied to your clipboard")
    # Message should display for 6 seconds before feedLabel returns to its usual state
    QTimer.singleShot(6000, self.RevertInstructionLabel)
```



When `ScanAgainBTN` is pressed, the slot `NewSearchSlot` restarts the video feed by restarting the thread:

```
# This method is executed when SearchResultPopupInstance emits NewSearchSignal
# It restarts the Worker1 thread, allowing the user to scan a new barcode
def NewSearchSlot(self):
    self.FeedLabel.setGraphicsEffect(None)
    # Set FeedLabel's image to a generic placeholder image whilst the webcam feed is being
      restarted
    self.FeedLabel.setPixmap(QPixmap("Resources/WebCamConnecting2.png"))
    self.Worker1.start()
```

Testing

Test	Expected Result	Actual Result
Issues capturing video from webcam	`VideoFeedWorker` should emit the appropriate signal when it can't capture a frame, leading to an error message being displayed in the window	When I unplug my webcam, the following message is displayed in `InstructionLabel`:  And this image is displayed in `FeedLabel`:  The webcam connection has been severed
Internet connection interrupted during barcode search	An error message should be displayed in `InstructionLabel`	I performed this test by disconnecting the WIFI on my computer whilst the program was searching a barcode. This is the error message displayed:  The user can then press the cancel button and reopen the window to try again
Failed barcode search	When the online barcode search returns no results, a suitable message should be displayed in `InstructionLabel` and the video feed should be retarted so a new barcode can be scanned	This is the message displayed in `InstructionLabel`:  The video feed is restarted by calling `Worker1.start()` which reinitialises the video feed thread
Testing buttons in `SearchResultPopup`	`CopyClipboardBTN` should copy the search result to the user's clipboard and `ScanAgainBTN` should restart `Worker1`, and scan another barcode	Both buttons work as expected

I also rigorously tested the barcode search by scanning a broad range of different items that people may collect (DVDs, CDs, records, books, etc.) and most of these searches successfully returned results. This iteration also passed integration testing as the window is successfully instantiated in `OpenCollectionWindow` when `BarcodeSearchBTN` is pressed.

Review

During this iteration I successfully coded a module which uses OpenCV and pyzbar to scan barcodes through a webcam. The program then uses webscraping modules to search for the barcode's corresponding item online and relays the results to the user in a popup window. This iteration made use of concurrent processing to allow multiple threads to run concurrently alongside the main program loop; one for processing the webcam input and one for performing the online search. My designs for this window in the design stage weren't very clear about how I would go about outputting the results of the barcode search to the user, so I had to devise a method during development. Testing during this iteration involved scanning a multitude of different items to ensure that the barcode search worked and testing the failsafe features I implemented for if there was an issue with the webcam or the Internet connection. The next iteration will be coding the loan management system.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- The ability to search for and add items using their ISBN codes (barcodes) (24)

Iteration 14; Loan Management System

This iteration involves programming a system by which users can loan out items from their collections and receive notifications when the item is due in. There are two aspects to this iteration: the front-end process of presenting the user with an intuitive way of loaning out items and the backend operations which involve writing loaned items to the SQL backend, periodically checking this backend for outstanding loaned items and notifying the user when an item is due.

I began this iteration by creating a SQL table called Loans which stores a loaned item's due date, its foreign keys and Boolean values for whether they would like to receive email notifications and push notifications or not when the item is due.

```
CREATE TABLE Loans
  (
     PK_Loans                INTEGER(255) NOT NULL AUTO_INCREMENT PRIMARY KEY,
     DueDate                 DATE,
     FK_Collections_Loans    INTEGER(255),
     FK_Users_Loans          INTEGER(255),
     KeyInCorrespondingTable INTEGER(255),
     Email                   BOOLEAN,
     Push                    BOOLEAN
  )
```

Each entry into this table represents an item that has been loaned out. DueDate stores a datetime object of the item's due date. `FK_Collections_Loans` is a foreign key linking the entry to the primary key of the collection it belongs to in the Collections table. `FK_Users_Loans` links the loaned item to the user whose collection it belongs to. `KeyInCorrespondingTable` is the item's primary key in the table for the collection it belongs to. Email is set to TRUE if the user wants to be emailed when the item is due and FALSE when the user doesn't want to be emailed. The same if true for Push, but it is for receiving push notifications instead of emails.

For example, an item in the book collection has primary key 87. The primary key of the book collection in the Collections table is 55 and the collection belongs to the user with primary key 3 in the Users table. If I wanted to loan this item out until the 12th of January 2021 and only receive email notifications, not push notifications, this is what the entry would look like:

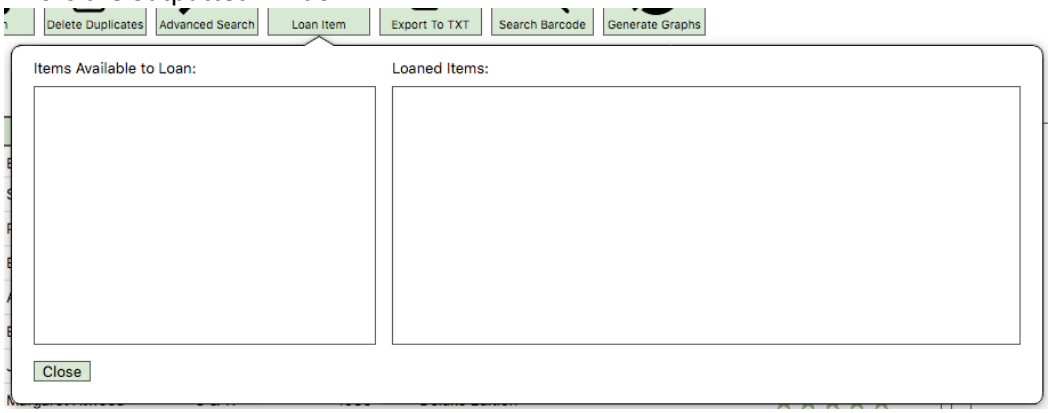
PK_Loans	DueDate	FK_Collections_Loans	FK_Users_Loans	KeyInCorrespondingTable	Email	Push
1	2021-01-12	55	3	87	TRUE	FALSE

With this backend job dealt with, the next step was to create the frontend window which displays items to the user and allows them to loan items out. I decided to split the window into two lists: one for items which have already been loaned out and another for items which haven't been. The window has the same constructor as the previous windows, the same arguments and the same `painEvent`. `InitUI` is the method called in the constructor which initialises the window's widgets and places them in layouts. As well as the two lists, the window will contain a `QDate` object for inputting the due date of loaned items, checkboxes for selecting whether emails and push notifications should be sent and a button for submitting an item to loan. These widgets will be hidden until the user selects an item to loan from `ToLoanList`.

```
# Initialise widgets
def InitUI(self):
    # The window is split into two columns; one for items that are available to loan and
      one for items which have already been loaned out
    self.ToLoanListLayout = QVBoxLayout()
    self.ToLoanListLayout.addWidget(QLabel("Items Available to Loan:"))
    # ToLoanList displays all the items in the collection which haven't been loaned out yet
    self.ToLoanList = QListWidget()
    # Style widget
    self.ToLoanList.setStyleSheet("selection-color: black;")
    self.ToLoanList.verticalScrollBar().setStyleSheet("QScrollBar { border-right: 0px;
    border-top: 0px; border-bottom: 0px; } QScrollBar::handle { border-top: 0px; border
    bottom: 0px; }")
    # Connect the signal emitted when the list's selection is changed to a slot
    self.ToLoanList.itemSelectionChanged.connect(self.ItemSelected)
    # Don't display focus rectangle when item is selected
    self.ToLoanList.setAttribute(Qt.WA_MacShowFocusRect, 0)
    # Add list to layout
    self.ToLoanListLayout.addWidget(self.ToLoanList)
    self.LoanItemHBL1.addLayout(self.ToLoanListLayout)
    self.ToLoanList.setFixedWidth(300)

    self.AlreadyLoanedListLayout = QVBoxLayout()
    self.AlreadyLoanedListLayout.addWidget(QLabel("Loaned Items:"))
    # AlreadyLoanedList displays all the items that have already been loaned out
    self.AlreadyLoanedList = QListWidget()
    # Style widget
    self.AlreadyLoanedList.setStyleSheet("selection-color: black;")
    self.AlreadyLoanedList.verticalScrollBar().setStyleSheet("QScrollBar { border-right:
    0px; border-top: 0px; border-bottom: 0px; } QScrollBar::handle { border-top: 0px;
    border-bottom: 0px; }")
    # Don't show focus rectangle
    self.AlreadyLoanedList.setAttribute(Qt.WA_MacShowFocusRect, 0)
    # Add to layout
    self.AlreadyLoanedListLayout.addWidget(self.AlreadyLoanedList)
    self.LoanItemHBL1.addLayout(self.AlreadyLoanedListLayout)

    # Run PopulatLists method to fill both lists with items
    self.PopulateLists()

    # QDateEdit object used to input the due date of the selected item when loaning it out
    self.PickDueDate = QDateEdit()
    self.ToLoanListLayout.addWidget(self.PickDueDate)
    # The widget is hidden until the user selects an item in ToLoanList
```

```
        self.PickDueDate.hide()

        # OptionsWidget displays two checkboxes
        self.OptionsWidget = QWidget()
        self.OptionsWidget.setObjectName("BorderlessWidget")
        self.OptionsHBL = QHBoxLayout()
        self.OptionsHBL.setContentsMargins(0, 0, 0, 0)
        # This checkbox allows the user to choose if they would like to receive push
          notifications when an item is due
        self.PushNotificationCB = QCheckBox("Push Notifications")
        self.PushNotificationCB.setChecked(True)
        self.OptionsHBL.addWidget(self.PushNotificationCB)
        # This checkbox allows the user to choose if they would like to receive email
          notifications when an item is due
        self.EmailNotificationCB = QCheckBox("Email Notifications")
        self.EmailNotificationCB.setChecked(True)
        self.OptionsHBL.addWidget(self.EmailNotificationCB)
        self.OptionsWidget.setLayout(self.OptionsHBL)
        self.ToLoanListLayout.addWidget(self.OptionsWidget)
        # These checkboxes are hidden until the user selects an item in ToLoanList
        self.OptionsWidget.hide()

        # This button allows the user to loan out the selected item in ToLoanList with the due
          date specified in PickDueDate
        self.SubmitLoanBTN = QPushButton("Loan Item")
        # Bind button to function
        self.SubmitLoanBTN.clicked.connect(self.SubmitLoanMethod)
        self.ToLoanListLayout.addWidget(self.SubmitLoanBTN)
        # Hide button until an item is selected in ToLoanList
        self.SubmitLoanBTN.hide()

        # Button for closing the window
        self.CloseWindowBTN = QPushButton("Close")
        self.CloseWindowBTN.setFixedWidth(50)
        self.CloseWindowBTN.clicked.connect(self.close)
        self.LoanItemVBL1.addWidget(self.CloseWindowBTN)
```

This is the outputted window:

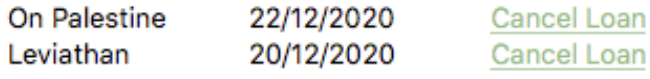


The method `PopulateLists` fills both lists with items. `ToLoanList` (left) will be filled with all the items in the collection which haven't been loaned out yet, whilst `AlreadyLoanedList` (right) will contain all the items which have been loaned out. Items in `ToLoanList` will be displayed using the standard `QListWidgetItem` object. However, `AlreadyLoanedList` will have to use a custom widget for items as I would like to include a button and two labels for each item which can't be done using the conventional widget. Below is the code for the `CustomListItem` object:

```
# This object is used to display items in AlreadyLoanedList
# Unlike standard list items, this widget can display multiple lables and buttons
class CustomListItem(QWidget):
    # This signal is emitted when CancelLoanBTN is pressed
```

```
    CancelLoanSignal = pyqtSignal(QObject)
    def __init__(self, Name, DueDate, ID):
        super(CustomListItem, self).__init__()

        # Reassign arguments as attribute
        # ID is the primary key of the item
        self.ID = ID
        # Name is the name of the item
        self.Name = Name
        # DueDate is the due date of the item
        self.DueDate = DueDate

        # Initialise horizontal layout
        self.RowHBL = QHBoxLayout()
        self.RowHBL.setContentsMargins(0, 0, 0, 1)
        self.setLayout(self.RowHBL)

        # If the length of the item's name is greater than 26 characters, shorten it and
          place ellipses at the end of the string
        if len(Name) >= 26:
            self.Name = self.Name[:26] + "..."

        # Create a label displaying the item's name and add it to the layout
        self.NameLBL = QLabel(self.Name)
        self.NameLBL.setFixedWidth(213)
        self.NameLBL.setContentsMargins(5, 0, 0, 0)
        self.RowHBL.addWidget(self.NameLBL)

        # Create a label displaying the item's due date and add it to the layout
        DateLBL = QLabel(self.DueDate)
        DateLBL.setFixedWidth(213)
        DateLBL.setContentsMargins(0, 0, 0, 0)
        self.RowHBL.addWidget(DateLBL)

        # CancelLoanBTN is used to delete loans
        self.CancelLoanBTN = QPushButton("Cancel Loan")
        self.CancelLoanBTN.setContentsMargins(0, 0, 0, 0)
        # Style button
        self.CancelLoanBTN.setStyleSheet("""
        QPushButton:pressed { color: black; }
        QPushButton:hover:!pressed { color: #D5E8D4; }
        QPushButton { border: 0px; background-color: rgba(0, 0, 0, 0); font-size: 13px;
        color: #96BA8A }
        """)
        ButtonFont = QFont()
        ButtonFont.setUnderline(True)
        self.CancelLoanBTN.setFont(ButtonFont)
        # Connect button's clicked signal so that it emits CancelLoanSignal when pressed
        self.CancelLoanBTN.clicked.connect(lambda: self.CancelLoanSignal.emit(self))
        self.RowHBL.addWidget(self.CancelLoanBTN)

        self.RowHBL.addStretch()
```

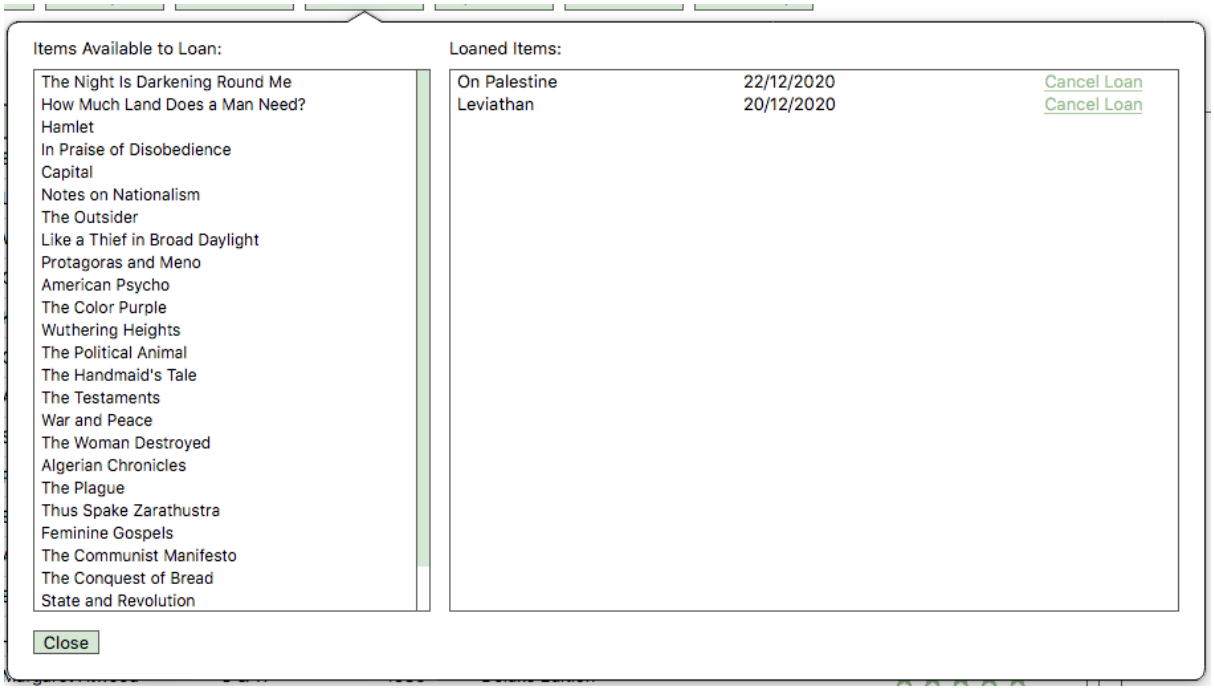
My initial approach to creating `CustomListItem` was to have it inherit from `QListWidgetItem` and then apply a few modifications to the child class. However, I found myself coming across the same limitations with this inherited class so I decided to have it inherit from `QWidget`. This means that each `CustomListItem` has to be placed within a container `QListWidgetItem` for it to display in a `QListWidget`. This is what instances of `CustomListItem` look like:



Now that the object which displays each item in `AlreadyLoanedList` is coded, I can progress onto programming `PopulateList`. Two forms of validation are required in this method: a presence check to see if there are any items on loan or if there are no items available to loan, and checks within the SQL backends to ensure that no item is loaned out twice. The function works by selecting all the items in the Loans table where `FK_Collections_Loans` corresponds with the primary key of the current collection in the Collections table. It also selects all the items in the current collection. It then compares the results of these two queries and uses them to form a list of items that are on loan and items that aren't. These lists are then used to populate their corresponding frontend list widgets.

```
# This method populates ToLoanList with items which haven't been loaned out yet and fills
  AlreadyLoanedList with the items in that collection which have already been loaned out
def PopulateLists(self):
    # Clear both lists of their contents before repopulating them
    self.ToLoanList.clear()
    self.AlreadyLoanedList.clear()
    # Get the primary key of every item in the current collection's table
    self.MyCursor.execute("SELECT PK_" + self.OpenCollectionTable + " FROM " +
    self.OpenCollectionTable)
    MyResult1 = self.MyCursor.fetchall()

    # Presence check: if there are no items in the collection, do not proceed
    if len(MyResult1) == 0:
        pass
    # If there are items in the collection, go on to populate the lists
    else:
        # Get every entry into Loans which corresponds with the primary key of the current
          collection
        self.MyCursor.execute("SELECT KeyInCorrespondingTable FROM Loans WHERE
        FK_Collections_Loans = " + str(self.OpenCollectionID))
        MyResult2 = self.MyCursor.fetchall()

        # Remove all items from MyResult1 which are also in MyResult2
        # This should leave us with all the loaned items in MyResult2 and all the unloaned
          items in MyResult1
        MyResult1 = list(set(MyResult1) - set(MyResult2))
        # Repopulate tuple as list
        MyResult1 = [item[0] for item in MyResult1]
        # Sort items into ascending order
        MyResult1.sort()
        # If all the items in the collection have already been loaned out, don't proceed
          with populating ToLoanList

        if len(MyResult1) == 0:
            pass
        else:
            # Construct a query to select all the items in the SQL table with primary keys
              in MyResult1
            if len(MyResult1) > 1:
                SQLStatement = "SELECT * FROM " + self.OpenCollectionTable + " WHERE "
                for x in range(0, len(MyResult1) - 1):
                    SQLStatement += "PK_" + self.OpenCollectionTable + " = " +
                    str(MyResult1[x]) + " OR "
                SQLStatement += "PK_" + self.OpenCollectionTable + " = " +
                str(MyResult1[len(MyResult1) - 1])
                self.MyCursor.execute(SQLStatement)

            else:
                self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " WHERE
                PK_" + self.OpenCollectionTable + " = " + str(MyResult1[0]))
            # Execute query to get a tuple of all the items which need to go into
              ToLoanList
            MyResult3 = self.MyCursor.fetchall()
```

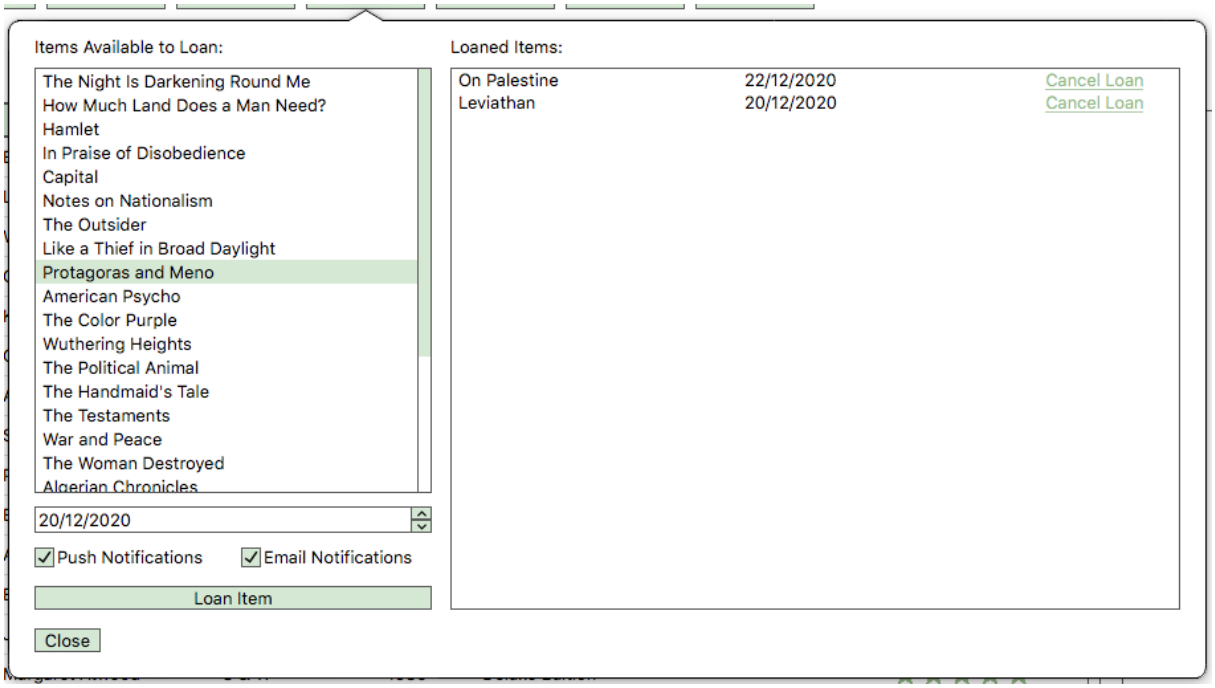
```
            # Populate ToLoanList with the results of the SQL query
            for item in MyResult3:
                ListItem = QListWidgetItem(str(item[1]))
                # setData attaches a hidden value to each list item which can be accessed
                  by calling data() on the item object
                # In this case, the program attaches the primary key of each item to its
                  corresponding list item
                ListItem.setData(Qt.UserRole, item[0])
                self.ToLoanList.addItem(ListItem)

        # Repopulate tuple as list
        MyResult2 = [item[0] for item in MyResult2]
        # If no items have been loaned out, don't proceed with populating AlreadyLoanedList
        if len(MyResult2) == 0:
            pass
        else:
            # Get all the entries into the Loans table which correspond with the current
              collection
            self.MyCursor.execute("SELECT * FROM Loans WHERE FK_Collections_Loans = " +
            str(self.OpenCollectionID))
            MyResult5 = self.MyCursor.fetchall()
            # Iterate through query result
            for x in range(0, len(MyResult5)):
                # For each item in the query result, get the corresponding entry into the
                  collection's table
                self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " WHERE
                PK_" + self.OpenCollectionTable + " = " + str(MyResult5[x][3]))
                MyResult6 = self.MyCursor.fetchall()
                # Create a list item to enclose the custom list widget
                ListItem = QListWidgetItem(self.AlreadyLoanedList)
                # Create an instance of CustomListItem, passing in the results of the SQL
                  query as parameters
                ListWidgetContents = CustomListItem(Name = MyResult6[0][1], DueDate =
                MyResult5[x][1].strftime("%d/%m/%Y"), ID = MyResult5[x][0])
                # Connect the CancelLoanSignal emitted by CustomListItem when CancelLoanBTN
                  is pressed to the corresponding slot
                ListWidgetContents.CancelLoanSignal.connect(self.CancelLoanSlot)
                ListItem.setSizeHint(ListWidgetContents.minimumSizeHint())
                # Set the layout of the container ListItem to the instance of
                  CustomListItem
                self.AlreadyLoanedList.setItemWidget(ListItem, ListWidgetContents)
```



As you can see above, the items in the collection are split into two lists based on whether they have been loaned out or not. Each item in `AlreadyLoanedList` has its name displayed along with its due date and a button for cancelling the loan. When it is clicked, `ToLoanList` emits the signal `ItemSelectionChanged`. This signal is connected to the method `ItemSelected`:

```
# This slot is executed when an item is selected in ToLoanList
def ItemSelected(self):
    # If no items are selected, hide PickDueDate, OptionsWidget and SubmitLoanBTN
    if len(self.ToLoanList.selectedItems()) == 0:
        self.PickDueDate.hide()
        self.SubmitLoanBTN.hide()
        self.OptionsWidget.hide()
    # If an item in the list has been selected, show these widgets
    else:
        self.PickDueDate.show()
        self.SubmitLoanBTN.show()
        self.OptionsWidget.show()
```

Now, when no items are selected in `ToLoanList`, the widgets will be hidden, but when an item is selected, the date picker, submit buttons and checkboxes will be shown:



`SubmitLoanMethod` is the function executed when `SubmitLoanBTN` is pressed. It collects the user inputs from the date picker and the checkboxes along with the primary key of the selected item in `ToLoanList` and inserts them into the Loans table in the backend. `PopulateLists` is then called to refresh the two list widgets and display the newly loaned item in `AlreadyLoanedList`.

```
# This method is executed when the user presses SubmitLoanBTN
# It writes the new loan to the Loans table
def SubmitLoanMethod(self):
    # Get the primary key corresponding with the item selected in ToLoanList
    SelectedItemID = self.ToLoanList.selectedItems()[0].data(Qt.UserRole)
    # Get the due date the user has inputted
    DueDate = self.PickDueDate.date().toPyDate()
    # Insert new entry into Loans table by formatting user inputs into SQL command
    self.MyCursor.execute("INSERT INTO Loans (DueDate, FK_Collections_Loans,
    KeyInCorrespondingTable, Email, Push) VALUES ('" + str(DueDate) + "', " +
    str(self.OpenCollectionID) + ", " + str(SelectedItemID) + ", " +
    str(self.EmailNotificationCB.isChecked()) + ", " +
    str(self.PushNotificationCB.isChecked()) + ")")
```

```
    # Save changes to database
    self.MyDB.commit()
    # Repopulate lists
    self.PopulateLists()
```

Newly loaned items are now recorded in the database:

PK_Loans	DueDate	FK_Collections_Loans	KeyInCorrespondingTable	Email	Push
112	2020-12-22	55	90	1	1
143	2020-12-20	55	88	1	1

The final method of `LoanItemWindow` is `CancelLoanSlot`. This slot is connected to the `CancelLoanSignal` of each instance of `CustomListItem`. The signal is emitted whenever the user presses `CancelLoanBTN`. The signal takes the object emitting it as a parameter, which is then passed to `CancelLoanSlot` as the parameter Sender. The slot then deletes the corresponding entry in the Loans table and refreshes the lists.

```
# This slot is executed when CancelLoanSignal is emitted
# It takes the parameter Sender which is the instance of CustomListItem which emitted the
  signal
def CancelLoanSlot(self, Sender):
    # Delete the corresponding item from the Loans SQL table
    self.MyCursor.execute("DELETE FROM Loans WHERE PK_Loans = " + str(Sender.ID))
    # Save changes to database
    self.MyDB.commit()
    # Repopulate lists
    self.PopulateLists()
```

The user can now loan out items and cancel these loans, with an entry being made into the Loans table for each loaned item. The next phase of this iteration is actually notifying the user when an item is due. The issue I faced was that the user will still want to be notified when an item is due even if the application is not running. This ruled out creating a thread that would wait until the item was due, as the thread would be killed when the application was closed. I also considered using an external server to handle sending notifications. However, the program database is hosted locally so setting up a remote connection would involve redesigning the whole database from scratch. Furthermore, a remote server would not be able to send push notifications to the device running the application. The solution I settled on was writing a short python script which would check the database for any outstanding items and then send notifications that those items were due. This script would be executed periodically (every 4 hours) using a unix-based job scheduler called crontab. This solution is specific to Linux and MacOS. I began by writing the python script, which is stored in a directory called Executables. The script uses the module smtplib for sending emails and uses the operating system's interface to display push notifications. The script begins by establishing a connection with the SQL database:

```
try:
    MyDB = mysql.connector.connect(
        host="localhost",
        user="root",
        password="[redacted]",
        database="Anthology"
    )
    MyCursor = MyDB.cursor()
    Connected = True
except:
    sys.exit(1)
```

It then records the current date and finds all the entries into the Loans table which are overdue

```
# Get current date
CurrentTime = datetime.date.today()
```

```
# List for storing all loans which are overdue
LoansRequiringAction = []
# Get all entries into Loans table
MyCursor.execute("SELECT * FROM Loans")
MyResult1 = MyCursor.fetchall()
# Iterate through query results
for item in MyResult1:
    # If an item's due date is older than or the same as the current date, add it to the
      list
    if item[1] <= CurrentTime:
        LoansRequiringAction.append(item)
```

The program then iterates through each of these entries. The Boolean values stored in the Email and Push columns indicate whether the user would like to receive push notifications and emails.

```
# Iterate through the list of overdue loans
for item in LoansRequiringAction:
    # Get conents of table that current item belongs to
    MyCursor.execute("SELECT * FROM Table" + str(item[2]) + " WHERE PK_Table" +
    str(item[2]) + " = " + str(item[3]))
    MyResult2 = MyCursor.fetchall()
    # Get entry into Collections table of collection which current item belongs to
    MyCursor.execute("SELECT * FROM Collections WHERE PK_Collections = " + str(item[2]))
    MyResult3 = MyCursor.fetchall()
    # Get the entry into the Users table for the user which the collection belongs to
    MyCursor.execute("SELECT * FROM Users WHERE PK_Users = " + str(MyResult3[0][3]))
    Recipient = MyCursor.fetchall()[0][1]
    # If the boolean value for Push notifications is set to true...
    if item[5]:
        # Display a notification, formatting the appropriate data in
        NotificationString = str(MyResult2[0][1]) + " is due today"
        os.system("""
        osascript -e 'display notification "{0}" with title "{1}"'
        """.format(NotificationString, "Anthology"))
```

The password for the development email is encoded in base64 in the source code. This is to ensure that people can't get the password by simply looking at the source code. However, this alone isn't a sufficient encryption method and so if I were to actually deploy the software I would have to find a better form of encryption, store the encrypted credential in a separate file and hide the source code. When the program needs to use the credential to get into the email account, it decodes the encoded binary object. Once it has logged in, the program can then send an email to the user, using the email address stored in the Users table which they use to log in.

```
    # If the boolean value for Email notifications if set to true...
    if item[4]:
        # The password for the dev email which sends emails to the user is encoded in
          base64 so people can't get the credentials for the account by looking at the
          source code
        EncodedCredential = b'c5B0EOxadGUjWDAd'
        # Decode base64
        # Begin by converting encoded byte object into utf-8
        ToString = EncodedCredential.decode("utf-8")
        # Then encode string into ascii
        Base64Bytes = ToString.encode('ascii')
        # Then decode string to get a byte object
        CredentialBytes = base64.b64decode(Base64Bytes)
        # Finally, decode binary object into ascii
        Credential = CredentialBytes.decode('ascii')

        # This text will make up the body of the email
        EmailText = """Subject: Item Due
        The item '{0}' which you loaned out from your {1} collection is due back today
        From the Anthology team
```

```
        """.format(str(MyResult2[0][1]), str(MyResult3[0][1]))

        # Start email server instance, using port 465
        with smtplib.SMTP_SSL("smtp.gmail.com", port = 465, context =
                              ssl.create_default_context()) as Server:
            # Login to dev email account
            Server.login("[redacted]", Credential)
            # Send email
            Server.sendmail("[redacted]", Recipient, EmailText)
            # Quit server instance
            Server.quit()
```

The final job for the script is to delete the entry from the Loans table:

```
# Delete entry into Loans item once notification has been sent
MyCursor.execute("DELETE FROM Loans WHERE PK_Loans = " + str(item[0]))
# Save changes to database
MyDB.commit()
```

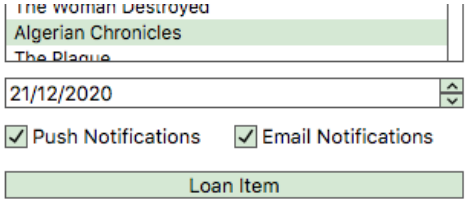
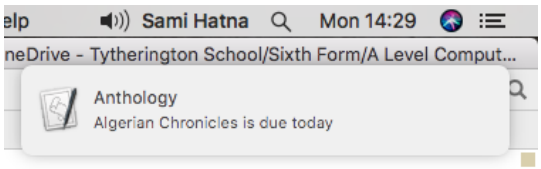
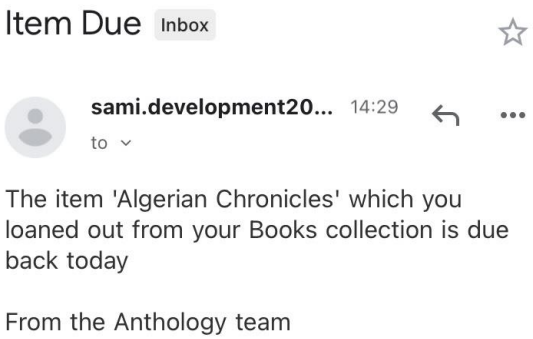
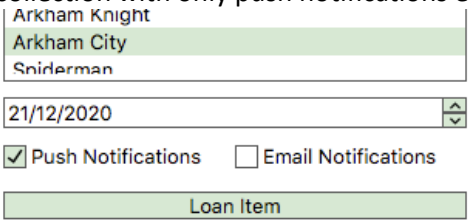
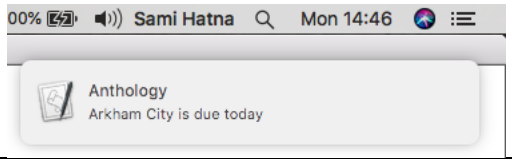
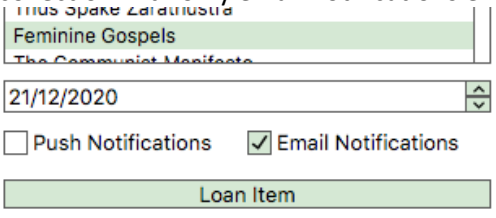
To get this script to execute every four hours, I opened the scheduler file in terminal's vi editor using the command `crontab -e`.
I then added the following command to the file:

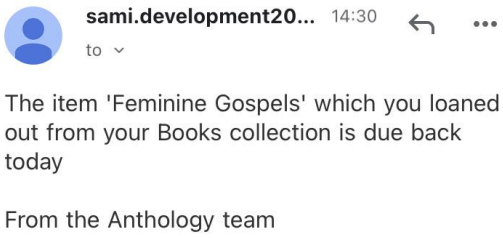
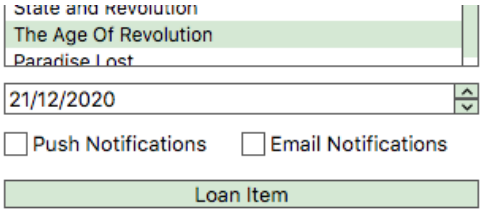
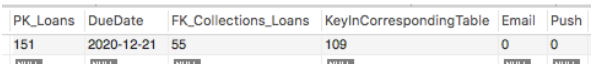
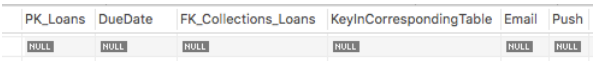
```
0 */4 * * * cd "/Users/rachkids/OneDrive - Tytherington School/Sixth Form/A Leve
l Computer Science/Anthology/Executables" && /usr/local/bin/python3 LoanChecker.
py
~
```

This command tells the system that every 4 hours it should go to the specified directory, it should then go to the directory where python is installed and use python3 to run the file LoanCheck.py

The script will now execute every four hours, checking for items which are due, notifying the user that they are due and then removing the loan from the database. It will be tested in the testing section.

Testing

Test	Expected Result	Actual Result
Loan item out with both email and push notifications enabled	On the day the item is due, the crontab script should display a push notification and send an email to the email address of the account the item belongs to	I loaned out Algerian Chronicles from my book collection until the 21st of December with both forms of notification enabled:  On the 21st of December, I received the following notification:  And an email: 
Loan item out with only push notifications enabled	The Boolean values stored under the Push and Email columns should instruct the program as to which types of notification to send	I loaned out Arkham City from my video game collection with only push notifications enabled:  This time I didn't receive an email, only a push notification 
Loan item out with only email notifications enabled	I should only receive an email notification when the item is due	I loaned out Feminine Gospels from my book collection with only email notifications enabled: 

		I received this email and no push notifications: 
Loan item out with both email and push notifications disabled	I should receive no notifications, but the entry into the Loans table should be deleted when the item is due and it should not display in the list for already loaned items after the due date	I loaned out The Age of Revolution from my book collection with both forms of notification disabled:  I didn't receive any notifications but the screenshots below show that the item was successfully deleted from the SQL table after the due date Before due date:  After due date:  As you can see, the loaned item's entry into the table has been deleted after the due date
Execute crontab script when SQL server is not running	The code forming the connection with the database is enclosed in a try/except clause so the script should terminate and try connecting again the next time it is executed	The test works as expected

For this iteration, the main concern with integration testing was ensuring that the crontab script could access the SQL backends and that any changes the script made (i.e., deleting entries into the Loans table) would be applied in the main application. The program passed integration testing because there are no instances where the main application and the crontab script would be accessing the same record in the Loans table. If there were any such instances I would have had to implement some form of record locking to maintain the integrity of the database.

Review

During this iteration, I coded a frontend system for loaning out items from a collection and a backend script to handle notifying users when items were due. This was accomplished using crontab to schedule the execution of Python scripts and using smtplib to send emails via python. Once again, I deviated massively from the UI mock-up I created during the design stage, although the logic of the

loan management system remained very similar. Testing during this iteration focussed on ensuring that notifications for overdue items were delivered at the right time and that the notification method matched the user's choice. The next iteration will involve exporting collections.

Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- A loan management system (22)
- Generate email notifications when a loaned item is due (23)

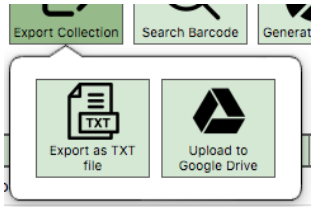
Iteration 15; Exporting Collections

The user should be able to export their collections in two different ways. The first will be an export to a TXT file stored in the directory of their choice. The second method will be uploading an export to their Google Drive account. I will make use of the Tabulate library to format the collection data in the TXT file and the pyDrive module will authenticate Google Drive accounts before uploading files. I began by creating a submenu implemented as a `QWidget`. This submenu will display when the user presses the export button in the toolbar. It will contain two buttons, one for each export method. The submenu object takes the same argument as `AddItemWindow` and it also has the same `paintEvent`.

```
# This object creates a submenu containing two new tool buttons
# One is for exporting the current collection as a TXT file, the other is for uploading the
  collection to Google Drive
class ExportSubMenu(QWidget):
    # This signal is emitted when the program is unable to export the collection
    Failed = pyqtSignal()
    # This signal is emitted when the program successfully exports the collection
    Success = pyqtSignal()
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID, OpenCollectionTable,
    widget = None):
        super(ExportSubMenu, self).__init__()

        # Reassign arguments as attributes
        self.ActiveUserID = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.OpenCollectionID = OpenCollectionID
        self.OpenCollectionTable = OpenCollectionTable

        # Set window attributes
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint | Qt.WindowStaysOnTopHint
        | Qt.Popup))
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setAttribute(Qt.WA_TranslucentBackground)

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Initialise main layout
        self.ExportHBL1 = QHBoxLayout()
        self.setLayout(self.ExportHBL1)

        # Create button for exporting as TXT file
        self.ExportToTxtBTN = QToolButton()
        self.ExportToTxtBTN.setText("Export as TXT\nfile")
        # Set button icon
        self.ExportToTxtBTN.setIcon(QIcon("Resources/ExportTXT.png"))
        self.ExportToTxtBTN.setIconSize(QSize(45, 45))
        self.ExportToTxtBTN.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)
```

```
        self.ExportToTxtBTN.setFixedWidth(90)
        self.ExportHBL1.addWidget(self.ExportToTxtBTN)
        # Bind button to function with the string parameter "Local" to indicate the user
          wants a local export rather than a Google Drive export
        self.ExportToTxtBTN.clicked.connect(lambda:self.ExportCollection(type = "Local"))

        # Create button for uploading collection to Google Drive
        self.ExportGDriveBTN = QToolButton()
        self.ExportGDriveBTN.setText("Upload to \nGoogle Drive")
        # Set button icon
        self.ExportGDriveBTN.setIcon(QIcon("Resources/GDriveIcon.png"))
        self.ExportGDriveBTN.setIconSize(QSize(45, 45))
        self.ExportGDriveBTN.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)
        self.ExportGDriveBTN.setFixedWidth(90)
        self.ExportHBL1.addWidget(self.ExportGDriveBTN)
        # Bind button to function with string parameter "GDrive" to indicate the user wants
          to upload the export to their Google Drive
        self.ExportGDriveBTN.clicked.connect(lambda:self.ExportCollection(type = "GDrive"))

        # Calculate window position
        x = widget.mapToGlobal(QPoint(0, 0)).x()
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        self.move(x, y)

        # Show window
        self.show()
```

This is what the menu looks like:



`ExportSubMenu` only has one method called `ExportCollection`. This method takes the parameter type, which indicates whether the user wants to export the collection locally or upload it to the cloud. So, when the user presses `ExportGDriveBTN`, the parameter type is set to “GDrive” whereas when the user presses `ExportToTxtBTN`, type is set to “Local”. The first job for `ExportCollection` to perform is retrieving data from the backend and presenting them in a TXT file:

```
# This method is called when either of the buttons are pressed
# The parameter type tells the program whether the user wants to create a local export or
  upload the export to their Google Drive
def ExportToTXT(self, type):
    # Get the collection's column names
    self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
    MyResult1 = self.MyCursor.fetchall()
    MyResult1 = [item[0] for item in MyResult1]
    # Remove first and last elements from list to get rid of primary key and thumbnails
      columns
    MyResult1 = MyResult1[:-1]
    del MyResult1[0]

    # Construct a SQL statement for selecting the data under all columns in the table
      except for the primary key and thumbnails
    SQLStatement = "SELECT "
    for x in range(0, len(MyResult1) - 2):
        SQLStatement += MyResult1[x] + ", "
    SQLStatement += MyResult1[len(MyResult1) - 1] + " FROM " + self.OpenCollectionTable
    # Execute SQL command
    self.MyCursor.execute(SQLStatement)
```

```
    MyResult2 = self.MyCursor.fetchall()
    # Use tabulate library to format query results as an ascii table
    ExportTable = tabulate(MyResult2, headers = MyResult1, tablefmt = "fancy_grid")
```

ExportTable is now a string containing all the data for the collection, formatted as an ascii table. If the user has pressed ExportToTxtBTN and type has thus been set to "Local", the method will go on to open a file dialog allowing the user to select the directory they would like to save the file to:

```
# If the user wants to export locally (they pressed ExportToTxtBTN)...
if type == "Local":
    # Create a file dialog instance, allowing the user to select the director they would
      like to export to
    self.DirectoryName = QFileDialog.getExistingDirectory(None, "Select Export Directory",
    os.path.abspath(os.sep))
    # Try/Except clause provides error checking
    try:
        # Create a new file in the user-specified directory called "AnthologyExport.txt"
        ExportFile = open(self.DirectoryName + "/AnthologyExport.txt", "w")
        # Write the contents of ExportTable to the new file
        ExportFile.write(ExportTable)
        # Close the file to prevent corruption
        ExportFile.close()
        # Emit signal, this should be received by the parent class (OpenCollectionWindow)
          and should display a message in the status bar informing the user that the export
          was successful
        self.Success.emit()
    except:
        # If there is an issue during the export, emit the Failed signal which will also
          display a message in the StatusBar when received
        self.Failed.emit()
```

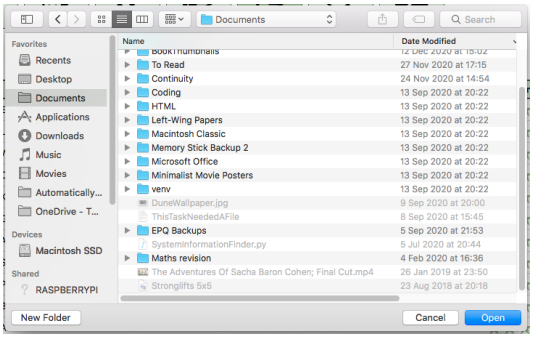
The `ExportSubMenu` object has two signals: `Success` and `Failed`. `Success` is emitted when the collection is exported without any issues and is connected to the following slot in the parent class:

```
self.ExportSubMenuInstance.Success.connect(lambda: self.StatusBar.showMessage("Your
collection was successfully exported!", 3000))
```

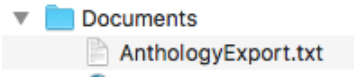
`Failed` is emitted when the program encounters an issue whilst exporting the collection and is connected to this slot which displays an error message in the status bar:

```
self.ExportSubMenuInstance.Failed.connect(lambda: self.StatusBar.showMessage("There was an
issue exporting your collection, please try again", 3000))
```

Now, when I try to export my video game collection to a TXT file, I am shown this file dialogue which is limited to only allow users to select directories:



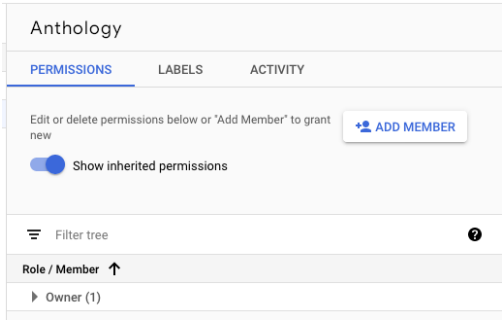
Then, if I select the Documents directory, a file entitled AnthologyExport.txt is written to that directory:



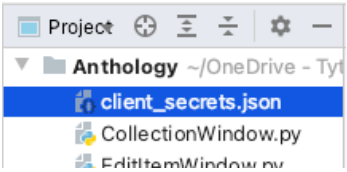
This is the contents of the file:

Title	Developer	Console	Rare
Skyrim	Bethesda	PS3	5
Red Dead Redemption 2	Rockstar	PS4	5
Red Dead Redemption	Rockstar	PS3	4
AC Origins	Ubisoft	PS4	3

With the local exports now working, I progressed onto uploading files to Google Drive. The first step was creating a project in Google Developer's Console:



After the project has been created, I needed to generate API keys to authenticate the project. These are stored in a JSON file called client_secrets stored in the working directory.

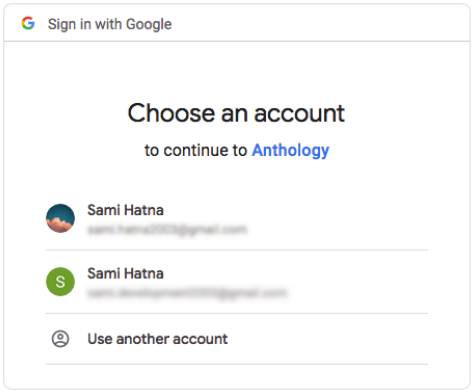


With this setup complete, I can now use pyDrive to login to a Google account and upload files to Google Drive. The program does this by saving a TXT export of the collection as a temporary file which will be deleted once the collection has been successfully uploaded. A GoogleAuth object is then created which opens a browser window on the Google login page, letting the user login to their account. A file object is then created, and its contents is set to the contents of the temporary file. The file can then be uploaded, and the `Success` signal will be emitted.

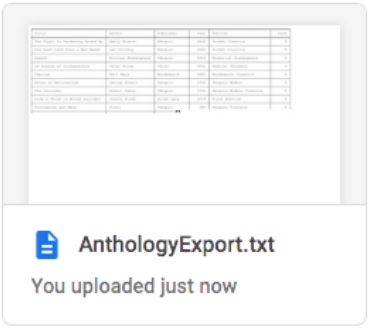
```
# If the user wants to upload the collection to their Google Drive (they pressed
  ExportGDriveBTN)…
else:
    # Try/Except clause once again provides error checking
    try:
        # Create an object to handle authentication
        GoogleLogin = GoogleAuth()
        # Opens browser and displays login screen
        GoogleLogin.LocalWebserverAuth()
        # Create Google Drive object to handle creating and uploading files
        Drive = GoogleDrive(GoogleLogin)

        # Create a file in the Temp directory
        ExportFile = open("Temp/AnthologyExport.txt", "w")
        # Write the contents of ExportTable to the new file
        ExportFile.write(ExportTable)
        ExportFile.close()

        # Open the file in the Temp directory
        with open("Temp/AnthologyExport.txt", "r") as file:
            # Create a new drive file object
```

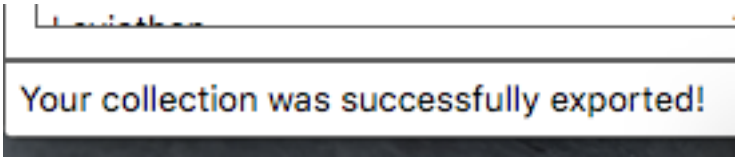
```
            FileForUpload = Drive.CreateFile({"title":os.path.basename(file.name)})
            # Set the contents of the drive file to the contents of the file in the Temp
              directory
            FileForUpload.SetContentString(ExportTable)
            # Upload the file to Google Drive
            FileForUpload.Upload()

        # delete the file in the Temp directory as it is no longer needed
        os.remove("Temp/AnthologyExport.txt")
        # Emit Success signal
        self.Success.emit()
    # If there is an issue whilst uploading the file to the cloud, execute the following
      code
    except:
        # Delete the file in the temp directory if it exists
        try:
            os.remove("Temp/AnthologyExport.txt")
        except:
            pass
        # Emit Failed signal
        self.Failed.emit()
```

Now, when the user presses `ExportGDriveBTN`, the following login screen is opened in the browser:

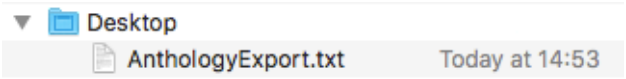


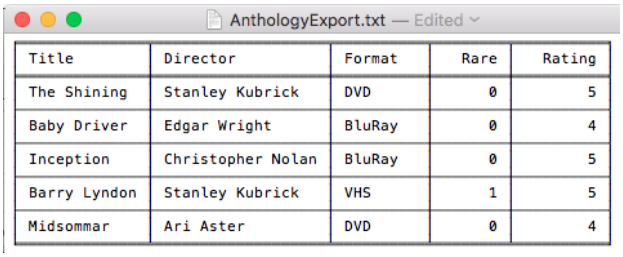
If I successfully login, a file will be uploaded to my Google Drive account containing the exported collection, and a message will be displayed in the status bar informing the user of the successful export:

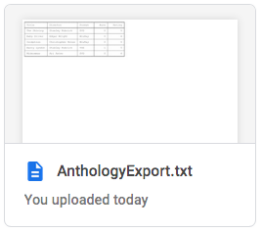


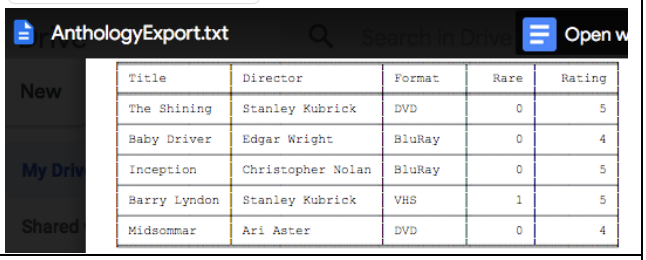


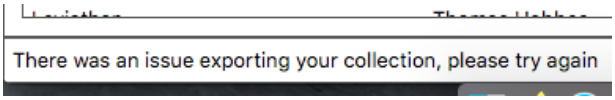
Testing

<table>
<tr><th>Test</th><th>Expected Result</th><th>Actual Result</th></tr>
<tr><td>Export collection to local directory as TXT file</td><td>The TXT file should be saved to the directory the user selects in the file dialogue</td><td>I exported my film collection to the desktop directory on my computer:
 

This is the contents of the exported file:
 

<table>
<tr><th>Title</th><th>Director</th><th>Format</th><th>Rare</th><th>Rating</th></tr>
<tr><td>The Shining</td><td>Stanley Kubrick</td><td>DVD</td><td>0</td><td>5</td></tr>
<tr><td>Baby Driver</td><td>Edgar Wright</td><td>BluRay</td><td>0</td><td>4</td></tr>
<tr><td>Inception</td><td>Christopher Nolan</td><td>BluRay</td><td>0</td><td>5</td></tr>
<tr><td>Barry Lyndon</td><td>Stanley Kubrick</td><td>VHS</td><td>1</td><td>5</td></tr>
<tr><td>Midsommar</td><td>Ari Aster</td><td>DVD</td><td>0</td><td>4</td></tr>
</table></td></tr>
<tr><td>Export collection to Google Drive</td><td>The user should be able to login to their Google account and then have the collection exported to their Google Drive</td><td>I tested this using the same film collection, this is the exported file in my Google Drive account:
 



<table>
<tr><th>Title</th><th>Director</th><th>Format</th><th>Rare</th><th>Rating</th></tr>
<tr><td>The Shining</td><td>Stanley Kubrick</td><td>DVD</td><td>0</td><td>5</td></tr>
<tr><td>Baby Driver</td><td>Edgar Wright</td><td>BluRay</td><td>0</td><td>4</td></tr>
<tr><td>Inception</td><td>Christopher Nolan</td><td>BluRay</td><td>0</td><td>5</td></tr>
<tr><td>Barry Lyndon</td><td>Stanley Kubrick</td><td>VHS</td><td>1</td><td>5</td></tr>
<tr><td>Midsommar</td><td>Ari Aster</td><td>DVD</td><td>0</td><td>4</td></tr>
</table></td></tr>
<tr><td>Issues whilst exporting a collection</td><td>The program has failsafe procedures coded in through the use of try/except clauses which should deal with any errors during both local and cloud exports</td><td>When the program can't access the directory during a local back up due to permissions errors, or if there is a problem during the Google authentication process, the Failed signal is emitted, which displays the following message in the main window's status bar:
 
</td></tr>
</table>

Review

During this iteration I programmed two methods by which the user can export their collection. Exporting the collection as a local TXT file was simple enough, however uploading files to Google Drive involved getting to grips with Google's API and how to securely handle client keys. I did not deviate massively from the pseudocode algorithms I created for this iteration during the Design stage. Testing during this iteration was a case of ensuring that collections could be successfully exported and that the failsafe measures would work when the program encountered an issue

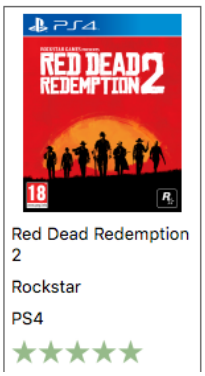
Aspects of the success criteria (page 22) I have fulfilled during this iteration:

- The option to export collections to a text file (20)
- The ability to back collections up to a Google Drive account (21)

Minor Changes

This section is for recording any small revisions or changes I made to my code during the development process.

I added the code below to the slot `DisplayImage` which displays the selected item's rating alongside its data. Previously, the application had only displayed the data.



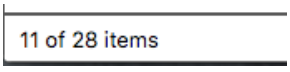

```
ThumbnailRatingPixmap = QPixmap("Resources/{0}
stars.png".format(str(MyResult3[len(MyResult3) - 2])))
ThumbnailRatingPixmap =
ThumbnailRatingPixmap.scaledToWidth(
                              100, Qt.SmoothTransformation
                              )
self.ThumbnailRatingLBL.setPixmap(ThumbnailRatingPixmap)
```

I also added this code to `DisplayImage` which updates the contents of the status bar when the user selects item(s) to display how many items they have selected:

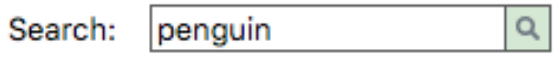
```
if len(self.CollectionTable.selectionModel().selectedRows()) > 1:
    self.StatusLBL.setText(
          str(len(self.CollectionTable.selectionModel().selectedRows())) + " of " +
          str(self.RowCount) + " items"
                               )
else:
    self.StatusLBL.setText(str(self.RowCount) + " items")
```

When no items are selected:

When items are selected:



As I began testing the program with larger collections, I realised that the simple search system devised during iteration 6 was not very efficient. The original system performed a search every time the contents of the search bar changed, by connecting the search bar's `textChanged` signal to the method performing the search. The issue with this system was that a new search was being performed each time the user inputted a new character into the search bar. This was less of an issue with smaller collections as querying the database doesn't take too long for them. However, with larger collections the simple search would lag and not work as efficiently as it should. I therefore changed the simple search by only performing the search when the user presses the search button or the enter key. This is what the new search bar looks like:



Search bar code:

```
# SearchBarHBL is the layout which stores the search bar and the sorting menu
self.SearchBarHBL = QHBoxLayout()
self.SearchBarHBL.addWidget(QLabel("Search:"))
self.SubHBL = QHBoxLayout()
self.SubHBL.setSpacing(0)
# SearchBarTB is the text box in which the user inputs search terms
self.SearchBarTB = QLineEdit()
self.SearchBarTB.setStyleSheet("border-right: 0px;")
self.SearchBarTB.setFixedWidth(150)
# When the user presses enter in a QLineEdit, it emits the editingFinished signal
```

```
# I have bound this signal to the function SimpleSearch so that whenever the user presses
enter, the function will be executed
self.SearchBarTB.editingFinished.connect(self.SimpleSearch)
self.SubHBL.addWidget(self.SearchBarTB)
self.SearchBTN = QPushButton()
self.SearchBTN.clicked.connect(self.SimpleSearch)
self.SearchBTN.setIcon(QIcon("Resources/SearchIcon.png"))
self.SearchBTN.setFixedSize(20, 20)
self.SubHBL.addWidget(self.SearchBTN)
self.SearchBarHBL.addLayout(self.SubHBL)
self.SearchBarHBL.addSpacing(50)
```

I added the following keyboard shortcuts for the application's key functionalities:

Shortcut	What it does
Ctrl + shift + A	Opens the window for adding items
Ctrl + shift + D	Deletes the selected item
Ctrl + shift + E	Opens the window for editing the selected item
Ctrl + shift + B	Opens the barcode search window
Ctrl + shift + G	Generates graphs based on the current collection
shift + up/down arrow	Changes the order of the collection
Ctrl + shift + H	Hides thumbnail and sorting panels
Ctrl + shift + S	Shows thumbnail and sorting panels

Below is an example of implementing the keyboard shortcut for deleting items in PyQt:

```
self.DeleteItemShortcut = QShortcut(QKeySequence("Ctrl+shift+D"), self)
self.DeleteItemShortcut.activated.connect(self.DeleteItemMethod)
```

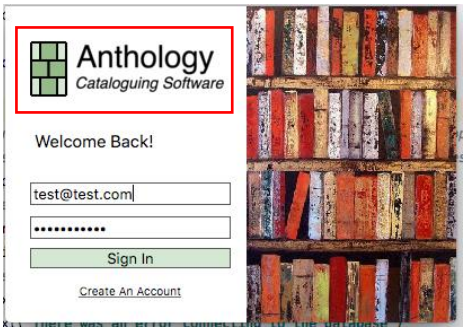
This fulfils criterion 26 in the success criteria: "Keyboard shortcuts assigned to different functions"

Section 4: Evaluation

Post-Development Testing

Before handing the program over to the end-users, I carried out a final round of testing to deal with any final bugs I could find. I started by testing for general aesthetic issues which may impact the program's usability. I found that on the login screen the application logo would display at the correct resolution on smaller laptop screens but would become blurry on larger monitors:

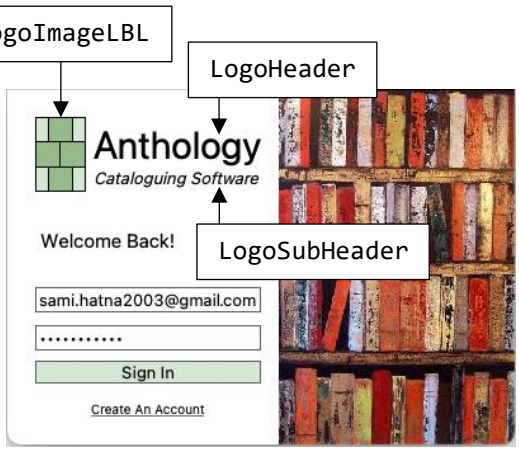
Login screen on larger monitors



Login screen on laptop display

As you can see above, the logo displays fine on smaller screens, however on larger screens the image quality suffers. I remedied this issue using the following code to reconstruct the logo using `QLabels` for the text rather than having them be a part of the image, so the quality of the text won't be affected by the display resolution:

```
# Create sublayout
InnerHBL = QHBoxLayout()
# Align elements within layout to the left
InnerHBL.setAlignment(Qt.AlignLeft)
InnerHBL.setSpacing(5)
# Create a label for displaying the logo image
LogoImageLBL = QLabel()
# Create pixmap for logo image
LogoPixmap = QPixmap("Resources/Logo3.png")
LogoImageLBL.setPixmap(LogoPixmap)
LogoImageLBL.setAlignment(Qt.AlignCenter)
InnerHBL.addWidget(LogoImageLBL)
# Create another sublayout for the text
InnerVBL = QVBoxLayout()
InnerVBL.setSpacing(0)
InnerVBL.setAlignment(Qt.AlignVCenter)
# Create heading label
LogoHeader = QLabel("Anthology")
# Style heading
LogoHeader.setStyleSheet("font-size: 30px;")
InnerVBL.addWidget(LogoHeader)
# Create subheading label
LogoSubHeader = QLabel("Cataloguing Software")
# Style subheading
LogoSubHeader.setStyleSheet("font: italic")
# Add widgets to layout
InnerVBL.addWidget(LogoSubHeader)
InnerHBL.addLayout(InnerVBL)
self.LoginVBL1.addLayout(InnerHBL)
```



The result is a logo which doesn't blur regardless of the resolution of the monitor it is being displayed on

There were no other glaring aesthetic issues, so I moved on to testing the core functionalities of the program.

Adding, Deleting and Editing Items

An issue I came across when testing these functions initially was the keyboard shortcuts I had used. On page 183, the keyboard shortcuts shown are all composed of command + shift + a letter. However, initially the commands were only command + a letter (e.g. command + A for adding items). The issue with this initial approach was that certain shortcuts would be overridden by system shortcuts. For example, if the user used the shortcut command + A whilst the item table was active, the program would select all the items in the table rather than displaying the add item window. I avoided this issue by adding the shift key into all the shortcuts so they wouldn't correspond with any system-wide shortcuts.

Another bug I had to deal with was that, when editing an item, the rarity would not change after I submitted the changes. Every other property would change but the rarity would remain as it was before. I revisited `EditItemWindow`'s code and added the following code which handles the bug by adding the status of the rare checkbox to the tuple of data used in the SQL command:

```
# If user has checked the rare check box, set boolean value of Rare column to TRUE
if self.RareCheckBox.isChecked():
    DataForInsertion += (True, )
# Otherwise, the boolean value will be FALSE
else:
    DataForInsertion += (False, )
```

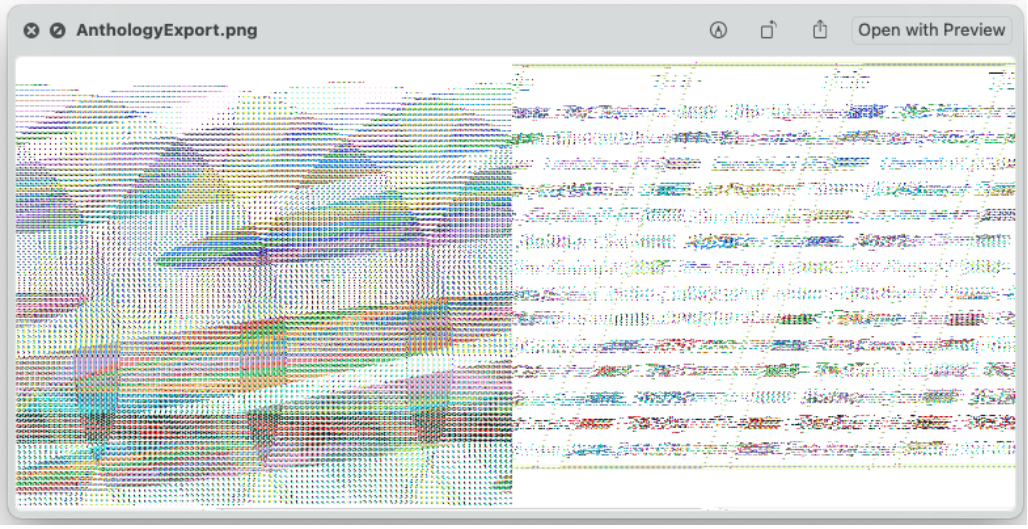
Adding, deleting and deleting duplicate items all worked well and I couldn't find any bugs.

Advanced Search

I couldn't find any bugs or issues with this aspect of the program as I had rigorously tested it during development.

Generating Graphs

The graphs were being generated and displayed properly within the app. However, after updating my machine to MacOS Big Sur, I was experiencing issues with exporting graphs as images. Graph exports would be distorted and unreadable as seen below:



I found that this distortion was the result of using the wrong image format within the arguments of the `QImage` constructor. Instead of using the format `ARGB8565_Premultiplied` I had to use the argument `RGBX888`.

Revised `QImage` constructor:

```
Export= QImage(Canvas.buffer_rgba(), GSize.width(), GSize.height(), QImage.Format_RGBX8888)
```

Loaning Items

When I try to display the loaning window whilst no items are out for loan, the program would crash and give me the following SQL error message:

```
  File "/Users/sami/OneDrive - Tytherington School/Sixth Form/A Level Computer Science/Anthology/Interpreter/lib/python3.8/site-packages/mysql/connector/connection_cext.py", line 508, in cmd_query
    raise errors.get_mysql_exception(exc.errno, msg=exc.msg,
mysql.connector.errors.ProgrammingError: 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '' at line 1

Process finished with exit code 134 (interrupted by signal 6: SIGABRT)
```

Because the error message I received in the Python console isn't very informative, I tried to emulate the crash in my SQL IDE. I found that the issue was a simple logic error solved by implementing the following presence check:

```
# Presence check: if there are no items out to loan, do not proceed
if len(MyResult1) == 0:
    pass
# If there are items out to loan, go on to populate the lists
else:
    ...
```

Another bug I found was that, when setting the due date for a loan, users could set due dates in the past. Although this is not a software-breaking bug, it means that items will be left in the SQL table indefinitely out on loan unless the user chooses to cancel the loan. This issue was dealt with by adding the following code to the `InitUI` method of the window which sets the minimum date the user can select in the `QDateEdit` widget to the current date:

```
self.PickDueDate.setMinimumDate(QDate.currentDate())
```

Exporting Collections

Making local exports works as expected, as does uploading exports to Google Drive. When packaging the project for end-user testing, I must conceal the client_secrets.json file for security reasons.

Barcode Searches

The main issue with this component of the program was that the video feed was very inconsistent and kept stuttering. Initially, I assumed that this issue may have been because Qt and OpenCV did not integrate together well. However, upon closer inspection I realised that the video feed was lagging because I had programmed the video thread to check every single frame for a barcode. This massively slowed down the thread's efficiency which left the video feed delayed and stuttering. To improve performance, I implemented a counter to ensure that the frame is only checked for barcodes every 5 frames. This made the live video playback much smoother, but the trade-off is that it can take longer for the program to pick up on a barcode in the frame. To test the effectiveness of this improvement, I recorded the length of time which elapses between a frame being captured and `ImageUpdateSignal` being emitted (using the method outlined on page 57). Before adding the counter, the time between capture and signal was 0.0197 seconds. With the counter added, the time taken on every fifth frame was 0.0101 seconds and 0.0056 seconds on every other frame. These results demonstrate that implementing the counter drastically improved the efficiency of video processing.

Opening Collections

When testing this aspect of the program, I came across a bug with the window's file menu. When creating a `QMenuBar` object, you need to specify which `QMainWindow` object the menu belongs to, otherwise it will apply the menu to all windows. I had forgotten to pass the parent window into the menu bar's constructor, leading to a lot of strange bugs which would log a user out of their account and into another one if they closed a collection. The fix for this bug was simply a case of passing self into the menu bar's constructor:

```
self.MainMenu = QMenuBar (self)
```

Another bug I encountered was when trying to open an action figure collection with the properties Name, Range, Price and Year. The program would produce the following error when I tried to open it:

```
  File "/Users/sami/OneDrive - Tytherington School/Sixth Form/A Level Computer Science/Anthology/Interpreter/lib/python3.8/site-packages/mysql/connector/connection_cext.py", lin
    raise errors.get_mysql_exception(exc.errno, msg=exc.msg,
mysql.connector.errors.ProgrammingError: 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax
to use near 'Range, COUNT(*) FROM Table15 GROUP BY Toy Range' at line 1
```

The error comes from the `PopulateTable` method in the code which populates the sorting panel. The issue was coming from the fact that "Range" is a reserved SQL key word. Because that collection was using "Range" as a column name, the SQL compiler was getting confused between the column name and the key word. The solution I used for this issue was to add backticks around the column names to distinguish them from key words:

```
self.MyCursor.execute("SELECT `" + str(MyResult1[x][0]) + "`, COUNT(*) FROM " +
self.OpenCollectionTable + " GROUP BY `" + str(MyResult1[x][0]) + "`")
```

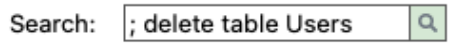
Creating New Collections

The process of creating new collections and deleting collections works as expected, I was unable to find any bugs during my testing

SQL Injection Tests

This set of tests was carried out as part of the destructive testing phase.

Test	Result

Using terminator characters (;) to begin new commands
 
This attack uses the semi-colon to chain a new statement to the search statement, thus injecting malicious code which can be used to delete or access sensitive data
	I wrote the following method to validate inputs and check for terminator characters:
	`def InjectionTest(self, input):`
	`  if ";" in input:`
	`  self.StatusBar.showMessage("System Integrity Error: Invalid Input")`
	`  else:`
	`  pass`
	This method is used whenever a user inputted string is used in a SQL statement to validate the input. If an input contains a semicolon, the operation is terminated, and the following message is shown in the status bar:
	 

Commenting out SQL code to crash the system
This form of attack is less severe than the use of terminator characters as it is less likely to give the attacker access to the system, but it can crash the program and mess with the data integrity of the backend.
Putting the following into the username text box on the login screen will compromise the system:
 
Giving this error:
`raise errors.get_mysql_exception(exc.errno, msg=exc.msg,`
`mysql.connector.errors.ProgrammingError: 1146 (42S02):`
	I had initially not accounted for this type of attack when coding the `InjectionTest` method, so I had to change the if statement to the following:
	`if ";" or "--" or "#" or "/*" or "*/" in input:`
	Now, the error message is shown in the status bar if the user tries to use the comment characters

Comments can be made using the characters #, -- and /* */	
SQL injection attacks using always true conditions Adding a condition like "1=1" or "False=False" after a terminator character can compromise the database. For example, inputting "WHERE 1=1" into the search bar will result in a bug which selects all the items in the database because the condition 1=1 will always evaluate to True	I updated the `InjectionTest` method to check for equals sign characters in user inputs to prevent against this kind of injection
As an extra layer of protection, I have endeavoured to use SQL parameters wherever possible. This means that, instead of using string concatenation operators (+) to pass values into SQL commands, I have used string substitution (%s). This method enforces validation through the SQL engine which checks each parameter to ensure that it is correct for its column and treats parameters literally, and not as part of the SQL to be executed.	

Speed Tests

In my test plan (page 57) I said that I would perform speed tests to measure the efficiency of the application. The tests are carried out by recording the time immediately before the test is carried out, then recording the time immediately after the test has finished and calculating the difference between the two recorded times. Each test was performed three times and a mean result was then calculated. These tests will also help me evaluate whether I have fulfilled success criteria numbers 2 and 27: "The software should be as optimised and lightweight as possible" and "Data should be stored in an efficient, concise and well-designed database".

Test	Time Taken	Comments
Launching the application	00:01.233	It must be noted that the application is currently being run through an interpreter which compiles and executes the code line-by-line. This time will probably be much faster when running a compiled version of the application.
Logging in to an account	00:00.059627	This is an acceptable time between the login button being pressed and the collection window being displayed. The delay is not noticeable.
Creating a new account	00:00.026030	Good time, delay not noticeable.
Creating a new collection	00:00.057922	Acceptable time, unnoticeable delay, note that this test was performed on a collection with 6 properties.
Opening a collection	Large collection (63 items): 00:00.197624	The delay is slightly noticeable; however, it is acceptable given the amount of processing the program is doing at this point. As the times show, performance worsens as the collection size grows
	Medium Collection (31 items): 00:00.176277	
	Small collection (10 items): 00:00.089334	

Adding an item	00:00.186517	This time includes the presence check and was carried out on an item with a thumbnail, acceptable time
Deleting an item	00:00.133660	This test was also performed on an item with a thumbnail
Editing an item	00:00.150570	I think this time could be improved by only updating the SQL columns that have been changed instead of resetting all of them, however this would only bring marginal improvements
Deleting duplicate items	00:00.106112	I was surprised by the result of this test as I was expecting it to be slower than just adding and deleting items, I think the reason for this is because the delete duplicates method doesn't have to fetch any user inputs. This test was carried out on a collection containing 4 duplicate items
Generating graphs	00:00.219394	This is definitely an aspect of the program which I think needs improving efficiency-wise. Launching this window and switching between graphs is not as snappy as it should be
Switching between graphs	00:00.163128	See above
Barcode search	Time to detect barcode: 00:00.179453 Time to carry out online search: 00:01.238431	Time to detect barcode is measured from the moment the barcode is present in the frame to the moment the program detects the barcode and highlights it in red. Time to carry out online search is measured from when the barcode is detected to when the search result is outputted to the user.
Performing an advanced search	00:00.113775	This test was carried out for an advanced search with 6 conditions performed on a collection with 31 items
Performing a simple search	00:00.068933	It makes sense that the simple search is quicker than the advanced search, however both search methods are efficient and both search functions worsen in performance on larger collections. Some form of indexing in the SQL backend would help improve performance for larger collections

You can watch the software being tested using this link:

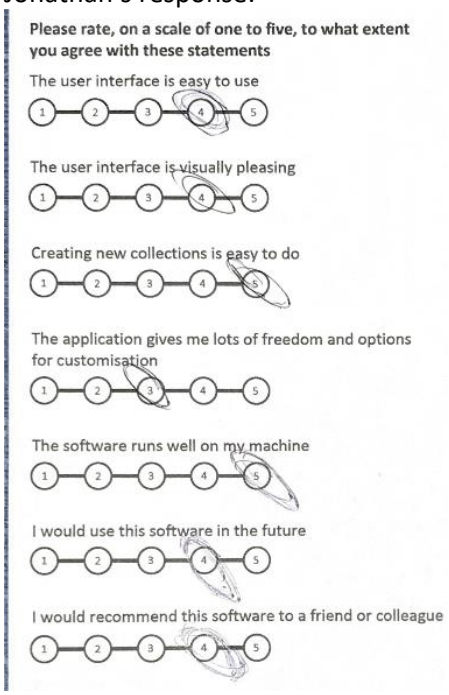
https://youtu.be/-m6wERMcz4Y

Stakeholder Testing

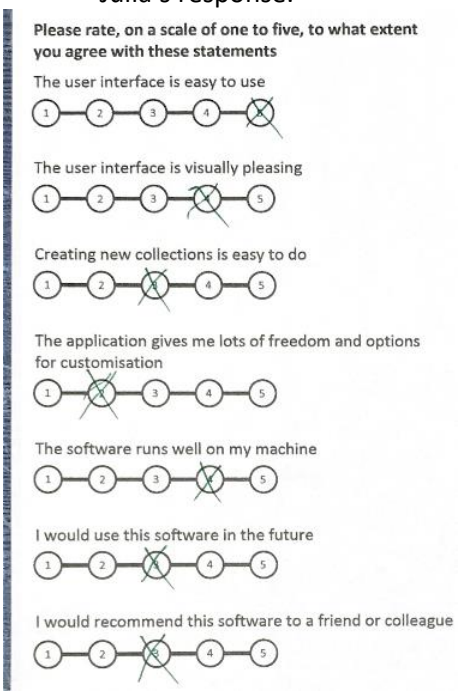
Jonathan, Julia and Sarah have been involved in the process since the analysis stage so I felt they were best placed to conduct my stakeholder testing as they could assess whether the software had met their original requirements. I installed a copy of the software on each of their machines (Jonathan is running Linux, Julia and Sarah are both running Windows) and gave them a questionnaire to fill out. The questionnaire consists of two parts: a section where stakeholders indicate how much they agree with a statement on a scale of 1 to 5, and a section for longer verbal answers.

Section 1

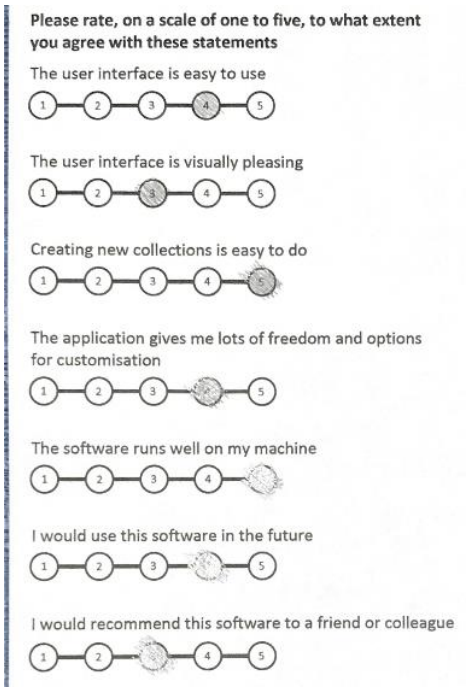
Jonathan's response:



Julia's response:



Sarah's response:



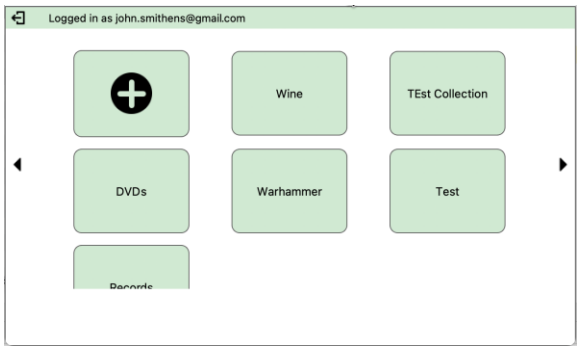
Section 1 Analysis

First of all, it is good to see that Jonathan, Sarah and Julia all answered positively when asked if they would use the software in the future and if they would recommend the software to acquaintances. This indicates that I have broadly fulfilled their requirements and created a usable application. I also think that overall, the feedback on the user interface, both in terms of ease-of-use and aesthetics, is largely positive. It is heartening to see that both John and Sarah answered 5 when asked if creating new collections was easy to do. Julia explains in section 2 why she put down 3 for that question. The fact that all respondents answered 4 or 5 to question 6 is proof that PyQt was a good framework to use for building my GUI as it seems to have ported well from my MacOS machine to Jonathan's Linux computer and Sarah and Julia's Windows machines. Question 4 seems to have received the most negative answers so giving the user more freedom and customisability is definitely an area in which the application could be improved.

Section 2

Q1: Did you encounter any bugs or issues whilst using the software?

Jonathan:
I sent you a screenshot of one of the bugs I encountered in the user interface running Anthology on Linux. Overall, however, that was the only bug I encountered. All the functions seemed to work well.



⟵ John's screenshot, the bottom of the collection window is being cut off.

Sarah:
I had no problems, the program didn't crash and all the buttons at the top of the window did their job well.

Julia:
I couldn't find any issues. There was one time when I thought there was a problem when I was creating new collections and I didn't remember adding a rating and rare column, but I soon realized that they were automatically added. I think you could maybe be a bit clearer that these columns will be added and maybe give us the option to delete them as I found the rare column a bit useless.

Q2: What are your thoughts on being able to create your own collections as opposed to pre-made collections?

Jonathan:
I think it's a good approach. It gives me lots of freedom as a user which is always a good thing. I do think that there maybe should have been some form of instruction for when I'm creating a collection for the first time. After you figure out the system it's easy to grasp and works well.

Sarah:

I prefer being able to create my own collections. I collect very niche items (Sarah collects novelty sewing thimbles) so having pre-made collections isn't really suitable to my needs. I also like that I can pick if a property is text or numerical, although it would be nice to have more property types like dates or currency.

Julia:
I think being able to create your own collections is the best approach. Maybe you could have had some templates though for some common types of collection.

Q3: Are there any features which you think are missing in the application?

Jonathan:
I think the app has a good balance between not having too little or too many features. That said, I would have like to have been able to change the colour scheme to something else and maybe some kind of grid layout for items as an alternative to the normal table.

Sarah:
It would have been nice to have a settings menu to give us even more control over the application. Like, it would have been nice to be able to change the theme colour for example. I also would have liked to have been able to open two or three different collections in the same window.

Julia:
The main thing missing for me was the option to edit collections after they had been created. I often found that I had created a collection and filled it with items, only to realise that I had missed a property and would be unable to store that piece of data without creating a whole new collection.

Q4: Are there any features which you think are unnecessary in the application?

Jonathan:
I didn't find myself using the barcode scanner very much but that's probably because the things I collect don't have barcodes.

Sarah:
Not that I can think of.

Julia:
I didn't use the advanced search very much. I also didn't use the Google Drive feature

Q5: How will this software benefit you and your collection?

Jonathan:
I uses to use a trusty pen and paper to keep track of my Warhammer figurines, but I think because Anthology lets me build my own collections and is so flexible, it could be a really useful tool for me. It will definitely help keep me more organized and it's just nice to have a record of everything to refer back to.

Sarah:
I collect items with a very practical use, I'm always taking thimbles from my collection and using them to sew or letting friends borrow them. Having some method of cataloguing them will help me keep track of things and hopefully stop me from losing anything.

Julia:
I don't think my collection is large enough yet to warrant regularly using the software, but I would definitely reconsider it as my collection grows.

Q6: Do you have any further comments?

Jonathan:
N.A.

Sarah:
I noticed that you took my advice about keeping the app easy to navigate on board, it's good to have a program that is powerful but not too complicated to use.

Julia:
N.A.

Section 2 Analysis

John's bug was unfortunate; however, it was only a cosmetic issue and none of my stakeholders encountered any bugs which broke the application. John's problem seems to be an issue with porting the application over to Linux and I resolved it by giving the affected widget a fixed size and changing its sizeHint. Overall, I believe the program is fairly stable and PyQt has done a good job of successfully porting the application to different operating systems. Julia's point about not being able to edit collections and delete default columns like Rating is really useful criticism and this is a feature which I had planned on including until time constraints got in the way.

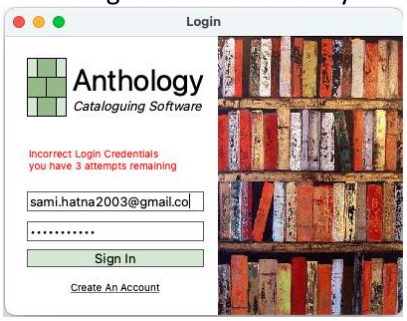
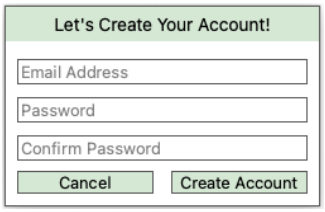
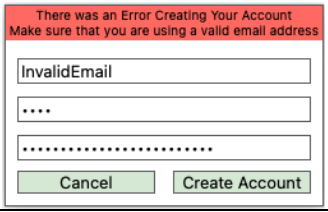
It was good to hear that all of my stakeholders preferred creating their own collections over having pre-made collections. John makes a good suggestion about including some instructions for using the application for new users and this would definitely improve the usability/accessibility of the software. Sarah's idea to have more data types for properties is a feature which I would definitely include if I were to do program this project again and Julia also makes a good recommendation about adding the option to use pre-made templates.

Two respondents mentioned that they would like to be able to choose the theme colour in their answers to question 3. This is definitely advice that I should take on board as one of my design objectives when beginning this project was to give the user as much freedom and control over the application as possible. I could also tie this colour customization into some of the accessibility features of my application by creating a high contrast mode for visually impaired users (this could be implemented as an alternative stylesheet to the default). Jonathan's idea about an alternative grid layout for displaying items by their thumbnails is an idea I considered during the analysis stage. It is definitely a more visually pleasing way of presenting items, but I do think that the table does a better job of presenting all the data about the items in an organized manner. However, it would be good to have both types of layout and allow the user to choose which one they would like to use.

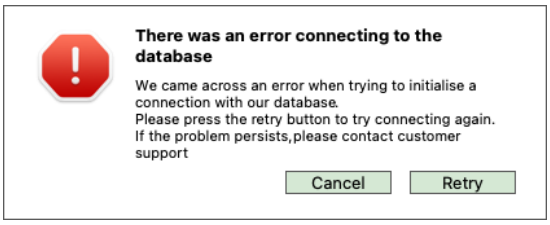
After reading Julia's answer to question 4, I asked her why she hadn't used the Google Drive feature and she told me it was because she doesn't have a Google Drive account. This means that it would be great for users like Julia if I added other cloud services to store backups in such as Dropbox or OneDrive, all of which have Python APIs which I could use.

Question 5 was used to gauge what benefits using the software had brought to each of my stakeholders. It's nice to see that the application will help John keep more organised as this is one of the principal benefits of collection cataloguing which I identified during the analysis stage. Julia's point about her collection not being large enough for Anthology to be of much use to her is interesting and suggests that this software is best aimed at more serious collectors with larger collections.

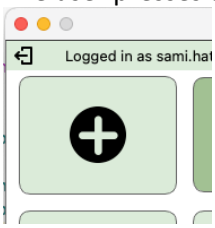
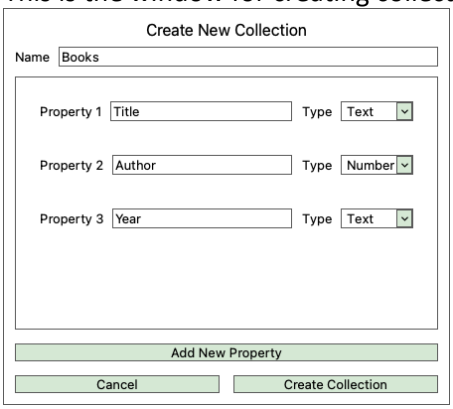
Evaluation Against Success Criteria

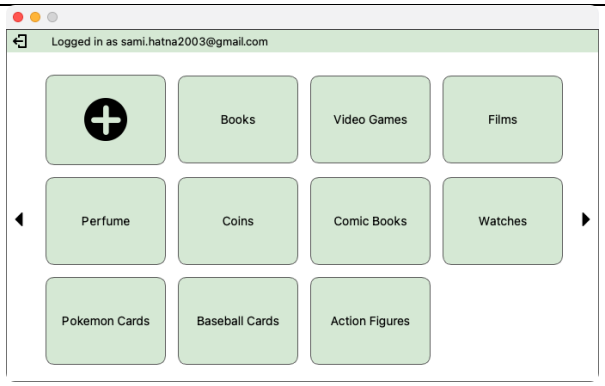
Success Criteria	Test Evidence
A login system to secure data and prevent unauthorised access and the ability to create new accounts	Login screen:  If the user inputs incorrect credentials, a message is displayed informing them of how many attempts they have remaining:  If the user exceeds 3 attempts, the window is temporarily disabled for 10 seconds. The time period will increase by 10 seconds with each group of 3 unsuccessful attempts:  Create account window:  New accounts are written to this SQL table: PK_Users | Email | PasswordHash 1 | sami.hatna2003@gmail.com | 3b4e2559b210ef2ef8f535df14cee16c3beeceb71… 2 | 14shatna@tythy.school | 93595faac52074bcb7efe99ceeb9b82fa5b3a1848… An error message is displayed if the user inputs an invalid email or the two passwords don't match: 

Evaluation:
The evidence shows that I have fully met this success criterion. The login system is fully functional and fully secure. The system is not vulnerable to SQL injection attacks (as evidenced on page 186). An extra level of protection is guaranteed through the disabling of the login screen as a way of dissuading brute force password attacks. The only way in which this aspect of the program could be improved upon would be using email verification to make users authenticate their accounts. This is a feature I would try to include if I were to do this project again.

An error message should be displayed if a connection can't be made with the SQL server	This is the popup displayed when the program fails to connect with the SQL backend:  When the cancel button is pressed, the application closes. When the retry button is pressed, the program tries connecting with the database again. If the connection is successful, the login screen will be displayed, if the connection fails, the popup will be displayed again

Evaluation:
The evidence shows that I have fully met this criterion. I think the text in the popup window is informative enough from a usability perspective to inform the user of what issues the program has encountered and what steps they can take to fix it. I also used a large informative icon in the window to immediately alert the user that something is wrong.

The ability to create your own collections with user specified column names	The user presses this button to create a new collection:  This is the window for creating collections:  Creating collections works by adding and deleting properties. Each property forms a column in the collection table. If the user presses the create collection button without filling in an input box, the empty box is highlighted in red and the collection is not created:  Collections are displayed in this window:



The user navigates collection using the arrow buttons. The top bar contains a logout button and displays the current logged in user. Users can right click on collections to delete them:

Collections are stored in this SQL table:

PK_Collections	CollectionName	TableName	FK_Users_Collections
2	Books	Table2	1
3	Video Games	Table3	1
4	Films	Table4	1
5	Records	Table5	1
6	Perfume	Table6	1
7	Coins	Table7	1
9	Comic Books	Table9	1

Each entry into this table is linked with another SQL table which contains the actual items:

PK_Ta...	Title	Developer	Console	Rare	Rating	Thumbnail
1	Skyrim	Bethesda	PS3	0	5	BLOB
2	Red Dead R...	Rockstar	PS4	0	5	BLOB
3	Red Dead R...	Rockstar	PS3	0	4	BLOB

Evaluation:

I believe that I have partially fulfilled this criterion. Users can create their collections and delete them. However, I would have liked to have allowed users to edit their collections by renaming them, deleting properties or adding new ones. The reason I didn't implement this was because I was unsure how adding and deleting columns may affect the data/referential integrity of my backend, so I felt it was safer to omit that feature and maintain the system's stability. However, if I had more time, I would have tried to implement this feature, adjusting the database structure where necessary.

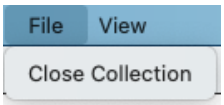
A table presenting items in a user's collection along with corresponding data

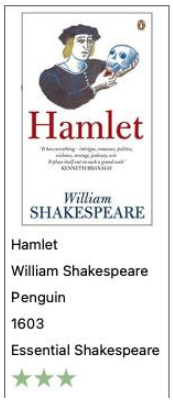
This is the table displayed for a video game collection I created:

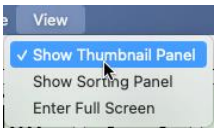
Title	Developer	Console	Rare	Rating
Skyrim	Bethesda	PS3		★★★★★
Red Dead Redemption 2	Rockstar	PS4		★★★★★
Red Dead Redemption	Rockstar	PS3		★★★★
Assassin's Creed Origins	Ubisoft	PS4		★★★
Assassin's Creed Unity	Ubisoft	PS4		★★★
Arkham Knight	Rocksteady	PS4	Rare	★★★★★
Arkham City	Rocksteady	Xbox 360		★★★★★
Spiderman	Insomniac	PS4		★★★★

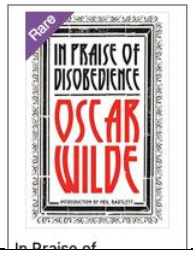
The table consists of the user-defined properties along with two default columns which are shown for every collection: Rare and Rating. Rare displays a custom purple tag for any item marked as rare and Rating displays a picture of 1 – 5 stars based on how the user rated the item.

The user can close the collection through the File menu:

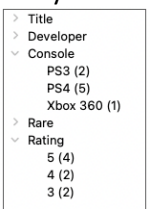


<table>
<tr><td colspan="2">Evaluation:
The evidence shows that I have fully met this criterion. The table displays all the items in the current collection in a clear and coherent manner with data sorted into columns. The table repopulates itself whenever an item is added, deleted or edited to display the changes to the collection.</td></tr>
<tr><td>The selected item's thumbnail and information should be displayed in a sidebar</td><td>When an item is selected in the table, it is displayed in the right sidebar:
 

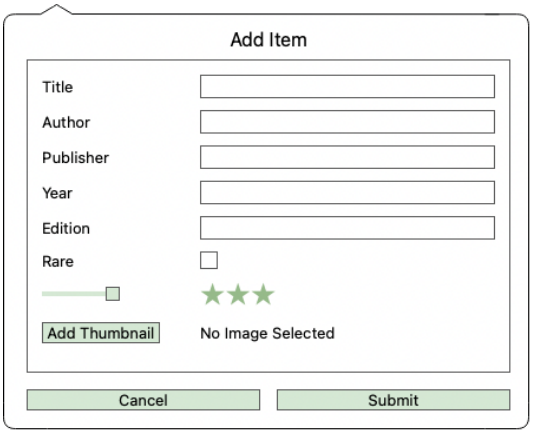
The user can show and hide this sidebar using the view menu:
 

Items marked as rare have a small tag displayed over their thumbnails:
 
</td></tr>
<tr><td colspan="2">Evaluation:
I believe that I have fully met this criterion. The screenshots show that selected items will have all their data presented in the sidebar along with the thumbnail they uploaded for the item. Furthermore, I went beyond the criterion by tagging the thumbnails of rare items and allowing the user to show/hide the thumbnail panel.</td></tr>
<tr><td>The ability to order collections in different ways</td><td>The user can change the order items are displayed in using a combobox above the items table:
 

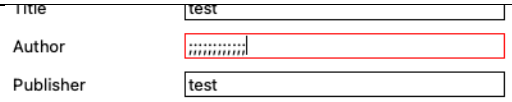
There are three order options: chronological, alphabetically ascending and alphabetically descending. The user can also use the keyboard shortcut shift + up/down arrow key to change the order</td></tr>
<tr><td colspan="2">Evaluation:
I have met this criterion; however, it would be possible in the future to add even more ordering options. For example, currently alphabetical ordering is determined using the first column in the collection, but I could add an option where the user can select which column they want to order the collection according to.</td></tr>
<tr><td>The option to filter/sort collections</td><td>Users can show the sorting panel via the view menu or using the keyboard shortcut command + shift + S. The sorting panel</td></tr>
</table>

<table>
<tr><td></td><td>breaks the collection down into its properties and presents them in a tree widget. Under each property is every type of entry for that property.
 

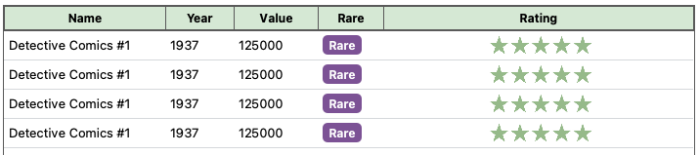
When the user selects something in the sorting panel, all the corresponding items are highlighted in the item table:
 

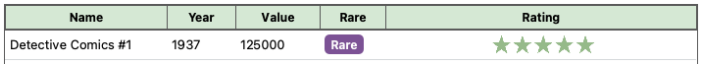
As you can see above, when I select “Rockstar” under the parent “Developer” in the sorting panel, all the video games developed by Rockstar are highlighted</td></tr>
<tr><td colspan="2">Evaluation:
The evidence shows that I have fully met this criterion. The tree widgets are dynamically generated and populated depending on the properties of the collection and can be used to filter collections. Having both the sorting and thumbnail panels displayed on the window can take up a lot of screen space on smaller laptop screens which is why I decided to allow them to be shown and hidden via keyboard shortcuts or the view menu.</td></tr>
<tr><td>The option to add and delete items from collections</td><td>Items are added to collections in a window which is displayed by pressing a tool button in the toolbar or using the keyboard shortcut command + shift + A. The window has input fields for each user-defined property of the collection as well as a rare checkbox, a rating slider and a button for uploading a thumbnail.
 

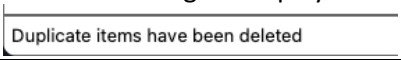
Each input field is validated before new items are added to the collection. Two forms of validation are used: a presence check and a check for suspicious characters used in SQL injection attacks. Integer properties are also limited to only allow numbers to be inputted. If the inputs don’t pass validation, the offending input boxes are highlighted in red:</td></tr>
</table>

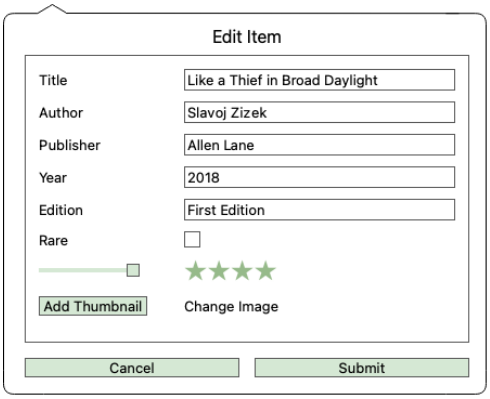
<table>
<tr><td></td><td> 

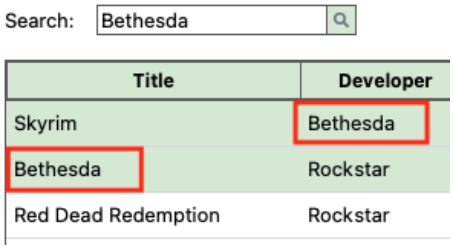
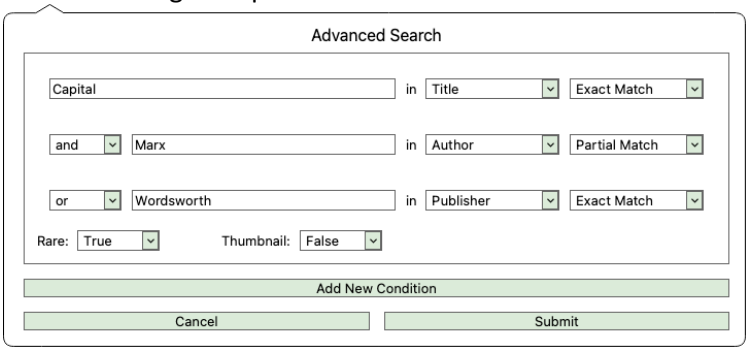
When a new item is added, the item table is refreshed to display the new item.
Items are deleted by selecting an item in the table and pressing the delete button.
 
</td></tr>
<tr><td colspan="2">Evaluation:
I have fully met this criterion. One of my design objectives was to make these core functions as simple and intuitive as possible and I believe I have met that standard.</td></tr>
<tr><td>The option to delete duplicate items</td><td>Users can delete duplicate items by pressing this button in the toolbar:
 

Take this example collection with containing four exact duplicate items:
 

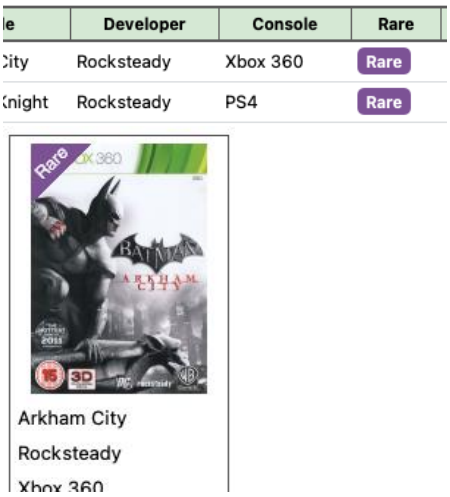
When the user presses the delete duplicates button, the duplicate items are deleted, leaving one item:
 

And this message is displayed in the status bar:
 
</td></tr>
<tr><td colspan="2">Evaluation:
I have fully met this criterion. I could not find any bugs during testing and the function runs quickly, even on large collections</td></tr>
<tr><td>The option to edit the data for items already stored in a collection</td><td>Users edit items by selecting the desired item in the table and then pressing the button in the toolbar.
 

This displays a window similar to the window for adding new items, but this time the input boxes are prefilled with the item's data
 
</td></tr>
</table>

	The user can then make their changes (including uploading a new thumbnail) and press submit to apply the changes or cancel to discard the edits. The item table is then updated to display the edits
Evaluation: I believe that I have partially met this criterion as editing items is simple and, after testing, all bugs have been ironed out. However, the feature could be fleshed out. For example, some features which I would add if I were to do this project again would be the ability to undo edits and a batch editing function for applying the same edit to multiple items	
A simple search function for quickly searching for an item	The simple search bar is located below the tool bar in the main window. Users input search strings into the bar and the SQL database is then queried based on this input. In this example, the search string "Bethesda" is used, and the search returns all items with Bethesda in one of their columns.  SQL code: `SELECT PK_Table3 FROM Table3 WHERE Title LIKE "%Bethesda%" OR Developer LIKE "%Bethesda%" OR Console LIKE "%Bethesda%" OR Rare LIKE "%Bethesda%" OR Rating LIKE "%Bethesda%"` The simple search is different from the advanced search as it performs a general search in all the table columns for the search data
Evaluation: Initially, I would say that I had not fulfilled this criterion as the original simple search was incredibly slow. However, with the changes outlined on page 181 I believe that I have now fulfilled the criterion as the function is much more efficient and usable. However, the simple search can be improved by having it take into account misspelt words.	
An advanced search function for creating more complicated queries	Advanced searches work by constructing queries by adding and deleting conditions. It is like a more high-level abstracted way of constructing SQL queries.  The results of advanced searches are highlighted in red to differentiate them from simple search results. 

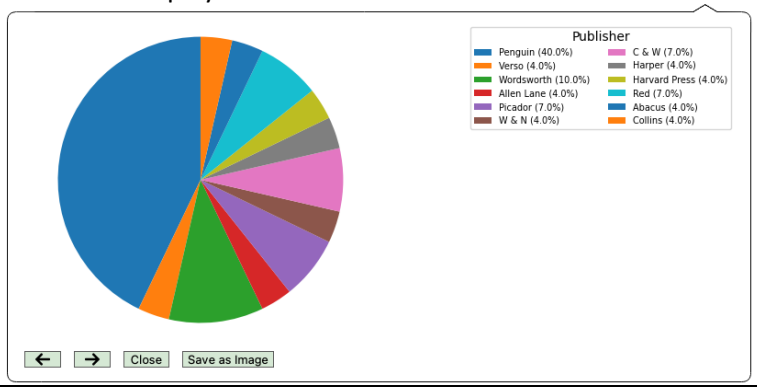
Evaluation:
This feature is intended for more serious users with large collections which they want to accurately query. With this in mind, I think I have found a good balance between complexity and ease of use

The option to mark rare or niche items with a special tag	When items are marked as rare a purple tag is displayed in the rare column of the item table and a label is displayed on their thumbnail: 

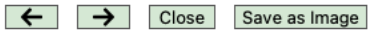
Evaluation:
This feature was heavily recommended by end-users in my questionnaire during the analysis stage. The evidence shows that I have fully met this criterion as rare items are tagged in a clear and noticeable way. The user can toggle an items rarity by editing the item and checking/unchecking the rarity check box. A possible improvement I could make would be to allow users to pick different colours apart from purple for the tags

The ability to store the market value of items	If users want to, they can add a value property to collections. However, they would have to find and record the values themselves

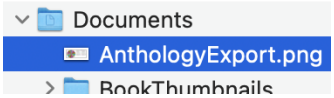
Evaluation:
I have not met this criterion as the process of recording the value of items is not automated. I could not implement this because users can create any kind of collection that they want to so it would be hard to find an online database which stores such a broad range of items that I could query. A work-around which I would do if I had more time would be to introduce collection templates for common collection types. For example, you could have a template for video game collections. If a user then creates a collection using one of these templates, this feature could be implemented as the application could then query a video game specific online database

The ability to generate graphical breakdowns of collections And **The option to export graphs as images**	When the user presses the graph button in the toolbar, this window is displayed: 

Users can navigate between graphs using the two arrow buttons and they can export their graphs as png files:



The save button opens a file dialogue and saves the image to the directory selected by the user:



A graph is generated for each property of the collection as well as a graph charting the size of the collection over time:

Collection sizes at different points in time are stored in this SQL table:

PK_Sizes	TimeRecorded	Magnitude	FK_Collections_Sizes
1	2021-01-04 18:11:53	0	2
2	2021-01-04 18:13:09	0	2
3	2021-01-04 18:20:33	1	2

Evaluation:

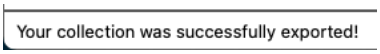
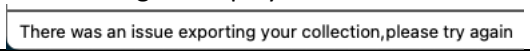
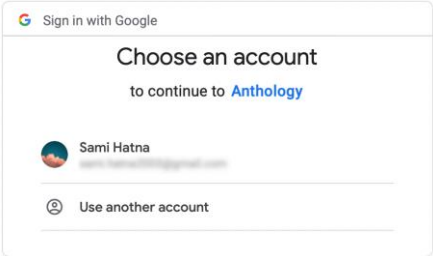
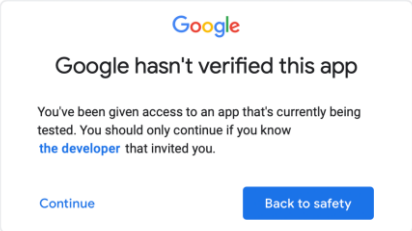
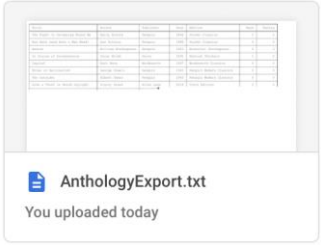
The evidence presented shows that I have fulfilled this criterion. Graphs are generated dynamically based on the properties of the collection and the user can export them. This component of the program can be expanded upon however by adding even more graphs presenting different statistics.

The option to export collections to a text file

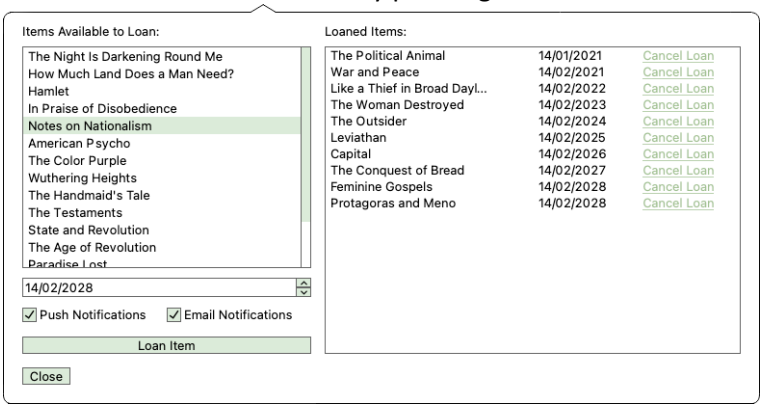
The user accesses this function through a sub-menu:

When they press the export as TXT file button, a file dialog is opened. The file dialog allows them to select a directory to save the TXT file to. This is what the final export looks like:

Title	Developer	Console	Rare	Rating
Skyrim	Bethesda	PS3	1	5
Red Dead Redemption 2	Rockstar	PS4	0	5
Red Dead Redemption	Rockstar	PS3	0	4
Assassin's Creed Origins	Ubisoft	PS4	0	3
Assassin's Creed Unity	Ubisoft	PS4	0	3
Arkham Knight	Rocksteady	PS4	0	5
Arkham City	Rocksteady	Xbox 360	0	5
Spiderman	Insomniac	PS4	0	4

	This message is displayed in the status bar if the export is successful:  This message is displayed if there is an issue whilst exporting: 
Evaluation: I believe that I have met this criterion as users can export collections as TXT files to their directory of choice. In the future, I could add options to allow the user to customise the appearance of the collection in the TXT file. I could also allow exports to different file formats.	
The ability to back collections up to a Google Drive account	When users press the Google Drive button in the Export submenu, a browser window is opened, taking them to the Google sign in page:  Because the application isn't currently verified with Google, the user will be asked if they trust the application and to grant it Google Drive permissions:  The browser window then closes and the collection is uploaded to the cloud: 
Evaluation: Similar to the previous criterion, I believe that I have met this criterion, but it could be improved by giving the user a variety of different file formats to export collections to. If I were to do the project again, I would have a dedicated export window where these decisions could be made rather than just a submenu	
A loan management system And **Generate email notifications when a loaned item is due**	The user loans items through a dedicated window composed of two lists. One list contains all the items which are available to loan and the other contains all the items already out on loan. Items are loaned out by selecting them in the first list, setting a due date in the menu, choosing how you would like to be notified (emails or push notifications) and then pressing the

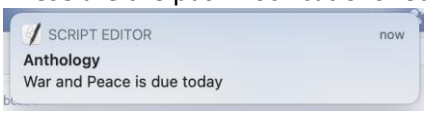
submit button. The item is then moved from one list to the other. Users can cancel loans by pressing the cancel button



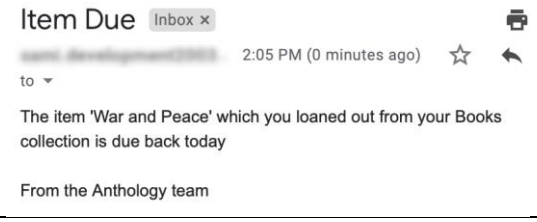
Loaned items are stored in this SQL table:

PK_Lo...	DueDate	FK_Co...	KeyInC...	Email	Push
25	2021-01-13	2	9	1	1
26	2021-01-13	2	17	1	1
27	2021-01-13	2	13	1	1

These are the push notifications received when an item is due:



And these are the emails sent when an item is due:



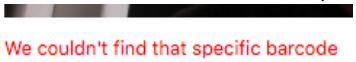
Evaluation:

The evidence presented shows that I have fully met this criterion. Users can easily loan items out and schedule when they are due back in. They are also given the freedom to choose how they can be notified when the item is due. An improvement I could add would be the option to quickly loan an item out by right clicking on it in the main item table. It would also be nice to use HTML to send more stylised emails when items are due. However, the core functionality is present and easy to use

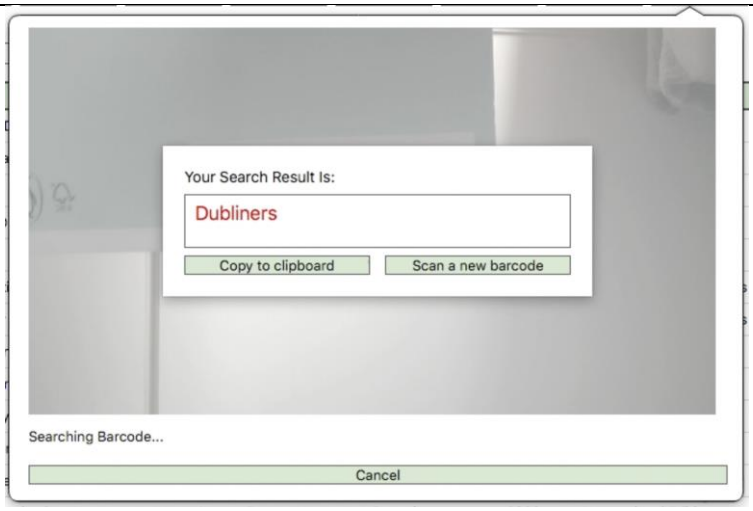
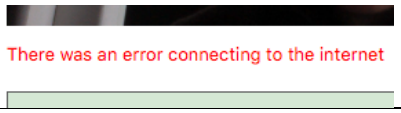
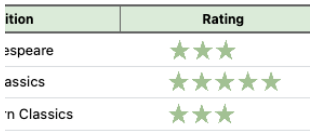
The ability to search for and add items using their ISBN codes (barcodes)

The barcode search takes its input from a live video feed. Users can then position a barcode within the frame and the program will scan it:

It will then search for the barcode in two online databases. If the barcode can't be found, this message is shown:



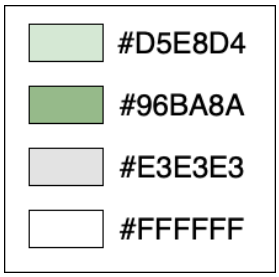
If the search is successful, this popup window is displayed:

	 The user can then copy the result to their clipboard and begin a new search If there is an error capturing video, this image is shown instead of the video feed:  The webcam connection has been severed If there is an error connecting to the internet, this message is shown: 
Evaluation: The evidence shows that I have fully met this criterion. Users can search for items by their barcode by positioning the item in the webcam's view and allowing the programmer to scan its barcode. The program also has several failsafe processes to deal with issues such as internet connectivity or webcam troubles.	
The option to store item ratings taken from online sources	I have not included a system for taking ratings from online sources for the same reasons I couldn't get online market values (page 195). I worked around this limitation by allowing the user to rate items themselves using a slider in the add item window:  The item ratings will then be displayed in the collection's rating column and underneath the item's thumbnail: 
Evaluation: I have not met this criterion; however, I have developed a work-around which keeps the idea of rating items but places it in the hands of the user rather than online sources.	
A settings menu	The program does not have a settings menu. However, I have tried to use the menu bar as an alternative by including features such as showing/hiding the thumbnail and sorting

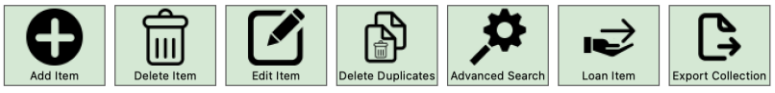
	panels and the ability to change the order items are presented in.
Evaluation: I have not met this criterion. Sarah picked up on this during stakeholder testing and made some suggestions as to what I could add to a future settings menu. If I were to do this project again and include a settings menu it would have options such as the ability to change the theme colours (possibly a dark mode), options for personalising accounts and an option for users to configure their own keyboard shortcuts	
Keyboard shortcuts assigned to different functions	Page 182 contains a list of all the program's shortcuts and what they do
Evaluation: I believe that I have fully met this criterion as the keyboard shortcuts cover all of the main functions of the program. The shortcuts are intended for power users who prioritise speed and efficiency. I believe that I have successfully captured this user base as using the shortcuts massively increases the efficiency of cataloguing items.	

Usability

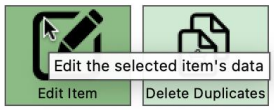
I have taken a number of steps to ensure that the application is accessible and usable for a wide range of users with different needs. I have used a consistent design scheme throughout the application to keep it visually cohesive. All the main toolbar buttons are large and brightly coloured as visual stimuli will help elderly and visually impaired users. Key buttons also have icons to make their purpose clear. Each button has a tooltip which displays when the cursor hovers over the button. The purpose of these tooltips is to inform users as to what each button does if they are unsure.



Application colour scheme



All the buttons in the toolbar have large, informative icons



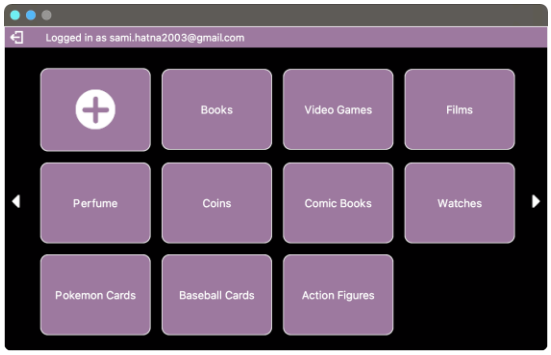
Each button has a tooltip

I have made a conscious attempt to make all the most complicated core features of the program, such as creating new collections or performing advanced searches, as simple and easy to use as possible to keep the application accessible for both advanced and novice users. For example, I have decomposed advanced searching into a system of adding and deleting condition widgets which I believe is much simpler and more intuitive than writing queries as SQL code.

Every window contains a cancel button. This is a response to Julia's feedback during the design stage when she said that she often found it hard to navigate applications with lots of different windows. The simple window structure I have decided to use means that a user can always return to the main application window by simply pressing the cancel button in the active window. Each window also has a large clear title to help with navigation.

The application does not need to be navigated using a mouse and keyboard. Instead, the user can use the tab key on their keyboard to move between different UI elements and the enter key to press buttons. This accessibility feature is aimed at users with motor impairments who may be unable to use a mouse or trackpad.

PyQt is a cross-platform GUI library so my application is able to run on MacOS, Linux and Windows operating systems. This was verified during stakeholder testing when the application ran successfully on John's Linux machine and Sarah's Windows machine. Furthermore, Qt's cross-platform capabilities also means that my application is compatible with native accessibility features for the OS it is running on such as text-to-speech programs or screen magnifiers.



This is the application running in inverted colours mode which is an accessibility feature for visually impaired users

There are some changes I could make to improve the accessibility of the application. For example, some of the stakeholder feedback I received mentioned that it was unclear how some features worked the first time they were using the application. I could solve this issue by creating a virtual tour for first-time users which would teach them how to use all the application's features.

Limitations

The main limitation of the application is the number of dependencies it requires. For it to run on a computer, it must have Python, Qt and MySQL installed. Qt is a particularly large dependency. This presents an issue for users with a limited amount of storage on their machines. There is no way to address this limitation beyond compressing the application file when it has been compiled.

Another limitation which comes with using Qt for my UI is licensing issues. Qt is licensed under the LGPL software license. One of the conditions of LGPL is that there be a sufficient separation between my code and the library code so as to give users the ability to run the application with their own installation of Qt rather than the one which comes packaged with the application. This condition makes it harder to package Qt applications as single binary deployments which complicates the installation process.

Currently the application is only available on desktop computers which cuts out a massive group of end-users on smartphones. In the future, a mobile app could be developed which would help collectors who want to keep track of their collections on the go.

Another limitation is the legality of web-scraping. My software uses web-scraping techniques to query online databases for scanned barcodes. This use of web scraping is legal as far as I am aware because I am not violating the intellectual property rights of the website owners, however if I were to implement more features which make use of web scraping into my program, I will have to make sure that I do not cross the line into illegality.

There are two limitations of the login/accounts system: there is currently no way to verify email addresses when a new account is created and there is no password recovery system. These are limitations which I would certainly address if I were to do the project again. I could verify emails by using smtplib to send a verification link to that email with a link for the user to click. Password

recovery would be done by sending an email to the user's email with a link sending them to a webpage where they can change their password.

The final limitation is that the program's backend is hosted locally through a UNIX socket connection. If I had more time and money, I could overcome this limitation by purchasing a SQL server to host the backend on which would be accessed remotely through an Internet connection.

Maintenance

The main functionality around which the program revolves is this system of creating collections and adding, editing and deleting items within that collection. Because of the modular nature of the software, any further features which are added can easily implemented by creating new modules and integrating them within the existing code. The program broadly consists of two dimensions: the SQL backend and the Python/PyQt frontend. This division between backend and frontend means that I can easily change the graphical user interface without worrying about messing up the backend. For example, during the final round of stakeholder feedback, two of my stakeholders expressed a desire to set their own colour scheme for the app instead of the default lime green. This is an aesthetic feature that can easily be implemented without worrying about compromising the core backend functionality of the program. The backend can also be maintained without compromising the frontend. For example, one improvement I would consider making to the database in the future is to add some form of indexing to speed up database queries and operations. One issue to consider with maintenance is mentioned above in the Limitations section and that is the complications associated with deploying Qt applications as single binary deployments because of the license. The temporary way to overcome this limitation is to provide the user with instructions on how to install the Qt dependencies on their own, however this is a complicated process for normal users, so the best solution is to purchase a license. The code is fully annotated which will assist maintenance if the program is handed over to a new developer.

Closing Remarks

I began this project with the intention of creating a piece of software which allows amateur collectors to quickly, efficiently and easily catalogue their collections. My end product successfully fulfils this original brief and expands upon it with a selection of extra features I conceived of during the course of the project. The finished application meets most of the success criteria I established in the analysis stage and received favourable reviews from a sample of end-users. Whilst carrying out the project I have learnt many programming skills as well as organisational skills such as keeping to a development time schedule. There are, however, also aspects in which the program could be improved. The improvements I believe to be most necessary if I were to do the project again would be allowing users to edit collections by adding new properties or deleting old ones and adding some form of indexing in the SQL database to speed up backend operations.

Final Code

Login.py

```
import sys
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
import mysql.connector
import hashlib
from functools import partial
import re
import CollectionWindow

# This object inherits from the QMessageBox class which is used to create popup windows
# This popup is displayed when a connection with the SQL database can't be established
class SQLErrorPopup(QMessageBox):
    def __init__(self, RootWindow):
        super(SQLErrorPopup, self).__init__()
        self.RootWindow = RootWindow
        # Initialise window elements
        self.setText("Database Error")
        self.setText("There was an error connecting to the database")
        self.setInformativeText(
        """We came across an error when trying to initialise a connection with our database.
Please press the retry button to try connecting again.
If the problem persists, please contact customer support""")
        self.setIcon(QMessageBox.Critical)
        self.setWindowFlags(Qt.FramelessWindowHint)

        self.setStandardButtons(QMessageBox.Retry | QMessageBox.Cancel)
        # If the user presses the retry button, the TestConnection function will be
          executed again
        self.button(QMessageBox.Retry).clicked.connect(self.RootWindow.TestConnection)
        # If they press the cancel button, the application will close
        self.button(QMessageBox.Cancel).clicked.connect(sys.exit)

        # Apply stylesheet to popup window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        self.exec()

# This object inherits from the QMainWindow class
# It displays the login window, validates user credentials and grants the user access if
  they provide valid login details
class LoginWindow(QMainWindow):
    def __init__(self):
        super(LoginWindow, self).__init__()

        # The central widget is a container for all the other widgets the window displays
        self.CentralWidget = QWidget()
        self.CentralWidget.setObjectName("BorderlessWidget")
        self.setCentralWidget(self.CentralWidget)
        self.setWindowTitle("Login")

        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Keeps track of how many unsuccessful login attempts have been made
        self.Attempts = 4
        # Keeps track of how many times the user has made more than 4 login attempts
        self.Magnitude = 1

        self.InitUI()
        # TestConnection has been repurposed as a method of the LoginWindow class
        self.TestConnection()
```

```
    # Tests the program's connection with the SQL database
    def TestConnection(self):
        # Try establishing a connection with the database and creating a cursor
        try:
            self.MyDB = mysql.connector.connect(
                host="localhost",
                user="root",
                password="B4tm4nisbae",
                database="Anthology"
            )
            self.MyCursor = self.MyDB.cursor()
            self.show()
        # If the program encounters an error when trying to connect, an instance of the
          error popup is executed
        except mysql.connector.Error:
            self.SQLErrorPopupInstance = SQLErrorPopup(RootWindow = self)

    # This method initialises all the UI elements in the window
    def InitUI(self):
        self.LoginVBL1 = QVBoxLayout()

        # Create sublayout
        InnerHBL = QHBoxLayout()
        # Align elements within layout to the left
        InnerHBL.setAlignment(Qt.AlignLeft)
        InnerHBL.setSpacing(5)
        # Create a label for displaying the logo image
        LogoLBL = QLabel()
        # Create pixmap for logo image
        LogoPixmap = QPixmap("Resources/Logo3.png").scaledToHeight(60)
        LogoLBL.setPixmap(LogoPixmap)
        LogoLBL.setAlignment(Qt.AlignCenter)
        InnerHBL.addWidget(LogoLBL)
        # Create another sublayout for the text
        InnerVBL = QVBoxLayout()
        InnerVBL.setSpacing(0)
        InnerVBL.setAlignment(Qt.AlignVCenter)
        # Create heading label
        LogoLBL1 = QLabel("Anthology")
        # Style heading
        LogoLBL1.setStyleSheet("font-size: 30px;")
        InnerVBL.addWidget(LogoLBL1)
        # Create subheading label
        LogoLBL2 = QLabel("Cataloguing Software")
        # Style subheading
        LogoLBL2.setStyleSheet("font: italic")
        # Add widgets to layout
        InnerVBL.addWidget(LogoLBL2)
        InnerHBL.addLayout(InnerVBL)
        self.LoginVBL1.addLayout(InnerHBL)

        self.WelcomeLBL = QLabel("Welcome Back!")
        self.WelcomeLBL.setStyleSheet("padding-top: 20px; padding-bottom: 20px; font-size:
        15px;")
        self.LoginVBL1.addWidget(self.WelcomeLBL)

        # Text box for user to input username
        self.UsernameTB = QLineEdit()
        self.UsernameTB.setPlaceholderText("Username")
        self.LoginVBL1.addWidget(self.UsernameTB)

        # Text box for user to input password
        self.PasswordTB = QLineEdit()
        # EchoMode determines how the text will be displayed
        # In this case it will be hidden because the password is sensitive information
        self.PasswordTB.setEchoMode(QLineEdit.Password)
        self.PasswordTB.setPlaceholderText("Password")
        self.LoginVBL1.addWidget(self.PasswordTB)

        self.SignInBTN = QPushButton("Sign In")
```

```
        # Bind sign in button to CheckCredentials function
        self.SignInBTN.clicked.connect(self.CheckCredentials)
        self.LoginVBL1.addWidget(self.SignInBTN)

        # This button is used to create a new account
        self.CreateAccountBTN = QPushButton("Create An Account")
        self.CreateAccountBTN.setStyleSheet("""
        QPushButton:pressed { color: D5E8D4; }
        QPushButton:hover:!pressed { color: #96BA8A; }
        QPushButton { border: 0px; background-color: white; font-size: 10px; }
        """)
        ButtonFont = QFont()
        ButtonFont.setUnderline(True)
        self.CreateAccountBTN.setFont(ButtonFont)
        # Bind CreateAccountBTN to CreateAccountMethod function
        self.CreateAccountBTN.clicked.connect(self.CreateAccountMethod)
        self.LoginVBL1.addWidget(self.CreateAccountBTN)

        # Displays artwork alongside login form
        ArtLBL = QLabel()
        ArtPixmap = QPixmap("Resources/LoginArt.png").scaledToHeight(280)
        ArtLBL.setPixmap(ArtPixmap)

        # Combine layouts and add layouts to central widget
        self.LoginHBL1 = QHBoxLayout()
        self.LoginHBL1.addLayout(self.LoginVBL1)
        self.LoginHBL1.addWidget(ArtLBL)
        self.LoginHBL1.setContentsMargins(0, 0, 0, 0)
        self.LoginVBL1.setContentsMargins(22, 22, 0, 22)
        self.CentralWidget.setLayout(self.LoginHBL1)

        # Prevent user from being able to resize window
        self.setFixedSize(400, self.minimumHeight())

    # This function checks user inputted credentials against the credentials stored in the
      database
    def CheckCredentials(self):
        # If the user hasn't exceeded the maximum amount of attempts permitted...
        if self.Attempts > 1:
            self.InputUsername = self.UsernameTB.text()
            # Passwords stored in the database are hashed so the user input has to be
              hashed before it can be compared
            self.InputPassword = hashlib.sha224(bytes(self.PasswordTB.text(), encoding =
            "utf-8")).hexdigest()

            # Retrieve all entries from the Users table
            self.MyCursor.execute("SELECT * FROM Users")
            MyResult = self.MyCursor.fetchall()

            # Boolean value indicates whether a successful login has been made
            self.SuccessfulLogin = False

            # Iterate through query result and compare each username and password
              combination with the user inputs
            for x in range(0, len(MyResult)):
                if MyResult[x][1] == self.InputUsername:
                    if MyResult[x][2] == self.InputPassword:
                        self.SuccessfulLogin = True
                        self.ActiveUserID = MyResult[x][0]
                        break

            # If the user has inputted correct credentials, they will be logged into the
              application and the login window will close
            if self.SuccessfulLogin:
                self.CollectionWindow = CollectionWindow.CollectionWindow(MyCursor =
                self.MyCursor, MyDB = self.MyDB, ActiveUserID = self.ActiveUserID)
                self.close()
            # If their crednetials are incorrect, an error message is displayed informing
              them of how many attempts they have remaining
            else:
```

```
                self.Attempts -= 1
                self.WelcomeLBL.setText("Incorrect Login Credentials\nyou have " +
                str(self.Attempts) + " attempts remaining")
                self.WelcomeLBL.setStyleSheet("color: red; font-size: 10px; padding-top:
                20px; padding-bottom: 10px;")
            # If user has exceeded max attempts, login screen is disabled for a period of time
            else:
                self.WelcomeLBL.setText("Too many incorrect login attempts\nLogin disabled for
                {0} seconds".format(10 * self.Magnitude))
                self.SignInBTN.setEnabled(False)
                self.UsernameTB.setEnabled(False)
                self.PasswordTB.setEnabled(False)
                self.UsernameTB.setStyleSheet("border-color: red;")
                self.PasswordTB.setStyleSheet("border-color: red;")
                self.SignInBTN.setStyleSheet("border-color: red; background-color: #ececec")
                self.setCursor(Qt.ForbiddenCursor)
                # Time period for which window is disabled is calculated by multiplying 10
                  seconds by the variable self.Magnitude
                # Each time the user exceeds the max, self.Magnitude is incremented so they
                  will have to wait 10 seconds longer
                QTimer.singleShot(10000 * self.Magnitude, partial(self.TimerSlot))

        # TimerSlot is called when the QTimer is complete
        # It re-enables the login window, resets the no. of attempts and increments
          self.Magnitude
        def TimerSlot(self):
            self.SignInBTN.setEnabled(True)
            self.UsernameTB.setEnabled(True)
            self.PasswordTB.setEnabled(True)
            self.WelcomeLBL.setText("Welcome Back!")
            self.WelcomeLBL.setStyleSheet("padding-top: 20px; padding-bottom: 20px; font-size:
            15px;")
            self.UsernameTB.setStyleSheet("border-color: black; border: 1px solid #5A5A5A;")
            self.PasswordTB.setStyleSheet("border-color: black; border: 1px solid #5A5A5A;")
            self.SignInBTN.setStyleSheet("border-color: black; border: 1px solid #5A5A5A;")
            self.setCursor(Qt.ArrowCursor)
            self.Attempts = 4
            self.Magnitude += 1

        # When the user presses the create account button, an instance of CreateAccountWindow
          is initialised
        def CreateAccountMethod(self):
            self.CreateAccountWindowInstance = CreateAccountWindow(RootPos = self.geometry(),
            MyCursor = self.MyCursor, MyDB = self.MyDB)

# This object inherits from the QMainWindow class
# It displays the create account window, validates user inputs and adds the new account to
  the database
class CreateAccountWindow(QMainWindow):
    def __init__(self, RootPos, MyCursor, MyDB):
        super(CreateAccountWindow, self).__init__()

        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # The database cursor and connection have to be passed into the object as
          parameters
        self.MyCursor = MyCursor
        self.MyDB = MyDB

        self.CentralWidget = QWidget()
        self.setCentralWidget(self.CentralWidget)
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint))
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setWindowModality(Qt.ApplicationModal)

        # Position new window in top center of parent window
        self.move(int(RootPos.x() - ((250 - 400) / 2)), int(RootPos.y()))

        self.InitUI()
```

```
        self.show()
        self.setFixedSize(250, 158)

    def InitUI(self):
        # Initialise layouts
        self.CreateAccountVBL1 = QVBoxLayout()
        self.CreateAccountVBL2 = QVBoxLayout()

        self.HeaderLBL = QLabel("Let's Create Your Account!")
        self.HeaderLBL.setAlignment(Qt.AlignCenter)
        self.HeaderLBL.setStyleSheet("font-size: 14px; padding: 5px; background-color:
        #D5E8D4; border-left: 1px solid #5A5A5A; border-right: 1px solid #5A5A5A; border
        top: 1px solid #5A5A5A;")
        self.CreateAccountVBL1.addWidget(self.HeaderLBL)
        self.CreateAccountVBL1.setContentsMargins(0, 0, 0, 0)
        self.CreateAccountVBL2.setContentsMargins(10, 0, 10, 10)

        # Text box for user to input their email address
        self.EmailTB = QLineEdit()
        self.EmailTB.setPlaceholderText("Email Address")
        self.CreateAccountVBL2.addWidget(self.EmailTB)

        # Text box for user to input their password
        self.PasswordTB = QLineEdit()
        self.PasswordTB.setPlaceholderText("Password")
        self.PasswordTB.setEchoMode(QLineEdit.Password)
        self.CreateAccountVBL2.addWidget(self.PasswordTB)

        # Text box for user to input their password a second time to make sure user doesn't
          mistype password
        self.ConfirmPasswordTB = QLineEdit()
        self.ConfirmPasswordTB.setPlaceholderText("Confirm Password")
        self.ConfirmPasswordTB.setEchoMode(QLineEdit.Password)
        self.CreateAccountVBL2.addWidget(self.ConfirmPasswordTB)

        self.CreateAccountBTN = QPushButton("Create Account")
        self.CreateAccountBTN.clicked.connect(self.SubmitNewAccount)

        self.CancelBTN = QPushButton("Cancel")
        self.CancelBTN.clicked.connect(self.close)

        # Combine layouts and add layouts to central widget
        self.CreateAccountHBL1 = QHBoxLayout()
        self.CreateAccountHBL1.addWidget(self.CancelBTN)
        self.CreateAccountHBL1.addWidget(self.CreateAccountBTN)
        self.CreateAccountVBL2.addLayout(self.CreateAccountHBL1)
        self.CreateAccountVBL1.addLayout(self.CreateAccountVBL2)
        self.CentralWidget.setLayout(self.CreateAccountVBL1)

    # This function validates user inputs and then adds credentials to database or displays
      an error message accordingly
    def SubmitNewAccount(self):
        # Use regex to verify input is a valid email address
        EmailCheck = re.match("[^z]+@[^z]+\.[^z]+", self.EmailTB.text())

        # If any text boxes are empty, the two password inputs don't match or the email
          address isn't valid, an error message is displayed
        if (self.ConfirmPasswordTB.text() != self.PasswordTB.text() or
            not self.ConfirmPasswordTB.text() or
                not self.PasswordTB.text() or
                not self.EmailTB.text() or
                EmailCheck == None):
            self.HeaderLBL.setStyleSheet("font-size: 10px; background-color: #FF6961;
            border-left: 1px solid #5A5A5A; border-right: 1px solid #5A5A5A; border-top:
            1px solid #5A5A5A;")
            self.HeaderLBL.setText("There was an Error Creating Your Account\nMake sure
            that you are using a valid email address")

        # If user inputs are valid, they are stored in the Users table
```

```
        else:
            PasswordHash = hashlib.sha224(bytes(self.PasswordTB.text(), encoding = "utf
            8")).hexdigest()
            self.MyCursor.execute("INSERT INTO Users (Email, PasswordHash) VALUES (%s,
            %s)", (self.EmailTB.text(), PasswordHash))
            self.MyDB.commit()
            self.close()

# Main program loop
if __name__ == "__main__":
    # A QApplication instance must be created before any windows are displayed
    App = QApplication(sys.argv)
    App.setWindowIcon(QIcon("Resources/Icon.png"))
    # Begin database connection test
    Root = LoginWindow()
    # Begin program execution
    sys.exit(App.exec())
```

CollectionWindow.py

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
import math
import Login
import OpenCollection
from datetime import datetime

# This object stores the window which displays collections belonging to the logged in user
  and allows them to create new collections
class CollectionWindow(QMainWindow):
    def __init__(self, MyCursor, ActiveUserID, MyDB):
        super(CollectionWindow, self).__init__()

        # The database connection, cursor and the id of the user currently logged in are
          passed into this object as arguments
        # The database connection, cursor and the id of the user currently logged in are
          passed into this object as arguments
        self.ActiveUserID = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window widget and initialise main layouts
        self.CentralWidget = QWidget()
        self.setCentralWidget(self.CentralWidget)
        self.CollectionVBL1 = QVBoxLayout()
        self.CollectionVBL1.setContentsMargins(0, 0, 0, 0)
        self.CollectionGL1 = QGridLayout()
        self.CentralWidget.setLayout(self.CollectionVBL1)

        # ContainerWidget contains the status bar displayed at the top of the window
        # The status bar conatins a lgout button and a label outputting the currently
          logged in user
        self.ContainerWidget = QWidget()
        self.ContainerWidget.setObjectName("ContainerWidget")
        self.CollectionHBL1 = QHBoxLayout()
        self.CollectionHBL1.setContentsMargins(0, 0, 0, 0)
        LogoutBTN = QPushButton()
        LogoutBTN.clicked.connect(self.Logout)
        LogoutBTN.setIcon(QIcon("Resources/LogoutIcon.png"))
        LogoutBTN.setStyleSheet("width: 20px; padding: 5px; border-bottom: 0px; border
        right: 0px;")
        self.MyCursor.execute("SELECT Email FROM Users WHERE PK_Users = " +
```

```
        str(self.ActiveUserID))
        SignedInUserLBL = QLabel("Logged in as " + str(self.MyCursor.fetchall()[0][0]))
        SignedInUserLBL.setStyleSheet("background-color: #D5E8D4; padding: 5px; border-top:
        1px solid #5A5A5A;")
        self.CollectionHBL1.addWidget(LogoutBTN)
        self.CollectionHBL1.addWidget(SignedInUserLBL)
        self.CollectionHBL1.addStretch()
        self.ContainerWidget.setLayout(self.CollectionHBL1)
        self.CollectionVBL1.addWidget(self.ContainerWidget)

        self.InitUI()

        # Horizontal layout contains the navigation buttons and the scroll area used to
          display the collections
        self.CollectionHBL2 = QHBoxLayout()
        self.CollectionHBL2.setContentsMargins(10, 0, 10, 0)
        self.CollectionVBL1.addLayout(self.CollectionHBL2)

        # Initialise scroll area and set the widget which will be displayed within it
        self.CollectionScrollArea = QScrollArea()
        self.CollectionScrollArea.setObjectName("BorderlessWidget")
        # The user will navigate the scroll area using buttons, so the scroll bars are set
          to hidden
        self.CollectionScrollArea.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.CollectionScrollArea.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.CollectionScrollArea.setWidgetResizable(True)
        self.CollectionWidget = QWidget()
        self.CollectionWidget.setObjectName("BorderlessWidget")
        self.CollectionWidget.setLayout(self.CollectionGL1)
        self.CollectionScrollArea.setWidget(self.CollectionWidget)

        # LeftArrowBTN and RightArrowBTN are used to navigate left and right within the
          scroll are
        self.LeftArrowBTN = QPushButton()
        self.LeftArrowBTN.setIcon(QIcon("Resources/LeftArrow.png"))
        self.LeftArrowBTN.setStyleSheet("background-color: white; border: 0px;")
        self.LeftArrowBTN.clicked.connect(self.MoveLeft)
        self.LeftArrowBTN.setFixedSize(10, 300)

        self.RightArrowBTN = QPushButton()
        self.RightArrowBTN.setIcon(QIcon("Resources/RightArrow.png"))
        self.RightArrowBTN.setStyleSheet("background-color: white; border: 0px;")
        self.RightArrowBTN.clicked.connect(self.MoveRight)
        self.RightArrowBTN.setFixedSize(10, 90)

        # Add widgets to layout
        self.CollectionHBL2.addWidget(self.LeftArrowBTN)
        self.CollectionHBL2.addWidget(self.CollectionScrollArea)
        self.CollectionHBL2.addWidget(self.RightArrowBTN)

        self.setFixedSize(690, 410)
        self.CollectionVBL1.addStretch()

        # Display window
        self.show()

    # This function is called when the user wants to log out of the current account
    # It creates a new instance of the LoginWindow class and closes the current window
    def Logout(self):
        self.LoginWindowInstance = Login.LoginWindow()
        self.close()

    # MoveLeft and MoveRight are executed when the left and right navigation buttons are
      pressed
    # They simply move the scrollbar along by 160 pixels
    def MoveLeft(self):
        self.CollectionScrollArea.horizontalScrollBar().setValue(
            self.CollectionScrollArea.horizontalScrollBar().value() - 160)

    def MoveRight(self):
```

```
        self.CollectionScrollArea.horizontalScrollBar().setValue(
            self.CollectionScrollArea.horizontalScrollBar().value() + 160)

    # This function populates the scroll area with buttons for each collection and a button
      for creating collections
    def InitUI(self):
        # Remove any widgets already in the layout before repopulating it
        for i in reversed(range(self.CollectionGL1.count())):
            self.CollectionGL1.itemAt(i).widget().setParent(None)

        # Initialise create collection button and add to grid layout
        self.CreateCollectionBTN = QToolButton()
        self.CreateCollectionBTN.setIcon(QIcon("Resources/AddItemIcon.png"))
        self.CreateCollectionBTN.setIconSize(QSize(50, 50))
        self.CreateCollectionBTN.setToolTip("Create a new collection")
        self.CreateCollectionBTN.setObjectName("CollectionButton")
        self.CreateCollectionBTN.setFixedWidth(140)
        self.CreateCollectionBTN.clicked.connect(self.CreateNewCollection)
        self.CollectionGL1.addWidget(self.CreateCollectionBTN, 0, 0)

        # Retrieve all collections that belong to the currently logged in user from the
          database
        self.MyCursor.execute("SELECT * FROM Collections WHERE FK_Users_Collections = " +
        str(self.ActiveUserID))
        self.MyResult1 = self.MyCursor.fetchall()

        # The following code iterates through the result of the database query and creates
          a button for each collection which is added to the grid layout
        # Counters are used to keep track of which row and column the program is in
        Populated = False
        RowCounter = 0
        ResultCounter = 0
        ColumnCounter = 1
        self.CollectionBTNArray = []
        Division = math.ceil(len(self.MyResult1) / 3)
        while not Populated:
            if ResultCounter == len(self.MyResult1):
                Populated = True
            elif ColumnCounter <= Division:
                self.CollectionBTNArray.append(CollectionButton(Data =
                self.MyResult1[ResultCounter]))
                self.CollectionBTNArray[ResultCounter].DeleteCollectionSignal.connect(
                          self.DeleteCollectionSlot)
                self.CollectionBTNArray[ResultCounter].OpenCollectionSignal.connect(
                          self.OpenCollectionSlot)
                self.CollectionGL1.addWidget(self.CollectionBTNArray[ResultCounter],
                RowCounter, ColumnCounter)
                ColumnCounter += 1
                ResultCounter += 1
            else:
                RowCounter += 1
                ColumnCounter = 0

    # This slot is executed when this class receives the DeleteCollection signal from the
      CollectionButton object
    # Its purpose is to delete the collection and its corresponding table from the database
      and refresh the window
    def DeleteCollectionSlot(self, IDForDeletion):
        self.MyCursor.execute("DROP TABLE Table" + str(IDForDeletion))
        self.MyCursor.execute("DELETE FROM Collections WHERE PK_Collections = " +
        str(IDForDeletion))
        self.MyCursor.execute("DELETE FROM Sizes WHERE FK_Collections_Sizes = " +
        str(IDForDeletion))
        self.MyDB.commit()
        self.InitUI()

    # This function is executed when the create collection button is pressed
    # It creates an instance of the AddCollectionWindow object
    def CreateNewCollection(self):
        self.AddCollectionWindowInstance = AddCollectionWindow(MyCursor = self.MyCursor,
```

```
        ActiveUserID = self.ActiveUserID, MyDB = self.MyDB, RootPos = self.geometry())
        self.AddCollectionWindowInstance.CreateCollectionSignal.connect(self.InitUI)

    # This function is executed when one of the collection buttons is pressed
    # It creates an instance of OpenCollectionInstance, passing in the relevant arguments
      for that collection
    def OpenCollectionSlot(self, IDForOpening):
        self.OpenCollectionInstance = OpenCollection.OpenCollectionWindow(MyCursor =
        self.MyCursor, ActiveUserID = self.ActiveUserID, MyDB = self.MyDB, OpenCollectionID
        = IDForOpening)
        self.close()

# Subclass of QPushButton used to create buttons for each collection
class CollectionButton(QPushButton):
    # This signal is emitted when the user selects Delete Collection in the widget's
      context menu
    DeleteCollectionSignal = pyqtSignal(int)
    OpenCollectionSignal = pyqtSignal(int)
    # The only argument passed into this object is the name and primary key of the
      collection it is supposed to represent
    def __init__(self, Data):
        super(CollectionButton, self).__init__()
        self.Data = Data
        self.setText(self.Data[1])
        # Properties are values attached to a widget, this will be used when the user wants
          to open or delete the collection by clicking on the button
        self.setProperty("Key", self.Data[0])
        self.setObjectName("CollectionButton")
        self.setFixedWidth(140)
        self.clicked.connect(self.OpenCollection)

    # This event occurs when the user right clicks on a button
    # It create a context menu with the option to delete a collection
    def contextMenuEvent(self, event):
        PropertyWidgetContextMenu = QMenu(self)
        DeleteAction = PropertyWidgetContextMenu.addAction("Delete Collection")
        Action = PropertyWidgetContextMenu.exec_(self.mapToGlobal(event.pos()))
        # If the user chooses to delete the collection, DeleteCollectionSignal emits
        if Action == DeleteAction:
            self.setParent(None)
            self.DeleteCollectionSignal.emit(self.Data[0])

    # When the button is clicked, this method is called
    def OpenCollection(self):
        self.OpenCollectionSignal.emit(self.Data[0])

# This object creates the window which allows users to create new collections
class AddCollectionWindow(QMainWindow):
    # This signal is emitted to inform the CollectionWindow class that a new collection has
      been added to the database and the display needs to be refreshed
    CreateCollectionSignal = pyqtSignal()
    def __init__(self, ActiveUserID, MyCursor, MyDB, RootPos):
        super(AddCollectionWindow, self).__init__()

        self.ActiveUserID = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB

        # Instances of PropertyWidget are created dynamically, so they will be stored in
          this array
        self.PropertyWidgetArray = []
        # Counter keeps track of how many instances of PropertyWidget exist
        self.PropertyWidgetCounter = 0

        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint))
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setWindowModality(Qt.ApplicationModal)

        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())
```

```
        # Set central widget and initialise vertical box layout
        self.CentralWidget = QWidget()
        self.setCentralWidget(self.CentralWidget)
        self.CreateCollectionVBL1 = QVBoxLayout()
        self.CentralWidget.setLayout(self.CreateCollectionVBL1)

        self.InitUI()

        self.setFixedSize(450, 400)
        self.move(int(RootPos.x() + 112), int(RootPos.y()))

        self.show()

    def InitUI(self):
        # Create header and add to layout
        HeaderLBL = QLabel("Create New Collection")
        HeaderLBL.setStyleSheet("font-size: 16px;")
        HeaderLBL.setAlignment(Qt.AlignCenter)
        self.CreateCollectionVBL1.addWidget(HeaderLBL)

        # The first text input box is for inputting the name of the collection
        self.CreateCollectionHBL1 = QHBoxLayout()
        self.CreateCollectionHBL1.addWidget(QLabel("Name"))
        self.NameTB = QLineEdit()
        self.CreateCollectionHBL1.addWidget(self.NameTB)
        self.CreateCollectionVBL1.addLayout(self.CreateCollectionHBL1)
        # The user creates collections by adding named properties, these property widgets
        are displayed in a scroll area
        self.CreateCollectionScrollArea = QScrollArea()
        self.CreateCollectionScrollArea.setWidgetResizable(True)
        self.CreateCollectionScrollArea.verticalScrollBar().setStyleSheet("QScrollBar {
        border-right: 0px; border-top: 0px; border-bottom: 0px; } QScrollBar::handle {
        border-top: 0px; border-bottom: 0px; }")
        self.ScrollAreaWidget = QWidget()
        self.ScrollAreaWidget.setObjectName("BorderlessWidget")
        self.ScrollAreaLayout = QVBoxLayout()
        self.ScrollAreaLayout.setAlignment(Qt.AlignTop)
        self.ScrollAreaWidget.setLayout(self.ScrollAreaLayout)
        self.CreateCollectionScrollArea.setWidget(self.ScrollAreaWidget)
        self.CreateCollectionVBL1.addWidget(self.CreateCollectionScrollArea)

        # Button for adding a new property
        self.AddNewPropertyBTN = QPushButton("Add New Property")
        self.AddNewPropertyBTN.clicked.connect(self.AddNewProperty)
        self.CreateCollectionVBL1.addWidget(self.AddNewPropertyBTN)

        # Two buttons, one for cancelling the operation and one for submitting the new
          collections
        self.CreateCollectionHBL2 = QHBoxLayout()
        self.CancelBTN = QPushButton("Cancel")
        self.CancelBTN.clicked.connect(self.close)
        self.CreateCollectionHBL2.addWidget(self.CancelBTN)
        self.CreateCollectionBTN = QPushButton("Create Collection")
        self.CreateCollectionBTN.clicked.connect(self.CreateCollection)
        self.CreateCollectionHBL2.addWidget(self.CreateCollectionBTN)
        self.CreateCollectionVBL1.addLayout(self.CreateCollectionHBL2)

        self.AddNewProperty()

    # This function adds a new property widget to the scroll area
    def AddNewProperty(self):
        self.PropertyWidgetArray.append(PropertyWidget(Num=self.PropertyWidgetCounter+ 1))
        self.ScrollAreaLayout.addWidget(
                            self.PropertyWidgetArray[self.PropertyWidgetCounter])
        self.PropertyWidgetArray[self.PropertyWidgetCounter].DeletePropertySignal.connect(
                            self.DeletePropertySlot)
        self.PropertyWidgetCounter += 1

    # This function stores a new entry in the Collections table and create the collection's
```

```
    corresponding table
    # See database design section of report for more info (page 36)
    def CreateCollection(self):
        # Presence check to ensure all fields are filled in
        EmptyString = False
        EmptyPropertyWidgetIndexes = []
        for x in range(0, len(self.PropertyWidgetArray)):
            if self.PropertyWidgetArray[x].PropertyNameTB.text() == "":
                EmptyString = True
                EmptyPropertyWidgetIndexes.append(x)

        # If presence check flags an unfilled field, execute error function
        if EmptyString or self.NameTB.text() == "":
            self.CreateCollectionError(EmptyPropertyWidgetIndexes)
        else:
            # Get name of new collections
            Name = self.NameTB.text()
            # Insert new entry into Collections table
            self.MyCursor.execute("INSERT INTO Collections (CollectionName,
            FK_Users_Collections) VALUES ('{0}', '{1}')".format(Name, self.ActiveUserID))
            self.MyCursor.execute("SELECT MAX(PK_Collections) FROM Collections")
            self.MyResult1 = self.MyCursor.fetchall()
            # New table name consists of "table" + primary key of the entry that has just
              been stored in Collections table
            SQLStatement = "CREATE TABLE Table" + str(self.MyResult1[0][0]) + " (PK_Table"
            + str(self.MyResult1[0][0]) + " INTEGER(255) NOT NULL AUTO_INCREMENT PRIMARY
            KEY, "
            # Piece together SQL statement for creation of new table
            for x in range(0, len(self.PropertyWidgetArray)):
                # Get property name and datatype and use as column in new table
                PropertyName = self.PropertyWidgetArray[x].PropertyNameTB.text()
                DataType = self.PropertyWidgetArray[x].TypeCB.currentText()
                if DataType == "Number":
                    DataType = "Integer"
                    SQLStatement = SQLStatement + "`" + PropertyName + "` " + DataType +
                    "(255), "
                else:
                    DataType = "Varchar"
                    SQLStatement = SQLStatement + "`" + PropertyName + "` " + DataType +
                    "(500), "
            # All collections will have a Rare column, a Rating column and a Thumbnail
              column
            SQLStatement = SQLStatement + " Rare BOOLEAN, Rating INTEGER(20), Thumbnail
            LONGBLOB)"
            self.MyCursor.execute(SQLStatement)
            # Add new table name to the entry in Collections
            self.MyCursor.execute("UPDATE Collections SET TableName = 'Table" +
            str(self.MyResult1[0][0]) + "' WHERE PK_Collections = " +
            str(self.MyResult1[0][0]))

            # When a new collection is created, an entry into Sizes is made with the
              starting size of 0
            self.MyCursor.execute("INSERT INTO Sizes (TimeRecorded, Magnitude,
            FK_Collections_Sizes) VALUES ('" + str(datetime.now()) + "', 0, " +
            str(self.MyResult1[0][0]) + ")")

            # Save changes to database
            self.MyDB.commit()
            # Emit signal to inform CollectionWindow that a new collection has been created
              and it needs to be refreshed
            self.CreateCollectionSignal.emit()
            self.close()

    # Function is executed when a field has been left empty
    # It takes a list containing the indexes of the empty widgets as a parameter and sets
      the border of these widgets to red
    def CreateCollectionError(self, OffendingWidgets):
        for x in OffendingWidgets:
            self.PropertyWidgetArray[x].PropertyNameTB.setStyleSheet("border: 1px solid
```

```
            #FF0000")

    # This function is executed when the Delete Property signal is received
    # It deletes the specified property widget
    def DeletePropertySlot(self, Sender):
        self.PropertyWidgetArray.remove(Sender)
        self.PropertyWidgetCounter -= 1
        for x in range(0, len(self.PropertyWidgetArray)):
            self.PropertyWidgetArray[x].PropertyLBL.setText("Property " + str(x + 1))

# Subclass of QWidget
# Each property widget consists of a text input for the name of the property and a combobox for
its datatype
class PropertyWidget(QWidget):
    # This signal is emitted when the user wants to delete a property
    DeletePropertySignal = pyqtSignal(QObject)
    def __init__(self, Number):
        super(PropertyWidget, self).__init__()

        self.setObjectName("BorderlessWidget")
        # Initialise widget layout
        self.PropertyWidgetHBL1 = QHBoxLayout()
        self.setLayout(self.PropertyWidgetHBL1)

        self.PropertyLBL = QLabel("Property " + str(Number))
        self.PropertyWidgetHBL1.addWidget(self.PropertyLBL)

        # Text box for inputting property name
        self.PropertyNameTB = QLineEdit()
        self.PropertyWidgetHBL1.addWidget(self.PropertyNameTB)

        TypeLBL = QLabel("Type")
        self.PropertyWidgetHBL1.addWidget(TypeLBL)

        # Combo box for specifiying if property datatype is text or numerical
        self.TypeCB = QComboBox()
        self.TypeCB.addItems(["Text", "Number"])
        self.PropertyWidgetHBL1.addWidget(self.TypeCB)

    # Event executed when user right clicks on property widget
    def contextMenuEvent(self, event):
        PropertyWidgetContextMenu = QMenu(self)
        DeleteAction = PropertyWidgetContextMenu.addAction("Delete Property")
        Action = PropertyWidgetContextMenu.exec_(self.mapToGlobal(event.pos()))
        # If user clicks Delete Property, emit signal
        if Action == DeleteAction:
            self.setParent(None)
            self.DeletePropertySignal.emit(self)
```

OpenCollection.py

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
import CollectionWindow
import AddItemWindow
import EditItemWindow
import AdvancedSearch
import GraphWindow
import BarcodeSearch
import LoanItem
import ExportCollection
from datetime import datetime

# This object is the window which displays the collection the user has chosen to open
class OpenCollectionWindow(QMainWindow):
    # The window takes the cursor, the database connection and the primary key of the logged in
      user as arguments
    # It also takes the primary key of the collection which is currently open
    def __init__(self, MyCursor, ActiveUserID, MyDB, OpenCollectionID):
```

```
        super(OpenCollectionWindow, self).__init__()

        self.ActiveUserID = ActiveUserID
        self.Yes = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.OpenCollectionID = OpenCollectionID
        self.MyCursor.execute("SELECT TableName, CollectionName FROM Collections WHERE
        PK_Collections = " + str(self.OpenCollectionID))
        # OpenCollectionTable is the name of the collection which is currently open
        # It is obtained by joining the string "TABLE" with the primary key of the collection in
          the Collections table
        MyResult = self.MyCursor.fetchall()
        self.OpenCollectionTable = MyResult[0][0]
        self.setWindowTitle(str(MyResult[0][1]))

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window's central widget and initialise main layout
        self.CentralWidget = QWidget()
        self.OpenCollectionVBL1 = QVBoxLayout()
        self.CentralWidget.setLayout(self.OpenCollectionVBL1)
        self.setCentralWidget(self.CentralWidget)

        self.InitUI()

        # Initialise menu bar
        self.MainMenu = QMenuBar(self)
        # Add file menu to menubar
        self.FileMenu = self.MainMenu.addMenu(" &File")
        # Create action for closing current collection window and going back to CollectionWindow
        self.CloseCollectionAction = QAction("Close Collection", self)
        # Bind action to function
        self.CloseCollectionAction.triggered.connect(self.CloseCollection)
        # Add action to file menu
        self.FileMenu.addAction(self.CloseCollectionAction)

        # Create view menu and add to menu bar
        self.ViewMenu = self.MainMenu.addMenu(" &View")
        # The view menu will have two actions: one for show/hiding the thumbnail panel and one
          for showing/hiding the sorting panel
        self.ShowThumbnailAction = QAction("Show Thumbnail Panel")
        # setCheckable displays a tick next to menu actions when they are active
        self.ShowThumbnailAction.setCheckable(True)
        self.ShowThumbnailAction.setChecked(True)
        # Connect action to method
        self.ShowThumbnailAction.triggered.connect(self.ShowThumbnail)
        # Add action to menu
        self.ViewMenu.addAction(self.ShowThumbnailAction)
        # Create action for showing/hiding the sorting panel
        self.ShowSortingAction = QAction("Show Sorting Panel")
        self.ShowSortingAction.setCheckable(True)
        self.ShowSortingAction.setChecked(False)
        # Connect action to slot
        self.ShowSortingAction.triggered.connect(self.ShowSorting)
        # Add action to menu
        self.ViewMenu.addAction(self.ShowSortingAction)

        # Create shortcuts and bind them to methods
        # Keyboard inputs are captured using the QKeySequence object
        self.AddItemShortcut = QShortcut(QKeySequence("Ctrl+shift+A"), self)
        self.AddItemShortcut.activated.connect(self.AddItemMethod)
        self.DeleteItemShortcut = QShortcut(QKeySequence("Ctrl+shift+D"), self)
        self.DeleteItemShortcut.activated.connect(self.DeleteItemMethod)
        self.EditItemShortcut = QShortcut(QKeySequence("Ctrl+shift+E"), self)
        self.EditItemShortcut.activated.connect(self.EditItemMethod)
        self.GraphShortcut = QShortcut(QKeySequence("Ctrl+shift+G"), self)
        self.GraphShortcut.activated.connect(self.GenerateGraphsMethod)
        self.OrderShortcut1 = QShortcut(QKeySequence(Qt.SHIFT + Qt.Key_Up), self)
        self.OrderShortcut1.activated.connect(lambda: self.TabShortcut(self.OrderShortcut1))
        self.OrderShortcut2 = QShortcut(QKeySequence(Qt.SHIFT + Qt.Key_Down), self)
        self.OrderShortcut2.activated.connect(lambda: self.TabShortcut(self.OrderShortcut2))
```

```
        self.ShowShortcut = QShortcut(QKeySequence("Ctrl+shift+S"), self)
        self.ShowShortcut.activated.connect(lambda: self.ShowHideShortcut(self.ShowShortcut))
        self.HideShortcut = QShortcut(QKeySequence("Ctrl+shift+H"), self)
        self.HideShortcut.activated.connect(lambda: self.ShowHideShortcut(self.HideShortcut))

        # Display window at maximum size
        self.showMaximized()
        self.show()

    # This method searches the parameter InputForTesting for any characters which may be used in
      a SQL injection attack
    def InjectionTest(self, InputForTesting):
        # If the string contains any suspicious characters, cancel the operation and output an
          error message in the status bar
        if ";" or "--" or "#" or "/*" or "*/" or "=" in InputForTesting:
            self.StatusBar.showMessage("System Integrity Error: Invalid Input")
        else:
            pass

    # This method acts as an arbiter for when the show/hide keyboard shortcut is used
    def ShowHideShortcut(self, sender):
        # If the user has pressed CMD+shift+S then show the two sidebars
        if sender == self.ShowShortcut:
            self.ThumbnailGroupBox.show()
            self.SortingGroupBox.show()
            self.ShowThumbnailAction.setChecked(True)
            self.ShowSortingAction.setChecked(True)
        # If the user has pressed CMD+shift+H then hide the two sidebars
        else:
            self.ThumbnailGroupBox.hide()
            self.SortingGroupBox.hide()
            self.ShowThumbnailAction.setChecked(False)
            self.ShowSortingAction.setChecked(False)

    # This method acts as an arbiter for when the shift+tab shortcut is pressed
    def TabShortcut(self, sender):
        # It changes the order items are displayed in
        if sender == self.OrderShortcut1:
            if self.OrderCB.count() - 1 == self.OrderCB.currentIndex():
                self.OrderCB.setCurrentIndex(0)
            else:
                self.OrderCB.setCurrentIndex(self.OrderCB.currentIndex() + 1)
        else:
            if self.OrderCB.currentIndex() == 0:
                self.OrderCB.setCurrentIndex(self.OrderCB.count() - 1)
            else:
                self.OrderCB.setCurrentIndex(self.OrderCB.currentIndex() - 1)

    # Initialise UI elements
    def InitUI(self):
        # InitToolbar is the function which creates the window toolbar
        self.InitToolbar()

        # Add 20 pixels of space between toolbar and search bar
        self.OpenCollectionVBL1.addSpacing(20)

        # SearchBarHBL is the layout which stores the search bar and the sorting menu
        self.SearchBarHBL = QHBoxLayout()
        self.SearchBarHBL.addWidget(QLabel("Search:"))
        self.SubHBL = QHBoxLayout()
        self.SubHBL.setSpacing(0)
        # SearchBarTB is the text box in which the user inputs search terms
        self.SearchBarTB = QLineEdit()
        self.SearchBarTB.setStyleSheet("border-right: 0px;")
        self.SearchBarTB.setFixedWidth(150)
        # When the user presses enter in a QLineEdit, it emits the editingFinished signal
        # I have bound this signal to the function SimpleSearch so that whenever the user
          presses enter, the function will be executed
        self.SearchBarTB.editingFinished.connect(self.SimpleSearch)
        self.SubHBL.addWidget(self.SearchBarTB)
        self.SearchBTN = QPushButton()
        self.SearchBTN.clicked.connect(self.SimpleSearch)
        self.SearchBTN.setIcon(QIcon("Resources/SearchIcon.png"))
        self.SearchBTN.setFixedSize(20, 20)
```

```
self.SubHBL.addWidget(self.SearchBTN)
self.SearchBarHBL.addLayout(self.SubHBL)
self.SearchBarHBL.addSpacing(50)
self.SearchBarHBL.addWidget(QLabel("Order:"))
# OrderCB is a combobox which allows user to choose which order they would like to
  display their collection in
self.OrderCB = QComboBox()
self.OrderCB.addItems(["Date Added", "Ascending", "Descending"])
self.OrderCB.setFixedWidth(130)

# When the user selects a new item in the combobox, it emits a signal, I have bound this
  signal to the function PopulateTable
# This means that if the user selects a different order for items, the table will be
  repopulated in that order
self.OrderCB.currentTextChanged.connect(self.PopulateTable)
self.SearchBarHBL.addWidget(self.OrderCB)
# Add horizontal box layout to main layout
self.OpenCollectionVBL1.addLayout(self.SearchBarHBL)
self.SearchBarHBL.addStretch()

# StatusBar displays a horizontal bar along the bottom of the window for presenting
  status information
self.StatusBar = QStatusBar()
self.setStatusBar(self.StatusBar)
# StatusLBL will be used to convey information to the user such as collection size or
  when processes are complete
self.StatusLBL = QLabel("")
self.StatusLBL.setStyleSheet("padding-left: 5px")
self.StatusBar.addWidget(self.StatusLBL, 1)
self.CreditLBL = QLabel("Anthology v1.0 created by Sami Hatna")
self.CreditLBL.setStyleSheet("padding-right: 5px")
self.StatusBar.addPermanentWidget(self.CreditLBL)

# This horizontal box layout displays the table in which the collection is presented and
  the group box for displaying item thumbnails
self.OpenCollectionHBL1 = QHBoxLayout()

# Create groupbox for displaying sorting TreeWidget
self.SortingGroupBox = QGroupBox()
self.SortingGroupBox.setFixedWidth(150)
# Set layout for group box
self.SortingLayout = QVBoxLayout()
self.SortingLayout.setContentsMargins(0, 0, 0, 0)
# Initialise tree widget
# The tree widget will have parent items for each column/property within the collection
# These parents will then have child items for each category within that column
self.SortingTree = QTreeWidget()
# Style tree widget
self.SortingTree.setStyleSheet("QTreeWidget { selection-color: black; }")
self.SortingTree.verticalScrollBar().setStyleSheet("QScrollBar { border-right: 0px;
border-top: 0px; border-bottom: 0px; } QScrollBar::handle { border-top: 0px; border
bottom: 0px; }")
self.SortingTree.setObjectName("BorderlessWidget")
# Hide tree widget headers
self.SortingTree.setHeaderHidden(True)
self.SortingTree.setAttribute(Qt.WA_MacShowFocusRect, 0)
# Connect the signal emitted when the selected item in the tree widget changes to a slot
self.SortingTree.itemSelectionChanged.connect(self.SortingTreeClicked)
# Add tree widget to layout
self.SortingLayout.addWidget(self.SortingTree)
self.SortingGroupBox.setLayout(self.SortingLayout)
# Add group box to main window layout
self.OpenCollectionHBL1.addWidget(self.SortingGroupBox)
# The sorting panel is hidden on startup, the user can show it through the view menu
self.SortingGroupBox.hide()

# CollectionTable is the table which displays collection items
self.CollectionTable = QTableWidget()
self.CollectionTable.setShowGrid(False)
# Disable table editing, if users want to edit items they have to do it using the edit
  toolbutton
self.CollectionTable.setEditTriggers(QAbstractItemView.NoEditTriggers)
# The first column is hidden because it stores the primary key of each item
self.CollectionTable.setColumnHidden(0, True)
```

```
        self.CollectionTable.horizontalHeader().setStretchLastSection(True)
        # Users can only select rows rather than individual cells
        self.CollectionTable.setSelectionBehavior(QAbstractItemView.SelectRows)
        self.CollectionTable.verticalHeader().setVisible(False)
        # When the user selects a new row in the table, DisplayImage is executed to display the
          thumbnail for that item in ThumbnailGroupBox
        self.CollectionTable.itemSelectionChanged.connect(self.DisplayImage)
        self.OpenCollectionHBL1.addWidget(self.CollectionTable)
        self.CollectionTable.verticalScrollBar().setStyleSheet("QScrollBar::handle { border:
        0px; }")
        self.CollectionTable.horizontalScrollBar().setStyleSheet("QScrollBar::handle { border
        left: 0px; border-right: 0px; } QScrollBar { border-left: 0px; border-right: 0px; }")
        self.PopulateTable()
        self.CheckTableScrollBar(False)

        # ThumbnailGroupBox displays the thumbnail and metadata of the currently selected item
        self.ThumbnailGroupBox = QGroupBox()
        self.ThumbnailGroupBox.setFixedWidth(150)
        # Set groupbox layout
        self.ThumbnailLayout = QVBoxLayout()
        self.ThumbnailLayout.setContentsMargins(5, 5, 5, 5)
        self.ThumbnailGroupBox.setLayout(self.ThumbnailLayout)
        # ThumbnailImageLabel is the label which will display the thumbnail of the selected item
        # Images in PyQt are displayed by loading a QPixmap onto a QLabel
        self.ThumbnailImageLabel = QLabel()
        self.ThumbnailImageLabel.setAlignment(Qt.AlignCenter)
        self.ThumbnailLayout.addWidget(self.ThumbnailImageLabel)
        # This list stores the labels which will be used to display info about the currently
          selected item
        # A label is created for each column in the SQL table and used to display its
         corresponding contents
        self.ThumbnailLabelsArray = []
        # Get the number of columns in the SQL table
        self.MyCursor.execute("SELECT COUNT(*) FROM information_schema.columns WHERE table_name
        = '" + self.OpenCollectionTable + "'")
        # For each column in the table, create a label (excluding the Rare, Rating and Thumbnail
          columns)
        for x in range(1, self.MyCursor.fetchall()[0][0] - 3):
            Temp = QLabel("")
            Temp.setWordWrap(True)
            self.ThumbnailLayout.addWidget(Temp)
            # Because there is a variable amount of labels, they are each appended to this list
              after they are created
            self.ThumbnailLabelsArray.append(Temp)
        self.ThumbnailRatingLBL = QLabel()
        self.ThumbnailLayout.addWidget(self.ThumbnailRatingLBL)
        self.ThumbnailLayout.addStretch()
        # Add groupbox to horizontal box layout
        self.OpenCollectionHBL1.addWidget(self.ThumbnailGroupBox)

        # Add horizontal box layout to main layout
        self.OpenCollectionVBL1.addLayout(self.OpenCollectionHBL1)

    # This function is executed when the itemSelectionChanged signal is emitted
    # It performs a simple search on all columns in the SQL table based on what the user has
      inputted in the search bar
    def SimpleSearch(self):
        # Get search string from search bar
        SearchRequest = self.SearchBarTB.text()
        # SearchResult stores the rows of the table widget which contain the items that have
          fulfilled the terms of the search
        SearchResult = []
        # Presence check
        if SearchRequest == "":
            self.CollectionTable.clearSelection()
        else:
            # Clear all previous selections
            self.CollectionTable.clearSelection()
            self.CollectionTable.setSelectionMode(QAbstractItemView.MultiSelection)
            # Retrieve all column names from SQL table
            self.MyCursor.execute("SHOW COLUMNS FROM " + self.OpenCollectionTable)
            MyResult4 = self.MyCursor.fetchall()
            # Concatenate strings together to form a SQL query that will search the table for
```

```
            target items and return their primary key
        SQLStatement = 'SELECT PK_Table' + str(self.OpenCollectionID) + ' FROM ' +
        self.OpenCollectionTable + ' WHERE '
        # The function searches all columns in the table except the Thumbnail column
        for x in range(1, len(MyResult4) - 2):
            # Uses LIKE statements to find instances of the string inputted in the search
              bar in all fields of the database
            SQLStatement += MyResult4[x][0] + ' LIKE "%' + SearchRequest + '%" OR '
        SQLStatement += MyResult4[len(MyResult4) - 2][0] + ' LIKE "%' + SearchRequest + '%"'
        self.MyCursor.execute(SQLStatement)
        print(SQLStatement)
        # Retrieve results of SQL query
        MyResult5 = list(set(self.MyCursor.fetchall()))

        # For each element in the result of the SQL query, find their corresponding row in
          the table widget by comparing their primary keys
        for z in range(0, self.CollectionTable.rowCount()):
            for x in range(0, len(MyResult5)):
                if self.CollectionTable.item(z, 0).text() == str(MyResult5[x][0]):
                    # Add each tableItem object to the list
                    SearchResult.append(self.CollectionTable.item(z, 0))

        # Highlight/Select each row stored in SearchResult to relay the results of the
          search to the user
        for z in range(0, len(SearchResult)):
                self.CollectionTable.selectRow(SearchResult[z].row())
        self.CollectionTable.setSelectionMode(QAbstractItemView.ExtendedSelection)

# This function is executed when the user selects an item in CollectionTable
# It displays the thumbnail for the selected item alongside the data stored about them in
  the SQL table
# The thumbnails of items which are marked as rare get a rare tag overlayed over their
  thumbnail
def DisplayImage(self):
    # Get currently selected row
    Row = self.CollectionTable.selectionModel().currentIndex()
    # Get the primary key of that row by reading the valus stored in the first (hidden)
      column
    PKValue = Row.sibling(Row.row(), 0).data()

    if len(self.CollectionTable.selectionModel().selectedRows()) > 1:
        self.StatusLBL.setText(str(len(
                    self.CollectionTable.selectionModel().selectedRows()))
                    + " of " + str(self.RowCount) + " items")
    else:
        self.StatusLBL.setText(str(self.RowCount) + " items")

    # Presence check
    if PKValue:
        # Retrieve the SQL entry which matches the currently selected row
        self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " WHERE PK_" +
        self.OpenCollectionTable + " = " + PKValue)
        MyResult3 = self.MyCursor.fetchall()[0]
        # If a binary large object has been stored in the Thumbnail column...
        if MyResult3[len(MyResult3) - 1]:
            # Load image as QImage from raw binary data
            ThumbnailImage = QImage.fromData(MyResult3[len(MyResult3) - 1])
            # Convert QImage into QPixmap
            ThumbnailPixmap = QPixmap.fromImage(ThumbnailImage)
            # Resize image
            ThumbnailPixmap = ThumbnailPixmap.scaledToWidth(120, Qt.SmoothTransformation)
            # Check if item is tagged as Rare
            if MyResult3[len(MyResult3) - 3] == 1:
                # If image is rare, load overlay image
                OverlayPixmap = QPixmap("Resources/RareTag.png")
                # Initialise QPainter object
                Painter = QPainter(ThumbnailPixmap)
                # Draw rare tag over item thumbnail
                Painter.drawPixmap(0, 0, OverlayPixmap)
                # End paint event
                Painter.end()
        # If there is nothing in the Thumbnail column, display a generic 'No Image
          Available' image
        else:
```

```
            ThumbnailPixmap = QPixmap("Resources/NoImageAvailable.png")
            ThumbnailPixmap = ThumbnailPixmap.scaledToWidth(120, Qt.SmoothTransformation)
        # Load pixmap into ThumbnailLabel
        self.ThumbnailImageLabel.setPixmap(ThumbnailPixmap)

        # Iterate through list of thumbnail labels and set each one's text to the
          corresponding data stored in the field for that item
        for x in range(0, len(self.ThumbnailLabelsArray)):
            self.ThumbnailLabelsArray[x].setText(str(MyResult3[x + 1]))
        ThumbnailRatingPixmap = QPixmap("Resources/{0}stars.png"
        .format(str(MyResult3[len(MyResult3) - 2])))
        ThumbnailRatingPixmap = ThumbnailRatingPixmap.scaledToWidth(100,
        Qt.SmoothTransformation)
        self.ThumbnailRatingLBL.setPixmap(ThumbnailRatingPixmap)

# This function populates the PyQt table widget with data from the SQL table
def PopulateTable(self):
    # Get columns from SQL table
    self.MyCursor.execute("SHOW COLUMNS FROM " + self.OpenCollectionTable)
    MyResult1 = self.MyCursor.fetchall()
    ColumnNames = []
    ColumnCounter = 0
    # LargestArray stores the size of each column
    # As the function progresses, LargestArray finds the size of the alrgest string for each
      column and appends it
    LargestArray = []
    # Iterate through the result of the SQL query and get amount of columns as well as
      column names
    for x in range(0, len(MyResult1) - 1):
        ColumnNames.append(MyResult1[x][0])
        ColumnCounter += 1
        Temp = QLabel(MyResult1[x][0])
        Temp.setFixedWidth(QFontMetrics(Temp.font()).width(Temp.text()))
        LargestArray.append(Temp.width())
    LargestArray[len(MyResult1) - 2] = 100
    # Set column count of table widget to column count of SQL table
    self.CollectionTable.setColumnCount(ColumnCounter)
    # Set the horizontal headings of the table widget to the headings of the SQL columns
    self.CollectionTable.setHorizontalHeaderLabels(ColumnNames)
    # Hide the first column as it stores the primary key for each item
    self.CollectionTable.setColumnHidden(0, True)

    self.CollectionTable.horizontalHeader().setObjectName("hHeader")

    # Clear table of rows before repopulating table
    self.CollectionTable.setRowCount(0)
    # Get number of entries in table
    self.MyCursor.execute("SELECT COUNT(*) FROM " + self.OpenCollectionTable)
    self.RowCount = self.MyCursor.fetchall()[0][0]
    # Set number of rows in table widget to number of entries in SQL table
    self.CollectionTable.setRowCount(self.RowCount)
    # Set label in status bar to display the number of items in the collection
    self.StatusLBL.setText(str(self.RowCount) + " items")
    # Select which order the items should be displayed in based on the value in OrderCB
    if self.OrderCB.currentText() == "Ascending":
        # Order items in ascending alphabetical order based on the first column (excluding
          the primary key)
        self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " ORDER BY " +
        str(MyResult1[1][0]) + " ASC")
    elif self.OrderCB.currentText() == "Descending":
        # Order items in descending order based on the first column
        self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " ORDER BY " +
        str(MyResult1[1][0]) + " DESC")
    else:
        # Order items in the order that they have been added to the collection
        self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable)

    # Retrieve results of SQL query
    MyResult2 = self.MyCursor.fetchall()
    # Iterate through each entry in the table
    for x in range(0, len(MyResult2)):
        # Store the primary key in the first (hidden) column of the table widget
        self.CollectionTable.setItem(x, 0, QTableWidgetItem(str(MyResult2[x][0])))
        # For loop here begins at 1 because we have already stored the primary key in a
```

```
            table cell
            # len(MyResult2[x]) - 3 excludes the Rare, Rating and Thumbnail columns as they are
              displayed in special widgets
            for y in range(1, len(MyResult2[x]) - 3):
                # Create a label and set label text to text value of current field the loop is
                  on
                Temp = QLabel(str(MyResult2[x][y]))
                # Add label to corresponding cell in the table widget
                self.CollectionTable.setCellWidget(x, y, Temp)
                # If the width of the label is greater than the width currently stored in
                  L argestArray, change the value at that index to the new largest width
                if int(Temp.width()) > LargestArray[y]:
                    LargestArray[y] = int(Temp.width())

            # If the item is tagged as rare...
            if MyResult2[x][len(MyResult2[x]) - 3] == 1:
                # Create an instance of RareWidget and add to table widget
                RarityCell = RareWidget()
                self.CollectionTable.setCellWidget(x, (len(MyResult2[x]) - 3), RarityCell)

            # Create an instance of RatingWidget, passing in the rating of the current item as a
              parameter
            RatingCell = RatingWidget(Rating = MyResult2[x][len(MyResult2[x]) - 2])
            self.CollectionTable.setCellWidget(x, (len(MyResult2[x]) - 2), RatingCell)

        # Set size of each column to the corresponding width in LargestArray
        for x in range(0, len(LargestArray)):
            self.CollectionTable.setColumnWidth(x, LargestArray[x] + 40)

        self.CheckTableScrollBar(False)

        # If the sorting tree widget is not hidden, populate it
        if not self.SortingTree.isHidden():
            # 3D list containing each parent and its corresponding children
            self.SortingTreeKey = []
            # Clear the tree's contents before repopulating it
            self.SortingTree.clear()
            # Iterate through the list containing each column in the table
            for x in range (1, len(MyResult1) - 1):
                # Create a parent item for each column in the table (excluding the thumbnail and
                  primary key columns) and add them to the tree widget
                Temp1 = QTreeWidgetItem(self.SortingTree, [MyResult1[x][0]])
                # Add the parent to the list
                self.SortingTreeKey.append([Temp1])
                # SQL query which returns each entry under the specified column along with the
                amount of times it occurs in the table
                self.MyCursor.execute("SELECT `" + str(MyResult1[x][0]) + "`, COUNT(*) FROM " +
                self.OpenCollectionTable + " GROUP BY `" + str(MyResult1[x][0]) + "`")
                MyResult3 = self.MyCursor.fetchall()
                # Iterate through the results of the query, creating a child item for each entry
                  and adding it to the tree widget
                for y in range(0, len(MyResult3)):
                    # Create child item and add to tree widget, the string passed in is the name
                      with the number of time it occurs appended to the end
                    Temp2 = QTreeWidgetItem(self.SortingTreeKey[x - 1][0], [str(MyResult3[y][0])
                    + " (" + str(MyResult3[y][1]) + ")"])
                    Temp2.setData(0, Qt.UserRole, MyResult3[y][0])
                    # Add child to 3D list in the same index as its parent
                    self.SortingTreeKey[x - 1].append(Temp2)

    # This slot is executed whenever a new item is selected in SortingTree
    # It highlights all the items which correspond with the user's selection
    def SortingTreeClicked(self):
        # Get the current selected item
        SelectedItem = self.SortingTree.currentItem()
        # If the currently selected item is a parent item do not proceed
        if SelectedItem.parent() is None:
            pass
        # If the selected item is a child item, proceed
        else:
            # Get the child item's parent
            SelectedItemParent = SelectedItem.parent()
            # Select the primary key of all entries where the contents of the parent column is
```

```
            the same as the selected child
            self.MyCursor.execute("SELECT PK_" + self.OpenCollectionTable + " FROM " +
            self.OpenCollectionTable + " WHERE " + str(SelectedItemParent.text(0)) + " = '" +
            str(SelectedItem.data(0, Qt.UserRole)) + "'")
            # Reassign query result as list, not tuple
            MyResult1 = list(set(self.MyCursor.fetchall()))

            # List for storing all the rows in the frontend table which correspond with the
              results of the SQL query
            SearchResult = []
            # Clear selection
            self.CollectionTable.clearSelection()
            # Set the table's selection mode so that multiple rows can be selected at once
            self.CollectionTable.setSelectionMode(QAbstractItemView.MultiSelection)

            # Iterate through all the rows in the frontend table and add each row which
              corresponds with the SQL query results to the list
            for z in range(0, self.CollectionTable.rowCount()):
                for x in range(0, len(MyResult1)):
                    if self.CollectionTable.item(z, 0).text() == str(MyResult1[x][0]):
                        SearchResult.append(self.CollectionTable.item(z, 0))

            # Iterate through the contents of SearchResult and highlight the rows
            for z in range(0, len(SearchResult)):
                    self.CollectionTable.selectRow(SearchResult[z].row())

            # Return table's selection mode to default
            self.CollectionTable.setSelectionMode(QAbstractItemView.ExtendedSelection)

    # Method executed when ShowThumbnailAction in the View menu is pressed
    def ShowThumbnail(self):
        if self.ShowThumbnailAction.isChecked():
            self.ThumbnailGroupBox.show()
        else:
            self.ThumbnailGroupBox.hide()

    # Method executed when ShowThumbnailAction in the View menu is pressed
    def ShowSorting(self):
        if self.ShowSortingAction.isChecked():
            self.SortingGroupBox.show()
        else:
            self.SortingGroupBox.hide()

    # Initialise window toolbar
    def InitToolbar(self):
        # Toolbar buttons are displayed in a horizontal box layout
        self.ToolBarHBL = QHBoxLayout()
        self.OpenCollectionVBL1.addLayout(self.ToolBarHBL)

        # Each Toolbar button is an instance of the ToolBarBTN class
        # AddItemBTN is used for adding new items to the collection
        self.AddItemBTN = ToolBarBTN(Text = "Add Item", IconPath = "Resources/AddItemIcon.png")
        self.AddItemBTN.clicked.connect(self.AddItemMethod)
        self.ToolBarHBL.addWidget(self.AddItemBTN)

        # DeleteItemBTN deletes the selected item(s) from the collection
        self.DeleteItemBTN=ToolBarBTN(Text="DeleteItem",IconPath="Resources/DeleteItemIcon.png")
        self.DeleteItemBTN.clicked.connect(self.DeleteItemMethod)
        self.ToolBarHBL.addWidget(self.DeleteItemBTN)

        # EditItemBTN allows the user to edit the data stored about the selected item
        self.EditItemBTN=ToolBarBTN(Text = "Edit Item", IconPath = "Resources/EditItemIcon.jpg")
        self.EditItemBTN.clicked.connect(self.EditItemMethod)
        self.ToolBarHBL.addWidget(self.EditItemBTN)

        # DeleteDuplicatesBTN deletes duplicate items from the collection
        self.DeleteDuplicatesBTN=ToolBarBTN(Text="Delete Duplicates",
                IconPath="Resources/DelDuplicatesIcon.png")
        self.DeleteDuplicatesBTN.clicked.connect(self.DeleteDuplicatesMethod)
        self.ToolBarHBL.addWidget(self.DeleteDuplicatesBTN)

        # AdvancedSearchBTN opens the advanced search window for constructing complex queries
        self.AdvancedSearchBTN = ToolBarBTN(Text="Advanced Search",IconPath=
```

```
        "Resources/AdvancedSearchIcon.png")
        self.AdvancedSearchBTN.clicked.connect(self.AdvancedSearchMethod)
        self.ToolBarHBL.addWidget(self.AdvancedSearchBTN)

        # LoanItemBTN is for loaning out items
        self.LoanItemBTN=ToolBarBTN(Text = "Loan Item", IconPath ="Resources/LoanItemIcon.png")
        self.LoanItemBTN.clicked.connect(self.LoanItemMethod)
        self.ToolBarHBL.addWidget(self.LoanItemBTN)

        # ExportBTN exports the current collection
        self.ExportBTN=ToolBarBTN(Text="Export Collection",IconPath= "Resources/ExportIcon.png")
        self.ExportBTN.clicked.connect(self.ExportCollectionMethod)
        self.ToolBarHBL.addWidget(self.ExportBTN)

        # BarcodeSearchBTN opens the barcode search window which allows users to add items to
          collections using their barcode
        self.BarcodeSearchBTN = ToolBarBTN(Text="Search Barcode",IconPath=
        "Resources/BarcodeIcon.png")
        self.BarcodeSearchBTN.clicked.connect(self.BarcodeSearchMethod)
        self.ToolBarHBL.addWidget(self.BarcodeSearchBTN)

        # GenerateGraphsBTN generates and displays graphs based on the user's collection
        self.GenerateGraphsBTN = ToolBarBTN(Text = "Generate Graphs", IconPath =
        "Resources/GenerateGraphsIcon.png")
        self.GenerateGraphsBTN.clicked.connect(self.GenerateGraphsMethod)
        self.ToolBarHBL.addWidget(self.GenerateGraphsBTN)

        self.ToolBarHBL.addStretch()

    # This function is executed when the advanced search is complete
    def SearchCompleteSlot(self, SearchItems):
        # This method of highlighting search results is similar to the one used for the simple
          search, with a few minor adjustments made
        SearchResult = []
        self.CollectionTable.clearSelection()
        # Set the selection colour for the table to light red in order to differentiate search
          results from normal selection
        self.CollectionTable.setStyleSheet("QWidget::item:selected { background: #FF6961; }")
        self.CollectionTable.setSelectionMode(QAbstractItemView.MultiSelection)
        for z in range(0, self.CollectionTable.rowCount()):
            for x in range(0, len(SearchItems)):
                if self.CollectionTable.item(z, 0).text() == str(SearchItems[x][0]):
                    SearchResult.append(self.CollectionTable.item(z, 0))
        for z in range(0, len(SearchResult)):
            self.CollectionTable.selectRow(SearchResult[z].row())
        self.CollectionTable.setSelectionMode(QAbstractItemView.ExtendedSelection)
        # Connect the signal emitted when the table is clicked to the slot
        self.CollectionTable.clicked.connect(self.CollectionTableClickedSlot)
        # Show message to user in status bar
        self.StatusBar.showMessage("Search results are highlighted in red", 3000)
        self.CheckTableScrollBar(AdvancedSearch = True)

    # This slot is executed when the table is clicked after an advanced search is completed
    # It sets the selection colour in the table back to pastel green instead of red
    def CollectionTableClickedSlot(self):
        self.CollectionTable.setStyleSheet("QWidget::item:selected {background: #D5E8D4}")
        self.CollectionTable.clicked.disconnect()

    def CheckTableScrollBar(self, AdvancedSearch):
        if (self.CollectionTable.rowHeight(0) * self.RowCount) > self.CollectionTable.height():
            if AdvancedSearch:
                self.CollectionTable.setStyleSheet("QTableWidget { border-right: 0px; }
QWidget::item:selected { background: #FF6961; }")
            else:
                self.CollectionTable.setStyleSheet("QTableWidget { border-right: 0px; }
QWidget::item:selected { background: #D5E8D4; }")
        else:
            if AdvancedSearch:
                self.CollectionTable.setStyleSheet("QTableWidget { border-right: 1px solid
rgb(90,90,90); } QWidget::item:selected { background: #FF6961; }")
            else:
                self.CollectionTable.setStyleSheet("QTableWidget { border-right: 1px solid
rgb(90,90,90); } QWidget::item:selected { background: #D5E8D4; }")
```

```
    def BarcodeSearchMethod(self):
        self.BarcodeSearchInstance = BarcodeSearch.BarcodeSearch(widget = self.BarcodeSearchBTN)

    # This function is executed when the user selects Close Collection from the file menu
    # It closes the current window and reopens the collection window
    def CloseCollection(self):
        self.MainMenu.clear()
        self.CollectionWindowInstance = CollectionWindow.CollectionWindow(MyCursor =
        self.MyCursor, MyDB = self.MyDB, ActiveUserID = self.Yes)
        self.close()

    # This method is executed when the user presses GenerateGraphsBTN
    def GenerateGraphsMethod(self):
        self.GraphWindowInstance = GraphWindow.GraphWidget(MyCursor=self.MyCursor,
        MyDB=self.MyDB, ActiveUserID=self.ActiveUserID, OpenCollectionID=self.OpenCollectionID,
        OpenCollectionTable=self.OpenCollectionTable, widget = self.GenerateGraphsBTN)

    # This method is executed when the user presses AddItemBTN
    def AddItemMethod(self):
        self.AddItemWindowInstance = AddItemWindow.AddItemWindow(MyCursor=self.MyCursor,
        MyDB=self.MyDB, ActiveUserID=self.ActiveUserID, OpenCollectionID=self.OpenCollectionID,
        OpenCollectionTable=self.OpenCollectionTable, widget = self.AddItemBTN)
        self.AddItemWindowInstance.ItemAddedSignal.connect(self.ItemAddedSlot)

    # This method is executed when the user presses LoanItemBTN
    def LoanItemMethod(self):
        self.LoanItemWindowInstance = LoanItem.LoanItemWindow(MyCursor=self.MyCursor,
        MyDB=self.MyDB, ActiveUserID=self.ActiveUserID,OpenCollectionID=self.OpenCollectionID,
        OpenCollectionTable=self.OpenCollectionTable, widget = self.LoanItemBTN)

    # This method is executed when the user presses AdvancedSearchBTN
    def AdvancedSearchMethod(self):
        self.AdvancedSearchInstance = AdvancedSearch.AdvancedSearch(MyCursor=self.MyCursor,
        MyDB=self.MyDB, ActiveUserID=self.ActiveUserID,OpenCollectionID=self.OpenCollectionID,
        OpenCollectionTable=self.OpenCollectionTable, widget = self.AdvancedSearchBTN)
        self.AdvancedSearchInstance.SearchCompleteSignal.connect(self.SearchCompleteSlot)

    # This method is executed when the user presses ExportCollectionBTN
    def ExportCollectionMethod(self):
        self.ExportSubMenuInstance = ExportCollection.ExportSubMenu(MyCursor=self.MyCursor,
        MyDB=self.MyDB, ActiveUserID=self.ActiveUserID,OpenCollectionID=self.OpenCollectionID,
        OpenCollectionTable=self.OpenCollectionTable, widget = self.ExportBTN)
        self.ExportSubMenuInstance.Success.connect(lambda: self.StatusBar.showMessage("Your
        collection was successfully exported!", 3000))
        self.ExportSubMenuInstance.Failed.connect(lambda: self.StatusBar.showMessage("There was
        an issue exporting your collection, please try again", 3000))

    # This function is executed when the user presses EditItemBTN
    # It retrieves the selected row from CollectionTable and passes it into a new instance of
      EditItemWindow as an argument
    def EditItemMethod(self):
        # Stores the selected row
        SelectedRowsArray = []
        # Stores the id corresponding to each selected row
        SelectedRowsIDArray = []
        # Iterate through selected rows in CollectionTable and append them to the list
        for x in self.CollectionTable.selectionModel().selectedRows():
            SelectedRowsArray.append(x.row())
        # Iterate through the contents of SelectedRowsArray and retrieve the corresponding
          primary key by getting the contents of the first (hidden) column
        for y in range(0, len(SelectedRowsArray)):
            SelectedRowsIDArray.append(self.CollectionTable.item(SelectedRowsArray[y],0).text())
        # If the user hasn't selected an item don't continue
        if len(SelectedRowsIDArray) == 0:
            pass
        else:
            # Create EditItemWindow instance
            # EditItemWindow takes the same parameters of AddItemWindow as well as
              ItemsForEditing which in this case is the first value in SelectedRowsIDArray
            self.EditItemWindowInstance = EditItemWindow.EditItemWindow(MyCursor=self.MyCursor,
            MyDB=self.MyDB, ActiveUserID=self.ActiveUserID,OpenCollectionID=OpenCollectionID,
            OpenCollectionTable=self.OpenCollectionTable, widget = self.EditItemBTN,
            ItemsForEditing = SelectedRowsIDArray[0])
            # Connect signal to slot
```

```
        self.EditItemWindowInstance.ItemAddedSignal.connect(self.ItemAddedSlot)

    # This function is executed when the signal ItemAddedSignal is emitted
    def ItemAddedSlot(self, Message):
        # Message is a boolean value passed into ItemAddedSignal
        # The value of Message is passed to this slot when the signal is emitted
        # Message is set to False if the window is an edit item window, so the program shows the
          according message in the status bar
        if not Message:
            self.StatusBar.showMessage("Item has been edited", 3000)
        # Message is set to True if the window is an add item window
        else:
            self.StatusBar.showMessage("New item has been added to collection", 3000)

        self.UpdateSizes()

        # Refresh front end table
        self.PopulateTable()

    # This function is called when DeleteItemBTN is pressed
    # It deletes the selected items in CollectionTable from the database
    def DeleteItemMethod(self):
        # List stores all the rows in CollectionTable the user has selected
        SelectedRowsArray = []
        # List will store the primary keys of the items the user has selected
        SelectedRowsIDArray = []

        # Iterate through all the selected rows and add them to the list
        for x in self.CollectionTable.selectionModel().selectedRows():
            SelectedRowsArray.append(x.row())

        # Iterate through the array of selected rows and add their primary key to the second
          list
        for y in range(0, len(SelectedRowsArray)):
            # The primary key of each item is stored in the first column, which is hidden
            SelectedRowsIDArray.append(self.CollectionTable.item(SelectedRowsArray[y],0).text())
        # If the user hasn't selected any items before they pressed the button, their is no
          point in proceeding
        if len(SelectedRowsIDArray) == 0:
            pass
        else:
            # Data has to be passed into a SQL command as a tuple
            SelectedRowsIDTuple = tuple(SelectedRowsIDArray)
            # Delete each item in the tuple from the database
            for z in range(0, len(SelectedRowsIDArray)):
                self.MyCursor.execute("DELETE FROM " + self.OpenCollectionTable + " WHERE PK_" +
                self.OpenCollectionTable + " = %s" % (SelectedRowsIDTuple[z]))
            # Save changes to database
            self.MyDB.commit()

            self.UpdateSizes()

            # Refresh table to get rid of deleted items
            self.PopulateTable()
            # Display temporary message to user informing them that operation was successful
            self.StatusBar.showMessage("Selected items have been deleted", 3000)

    # This function is executed when DeleteDuplicatesBTN is pressed
    # It deletes all duplicate items from the database
    def DeleteDuplicatesMethod(self):
        # Get columns from open collection table
        self.MyCursor.execute("SHOW COLUMNS FROM " + self.OpenCollectionTable)
        MyResult6 = self.MyCursor.fetchall()

        # Construct a SQL command which will select all the duplicate entries in the table using
          the COUNT(*) command
        SQLStatement = "SELECT "
        for x in range(1, len(MyResult6) - 4):
            SQLStatement += "`" + MyResult6[x][0] + "`, "
        SQLStatement += MyResult6[len(MyResult6) - 4][0] + " FROM " + self.OpenCollectionTable +
        " GROUP BY "
        for x in range(1, len(MyResult6) - 4):
            SQLStatement += "`" + MyResult6[x][0] + "`, "
        SQLStatement += MyResult6[len(MyResult6) - 4][0] + " HAVING COUNT(*) > 1"
```

```
        self.MyCursor.execute(SQLStatement)
        MyResult7 = self.MyCursor.fetchall()

        # Iterate through duplicate entries
        for x in range(0, len(MyResult7)):
            # Construct a SQL command which returns the primary key of each duplicate record
            SQLStatement2 = "SELECT PK_" + self.OpenCollectionTable + " FROM " +
            self.OpenCollectionTable + " WHERE "
            # For loop starts at 1 because the first instance of the duplicate entry isn't
              deleted
            for y in range(1, len(MyResult6) - 4):
                SQLStatement2 += "`" + str(MyResult6[y][0]) + "` = '" + str(MyResult7[x][y - 1])
                + "' AND "
            SQLStatement2 += "`" + str(MyResult6[len(MyResult6) - 4][0]) + "` = '" +
            str(MyResult7[x][len(MyResult7[x]) - 1]) + "' ORDER BY PK_" +
            self.OpenCollectionTable + " DESC"
            self.MyCursor.execute(SQLStatement2)
            # MyResult8 stores all the duplicate entry primary keys
            MyResult8 = self.MyCursor.fetchall()

            # For each duplicate record (there may be more than one duplicate) delete it from
              the SQL table
            for z in range(0, len(MyResult8) - 1):
                self.MyCursor.execute("DELETE FROM " + self.OpenCollectionTable + " WHERE PK_" +
                self.OpenCollectionTable + " = " + str(MyResult8[z][0]))
                # Save changes to database
                self.MyDB.commit()

        self.UpdateSizes()

        # Refresh table to get rid of deleted items
        self.PopulateTable()
        # Display temporary message to user informing them that operation was successful
        self.StatusBar.showMessage("Duplicate items have been deleted", 3000)

    # Executed when the size of the collection changes (i.e. when items are added or deleted)
    def UpdateSizes(self):
        # Get the size of the collection
        self.MyCursor.execute("SELECT COUNT(*) FROM " + self.OpenCollectionTable)
        MyResult9 = self.MyCursor.fetchall()
        # Store the size in the database alongside the date that size was recorded at
        self.MyCursor.execute("INSERT INTO Sizes (TimeRecorded, Magnitude, FK_Collections_Sizes)
        VALUES ('" + str(datetime.now()) + "', " + str(MyResult9[0][0]) + ", " +
        str(self.OpenCollectionID) + ")")
        # Save changes to database
        self.MyDB.commit()


# Inherits from QToolButton
# Used to create toolbar buttons in InitToolbar()
class ToolBarBTN(QToolButton):
    # Takes the arguments Text and IconPath
    # IconPath is the file path for the icon the button displays
    def __init__(self, Text, IconPath):
        super(ToolBarBTN, self).__init__()
        self.setText(Text)
        self.setIcon(QIcon(IconPath))
        self.setIconSize(QSize(50, 50))
        self.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)
        self.setFixedWidth(90)

# RareWidget is a styled QLabel enclosed within a QWidget
# It is displayed in the table widget for any item that is tagged as rare
class RareWidget(QWidget):
    def __init__(self):
        super(RareWidget, self).__init__()
        RarityLBL = QLabel("Rare")
        RarityLBL.setStyleSheet("background: #7C5295; color: white; font-weight: bold; font
        size: 12px; border-radius: 5px;")
        RarityLBL.setAlignment(Qt.AlignCenter)
        RarityLBL.setFixedSize(40, 20)
        self.setObjectName("BorderlessWidget")
        self.setStyleSheet("background-color: rgba(0,0,0,0%)")
        RarityLayout = QHBoxLayout(self)
```

```
        RarityLayout.addWidget(RarityLBL)
        RarityLayout.setAlignment(Qt.AlignVCenter)
        RarityLayout.setContentsMargins(0, 0, 0, 0)

# RatingWidget displays an image of stars out of five based on the value stored in the Rating column
class RatingWidget(QWidget):
    # Takes the argument Rating, this is just an integer out of five
    def __init__(self, Rating):
        super(RatingWidget, self).__init__()
        RatingLBL = QLabel()
        RatingPixmap = QPixmap()
        # Format value of Rating into file path to get correct picture
        RatingPixmap.load("Resources/{0} stars.png".format(Rating))
        RatingLBL.setPixmap(RatingPixmap.scaledToHeight(20))
        RatingLBL.setAlignment(Qt.AlignCenter)
        self.setObjectName("BorderlessWidget")
        self.setStyleSheet("background-color: rgba(0,0,0,0%)")
        RatingLayout = QHBoxLayout(self)
        RatingLayout.addWidget(RatingLBL)
        RatingLayout.setAlignment(Qt.AlignVCenter)
        RatingLayout.setContentsMargins(0, 0, 0, 0)
```

AddItemWindow.py

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
import os

# This class is the window which users add items to the collection from
# It inherits from QWidget rather than QMainWindow because I need to access its painEvent
class AddItemWindow(QWidget):
    # This signal is emitted when the user has successfully added an item to the collection
    # When the OpenCollection object receives this signal, it will execute the function
      PopulateTable()
    ItemAddedSignal = pyqtSignal(bool)
    # Takes the same arguments as OpenCollectionWindow except for widget which is the toolbar
      button which is pressed to open the window
    # Widget is an argument because it is used to determine the position of this new window on
      the screen
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID, OpenCollectionTable,
    widget = None):
        super(AddItemWindow, self).__init__()

        # Reassign arguments as attributes
        self.ActiveUserID = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.OpenCollectionID = OpenCollectionID
        self.OpenCollectionTable = OpenCollectionTable
        # Boolean for keeping track of if the user has uploaded a thumbnail or not
        self.ContainsImage = False

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Initialise main window layout
        self.AddItemVBL1 = QVBoxLayout()
        self.setLayout(self.AddItemVBL1)

        self.InitUI()

        # Set this window so that it closes when its parent window is closed
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setAttribute(Qt.WA_TranslucentBackground)
        # Modal windows force the user to interact with it before they can go back to using the
          parent application
        self.setWindowModality(Qt.ApplicationModal)
        # Get rid of window frame and ensure this window always stays on top of other applicaton
```

```
        windows
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint | Qt.WindowStaysOnTopHint))

        # Determine where window should be positioned based on the widget which is passed in as
          an argument
        x = widget.mapToGlobal(QPoint(0, 0)).x()
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        # The window should appear on the screen just below the AddItemBTN which is clicked to
          create it
        self.move(x, y)

        # Show window
        self.show()

    # Initialise and display window widgets
    def InitUI(self):
        # Header label informs users exactly what the window's purpose is
        self.HeaderLBL = QLabel()
        self.HeaderLBL.setText("Add Item")
        self.HeaderLBL.setStyleSheet("font-size: 16px;")
        self.HeaderLBL.setAlignment(Qt.AlignCenter)
        self.AddItemVBL1.addWidget(self.HeaderLBL)

        # The widgets which users use to input data about the item are presented in a scroll
          area
        self.AddItemScrollArea = QScrollArea()
        self.AddItemScrollArea.verticalScrollBar().setStyleSheet("QScrollBar { border-right:
        0px; border-top: 0px; border-bottom: 0px; } QScrollBar::handle { border-top: 0px;
        border-bottom: 0px; }")
        self.AddItemVBL1.addWidget(self.AddItemScrollArea)
        self.AddItemScrollArea.setWidgetResizable(True)
        # This is the widget the scroll area displays
        self.ScrollAreaWidget = QWidget()
        self.ScrollAreaWidget.setObjectName("BorderlessWidget")
        # This is the grid layout of the widget in the scroll area
        self.ScrollAreaLayout = QGridLayout()
        self.ScrollAreaLayout.setAlignment(Qt.AlignTop)
        self.ScrollAreaWidget.setLayout(self.ScrollAreaLayout)
        self.AddItemScrollArea.setWidget(self.ScrollAreaWidget)

        # Two buttons in this Horizontal layout, one for adding the new item to the collection,
          and one for cancelling the operation
        ActionBTNsHBL = QHBoxLayout()
        self.CancelBTN = QPushButton("Cancel")
        self.CancelBTN.clicked.connect(self.close)
        self.SubmitBTN = QPushButton("Submit")
        self.SubmitBTN.clicked.connect(self.AddItem)
        ActionBTNsHBL.addWidget(self.CancelBTN)
        ActionBTNsHBL.addWidget(self.SubmitBTN)
        self.AddItemVBL1.addLayout(ActionBTNsHBL)

        self.PopulateScrollArea()

    # This function populates the scroll area with input fields for entering the data about the
      new item
    def PopulateScrollArea(self):
        # Get all column names and their datatypes from the current collectin's table
        self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
        MyResult1 = self.MyCursor.fetchall()
        # Because there will be a variable number of input boxes, they have to be stored in a
          list for future access
        self.FieldTextBoxes = []

        # Iterate through over each column, creating a new input box for each one (excluding
          primary key, rare, rating and thumbnail columns)
        for x in range(1, len(MyResult1) - 3):
            # Add a label to the grid layout displaying the column name
            self.ScrollAreaLayout.addWidget(QLabel(MyResult1[x][0]), x, 0)
            # Create a text input box for the current column
            Temp = QLineEdit()
```

```
            Temp.setFixedWidth(250)
            # If the datatype of the current column is integer, apply a QIntValidator to that
              text box
            if MyResult1[x][1] == b'int':
                # QValidators limit the input into a text box, in this case it limits the input
                  to only accepts integers
                Temp.setValidator(QIntValidator())
            # Add text box to grid layout
            self.ScrollAreaLayout.addWidget(Temp, x, 1)
            # Add text box to list
            self.FieldTextBoxes.append(Temp)

        # Add Rare label
        self.ScrollAreaLayout.addWidget(QLabel("Rare"), len(MyResult1) - 2, 0)
        # The user marks an item as rare by checking a check box
        self.RareCheckBox = QCheckBox()
        self.RareCheckBox.setFixedWidth(15)
        # Add check box to grid layout
        self.ScrollAreaLayout.addWidget(self.RareCheckBox, len(MyResult1) - 2, 1)

        # The user uses a slider to select a value from one to five for the item's rating
        self.RatingSlider = QSlider()
        self.RatingSlider.setOrientation(Qt.Horizontal)
        self.RatingSlider.setTickInterval(1)
        self.RatingSlider.setMinimum(1)
        self.RatingSlider.setMaximum(5)
        # When the value of the slider is changed, the signal valueChanged is emitted
        # This signal is triggers the function ChangeRatingImage, which changes the pixmap in
          RatingImageLBL to display the amount of star corresponding with value of the slider
        self.RatingSlider.valueChanged.connect(self.ChangeRatingImage)
        self.ScrollAreaLayout.addWidget(self.RatingSlider, len(MyResult1) - 1, 0)
        # This is the label which displays the image of the rating the user has selected
        self.RatingImageLBL = QLabel()
        self.ChangeRatingImage()
        self.ScrollAreaLayout.addWidget(self.RatingImageLBL, len(MyResult1) - 1, 1)

        # The user presses this button to open their system's file explorer
        self.UploadImageBTN = QPushButton("Add Thumbnail")
        self.UploadImageBTN.clicked.connect(self.UploadImage)
        self.UploadImageBTN.setFixedWidth(100)
        self.ScrollAreaLayout.addWidget(self.UploadImageBTN, len(MyResult1), 0)
        # If the user chooses a thumbnail image, this label will be changed to show the name of
          the currently selected image
        self.UploadedImageLBL = QLabel("No Image Selected")
        self.UploadedImageLBL.setFixedWidth(250)
        self.ScrollAreaLayout.addWidget(self.UploadedImageLBL, len(MyResult1), 1)

    # This function is executed when the valueChange signal is emitted
    # It displays an image of 1 to 5 stars, dependent on the value of RatingSlider
    def ChangeRatingImage(self):
        # Get value of slider
        SliderValue = self.RatingSlider.value()
        # Set label pixmap to corresponding image
        self.RatingImageLBL.setPixmap(QPixmap("Resources/" + str(SliderValue) + "
        stars.png").scaledToHeight(20, Qt.SmoothTransformation))

    # This function is executed when the user presses UploadImageBTN
    def UploadImage(self):
        # Open file dialog
        # "Image files (*.jpg *.png) limits user selection to only Jpeg or png files
        # This function returns the file path of the selected image
        self.FileName, _ = QFileDialog.getOpenFileName(self, "Open Image File",
        os.path.abspath(os.sep), "Image files (*.jpg *.png)")
        # If the user selects a file...
        if self.FileName:
            # Change the label text to the name of the selected file
            self.UploadedImageLBL.setText("selected " + os.path.basename(self.FileName))
            # Set boolean value created in __init__ method to true
            self.ContainsImage = True
            # Convert selected image into raw binary data and store under variable
```

```
            self.BinaryData
        with open(self.FileName, "rb") as file:
            self.BinaryData = file.read()

# This function performs a presence check on the input fields before the item is added to
  the collection
def PresenceCheck(self):
    # First perform presence check to ensure that no input fields have been left empty
    EmptyTextBoxes = []
    FilledTextBoxes = []
    Error = False
    # Iterate over each text box and check if they are empty
    for x in range(0, len(self.FieldTextBoxes)):
        if self.FieldTextBoxes[x].text() == "":
            # Append all empty text boxes to this list
            EmptyTextBoxes.append(self.FieldTextBoxes[x])
            # Boolean indicates whether the program has encountered an empty text box yet
            Error = True
        else:
            # Append all filled text boxes to this list
            FilledTextBoxes.append(self.FieldTextBoxes[x])
    # If there are any empty text boxes, run this error handling function
    if Error:
        self.AddItemError(EmptyTextBoxes, FilledTextBoxes)
    else:
        return True

# This function is executed when the user presses SubmitBTN
# It collects all the user inputs and formats them into a SQL statement which it then
  executes
def AddItem(self):
    if self.PresenceCheck():
        # SQLStatement is a dynamically generated string which stores the SQL command
        SQLStatement = "INSERT INTO " + self.OpenCollectionTable + " ("
        # Get column names from database
        self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
        MyResult2 = self.MyCursor.fetchall()
        # Add each column name to the SQL statement
        # The loop starts from 1 because the primary key column should automatically
          increment itself
        for x in range(1, len(MyResult2) - 3):
            SQLStatement += "`" + MyResult2[x][0] + "`, "
        # If the user has uploaded a thumbnail, add the Thumbnail column to the list of
          columns in which we are inserting data
        if self.ContainsImage:
            SQLStatement += "Rare, Rating, Thumbnail) VALUES ("
            Count = len(MyResult2) - 1
        # If the user hasn't uploaded a thumbnail, there is no data to insert into Thumbnail
          column, so it is left out of the command
        else:
            SQLStatement += "Rare, Rating) VALUES ("
            Count = len(MyResult2) - 2
        # %s operator is used to substitute values from DataForInsertion tuple into
          statement when it is executed
        for x in range(1, Count):
            SQLStatement += "%s, "
        SQLStatement += "%s)"

        # DataForInsertion is the tuple which stores all the data we have retrieved from the
          user input fields
        DataForInsertion = ()
        # Get the contents of each text box in the window and add it to the tuple
        for x in range(0, len(self.FieldTextBoxes)):
            DataForInsertion += (self.FieldTextBoxes[x].text(), )
        # If user has checked the rare check box, set boolean value of Rare column to TRUE
        if self.RareCheckBox.isChecked():
            DataForInsertion += (True, )
        # Otherwise, the boolean value will be FALSE
        else:
            DataForInsertion += (False, )
```

```
            # Add the value of the slider for the Rating column
            DataForInsertion += (self.RatingSlider.value(), )
            # If the user has uploaded an image, add the binary data to the tuple
            if self.ContainsImage:
                DataForInsertion += (self.BinaryData, )
            # Execute the SQL command, substituting the contents of the tuple in for each of the
              %s operators
            self.MyCursor.execute(SQLStatement, DataForInsertion)
            # Save changes to database
            self.MyDB.commit()
            # Emit signal telling root window to repopulate CollectionTable
            self.ItemAddedSignal.emit(True)
            # Close AddItemWindow
            self.close()

    # This function is executed when the user submits an item but has left one of the input
      fields blank
    # It highlights each empty text box with a red border
    # It takes two parameters: a list of empty text boxes and a list of filled text boxes
    def AddItemError(self, OffendingWidgets, NonOffendingWidgets):
        # Iterate through list of empty text boxes and set their border to red
        for TextBox in OffendingWidgets:
            TextBox.setStyleSheet("border: 1px solid #FF0000")
        # Iterate through list of filled text boxes and set their border to black
        for TextBox in NonOffendingWidgets:
            TextBox.setStyleSheet("border: 1px solid black")

    # Inbuilt PyQt function
    # sizeHint is the preferred size of the widget
    # It has to be set so that the painter event can paint the outline image according to these
      dimensions
    def sizeHint(self):
        return QSize(450, 365)

    # This event runs when the window is first initialized
    # It draws the outline/background of the window as the image "AddItemOutline.png"
    # This allows me to create irregularly shaped windows
    def paintEvent(self, event):
        # Create painter
        Painter = QPainter()
        # Start painter, set painter to perform on AddItemWindow (self)
        Painter.begin(self)
        # Create pixmap to act as guide for painter
        Outline = QPixmap()
        Outline.load('Resources/AddItemOutline.png')
        # Paint window as Outline pixmap
        Painter.drawPixmap(QPoint(0, 0), Outline)
        # End paint event
        Painter.end()
```

EditItemWindow.py

```
import AddItemWindow

# This object inherits from the AddItemWindow class I programmed in Iteration 7
# This subclass takes the new argument ItemsForEditing which is the primary key of the item the
  user has selected in CollectionTable
# Its appearance is the same as the add item window, but each input field is populated with the
  item's data, for the user to edit
class EditItemWindow(AddItemWindow.AddItemWindow):
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID, OpenCollectionTable,
    ItemsForEditing, widget = None):
        # super calls the __init__ method of the parent object
        super(EditItemWindow, self).__init__(MyCursor = MyCursor, ActiveUserID = ActiveUserID,
        MyDB = MyDB, OpenCollectionID = OpenCollectionID, OpenCollectionTable =
        OpenCollectionTable, widget = widget)

        # Reassign arguments as attributes
```

```
        self.ItemsForEditing = ItemsForEditing

        self.PopulateForEditing()
        # In the init method of the parent object, SubmitBTN was connected to the function
          AddItem()
        # For editing items, this button has to be disconnected from AddItem() and connected to
          EditItem()
        self.SubmitBTN.clicked.disconnect()
        self.SubmitBTN.clicked.connect(self.EditItem)
        # Change window header
        self.HeaderLBL.setText("Edit Item")

    # This function fills input fields with the selected item's data
    # The user can then edit the item's data before pressing submit to save their changes to the
      database
    def PopulateForEditing(self):
        # Get data from db for selected item using its primary key
        self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " WHERE PK_" +
        self.OpenCollectionTable + " = " + self.ItemsForEditing)
        MyResult3 = self.MyCursor.fetchall()[0]
        # Iterate through each field (excluding the primary key, Rating, Rare and Thumbnail
          columns) and set the contents of each text input to the field's contents
        for x in range(1, len(MyResult3) - 3):
            self.FieldTextBoxes[x - 1].setText(str(MyResult3[x]))
        # Tick the rare checkbox if the item is rare
        if MyResult3[len(MyResult3) - 3]:
            self.RareCheckBox.setChecked(True)
        # Set the rating slider's value to the integer stored in the Rating column
        self.RatingSlider.setValue(MyResult3[len(MyResult3) - 2])
        # Change text of label showing which image has been selected
        # This object uses the UploadImage method of the parent object
        self.UploadedImageLBL.setText("Change Image")

    # This method is called when the user presses SubmitBTN
    # It fetches the editted data for the item and updates the corresponding entry in the SQL DB
      to save the changes
    def EditItem(self):
        # Proceed if the presence check returns TRUE
        if self.PresenceCheck():
            # Get column names
            self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
            MyResult4 = self.MyCursor.fetchall()
            # Construct UPDATE statement
            SQLStatement = "UPDATE " + self.OpenCollectionTable + " SET "
            # Loop through list of column names and add to SQL statement
            for x in range(1, len(MyResult4) - 3):
                # %s operator is used for string substitution
                SQLStatement += "`" + MyResult4[x][0] + "` = %s, "
            SQLStatement += "Rare = %s, Rating = %s"
            # If the image has been changed, add that to the UPDATE statement
            if self.ContainsImage:
                SQLStatement += ", Thumbnail = %s"
            # Limit SQL operation to selected item using its primary key
            SQLStatement+=" WHERE PK_" + self.OpenCollectionTable + " = " +self.ItemsForEditing

            # DataForInsertion is the tuple which stores all the data retrieved from the user
              input fields
            DataForInsertion = ()
            # Get the contents of each text box in the window and add it to the tuple
            for x in range(0, len(self.FieldTextBoxes)):
                DataForInsertion += (self.FieldTextBoxes[x].text(), )
            # If user has checked the rare check box, set boolean value of Rare column to TRUE
            if self.RareCheckBox.isChecked():
                DataForInsertion += (True, )
            # Otherwise, the boolean value will be FALSE
            else:
                DataForInsertion += (False, )
            # Add the value of the slider for the Rating column
            DataForInsertion += (self.RatingSlider.value(), )
            # If the user has uploaded an image, add the binary data to the tuple
```

```
            if self.ContainsImage:
                DataForInsertion += (self.BinaryData, )

            # Execute the SQL command, substituting the contents of the tuple in for each of the
              %s operators
            self.MyCursor.execute(SQLStatement, DataForInsertion)
            # Save changes to database
            self.MyDB.commit()
            # Emit signal telling root window to repopulate CollectionTable
            self.ItemAddedSignal.emit(False)
            # Close EditItemWindow
            self.close()
```

AdvancedSearch.py

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

# This is the object which contains the advanced search window
# It inherits from QWidget and takes the same arguments as AddItemWindow
class AdvancedSearch(QWidget):
    # This signal is emitted when a successful search has been carried out
    SearchCompleteSignal = pyqtSignal(list)
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID, OpenCollectionTable,
    widget = None):
        super(AdvancedSearch, self).__init__()

        # Set arguments to attributes
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.ActiveUserID = ActiveUserID
        self.OpenCollectionID = OpenCollectionID
        self.OpenCollectionTable = OpenCollectionTable

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Initialise main layout
        self.AdvancedSearchVBL1 = QVBoxLayout()
        self.setLayout(self.AdvancedSearchVBL1)

        # Get columns from SQL table
        self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
        MyResult1 = self.MyCursor.fetchall()
        # ConditionList stores the column names
        self.ConditionList = []
        # ConditionDataTypeList stores each columns data type
        self.ConditionDataTypeList = []
        for x in range(1, len(MyResult1) - 1):
            self.ConditionList.append(MyResult1[x][0])
            self.ConditionDataTypeList.append(MyResult1[x][1])
        # Remove the rare column from the list of conditions
        self.ConditionList.remove("Rare")
        # Remove its data type from the list of datat types
        self.ConditionDataTypeList.remove(b'tinyint(1)')

        # This list stored each instance of ConditionWidget for future reference
        self.ConditionWidgetsList = []

        self.InitUI()

        # Set the window to close when the parent window closes
        self.setAttribute(Qt.WA_QuitOnClose, False)
        # Give the window a transparent background so the outline image can be used as the
          window background
        self.setAttribute(Qt.WA_TranslucentBackground)
```

```
        # Make window modal
        self.setWindowModality(Qt.ApplicationModal)
        # Make window frameless and keep it on top of all other windows
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint | Qt.WindowStaysOnTopHint))

        # Determine where window should be positioned based on the widget which is passed in as
          an argument
        x = widget.mapToGlobal(QPoint(0, 0)).x()
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        # The window should appear on the screen just below the AddItemBTN which is clicked to
          create it
        self.move(x, y)
        # Show window
        self.show()

    # This method initialises widgets and arranges them in layouts
    def InitUI(self):
        # Header label informs the user of the purpose of the window
        HeaderLBL = QLabel("Advanced Search")
        HeaderLBL.setStyleSheet("font-size: 16px;")
        HeaderLBL.setAlignment(Qt.AlignCenter)
        self.AdvancedSearchVBL1.addWidget(HeaderLBL)

        # Users will add conditions to this scroll area
        self.AdvancedSearchScrollArea = QScrollArea()
        # Add scroll area to main layout
        self.AdvancedSearchVBL1.addWidget(self.AdvancedSearchScrollArea)
        self.AdvancedSearchScrollArea.setWidgetResizable(True)
        # Widget which displays within the scroll area
        self.ScrollAreaWidget = QWidget()
        self.ScrollAreaWidget.setObjectName("BorderlessWidget")
        # Initialise layout of ScrollAreaWidget
        self.ScrollAreaLayout = QVBoxLayout()
        self.ScrollAreaLayout.setAlignment(Qt.AlignTop)
        self.OuterLayout = QVBoxLayout()
        self.OuterLayout.addLayout(self.ScrollAreaLayout)
        self.ScrollAreaWidget.setLayout(self.OuterLayout)
        self.AdvancedSearchScrollArea.setWidget(self.ScrollAreaWidget)

        self.CheckBoxLayouts = QHBoxLayout()
        self.OuterLayout.addLayout(self.CheckBoxLayouts)
        # This combobox allows users to filter query results based on their rarity
        self.CheckBoxLayouts.addWidget(QLabel("Rare:"))
        self.RareCB = QComboBox()
        self.RareCB.addItems(["Both", "True", "False"])
        self.RareCB.setFixedWidth(80)
        self.CheckBoxLayouts.addWidget(self.RareCB)
        self.CheckBoxLayouts.addSpacing(50)
        self.CheckBoxLayouts.addWidget(QLabel("Thumbnail:"))
        # This combobox allows users to filter query results based on whether they have a
          thumbnail or not
        self.ThumbnailCB = QComboBox()
        self.ThumbnailCB.addItems(["Both", "True", "False"])
        self.ThumbnailCB.setFixedWidth(80)
        self.CheckBoxLayouts.addWidget(self.ThumbnailCB)
        self.CheckBoxLayouts.addStretch()

        # This label displays when the advanced search returns no results
        self.ErrorLBL = QLabel("Your search returned no results")
        self.ErrorLBL.setAlignment(Qt.AlignCenter)
        self.ErrorLBL.setStyleSheet("color: red; ")
        self.AdvancedSearchVBL1.addWidget(self.ErrorLBL)
        # The label is initially hidden
        self.ErrorLBL.hide()

        # This is the button the user presses when they want to add a new condition
        self.AddConditionBTN = QPushButton("Add New Condition")
        # Bind button to function
        self.AddConditionBTN.clicked.connect(self.AddCondition)
        self.AdvancedSearchVBL1.addWidget(self.AddConditionBTN)
```

```
        ActionBTNsHBL = QHBoxLayout()
        # CancelBTN closes the advanced search window when it is clicked
        self.CancelBTN = QPushButton("Cancel")
        self.CancelBTN.clicked.connect(self.close)
        # SubmitBTN carries out the advanced search
        self.SubmitBTN = QPushButton("Submit")
        self.SubmitBTN.clicked.connect(self.ConductSearch)
        ActionBTNsHBL.addWidget(self.CancelBTN)
        ActionBTNsHBL.addWidget(self.SubmitBTN)
        self.AdvancedSearchVBL1.addLayout(ActionBTNsHBL)

        # Create an initial condition widget
        # The argument First is set to True for this instance to inform the object that this is
          the first condition widget in the window
        ConditionWidgetInstance = ConditionWidget(Conditions = self.ConditionList, DataTypes =
        self.ConditionDataTypeList, First = True)
        # Add widget to list
        self.ConditionWidgetsList.append(ConditionWidgetInstance)
        # Add widget to scroll area
        self.ScrollAreaLayout.addWidget(self.ConditionWidgetsList[0])

    # This method is executed when the user clicks AddConditionBTN
    def AddCondition(self):
        # Create an instance of ConditionWidget
        # Set arguments Conditions and DataTypes to the lists of column names and data types
          created in the constructor
        ConditionWidgetInstance = ConditionWidget(Conditions = self.ConditionList, DataTypes =
        self.ConditionDataTypeList, First = False)

        # Add instance to list for future reference
        self.ConditionWidgetsList.append(ConditionWidgetInstance)
        # Bind signal emitted when the condition is deleted to the slot
        self.ConditionWidgetsList[len(self.ConditionWidgetsList) -
        1].DeleteConditionSignal.connect(self.DeleteConditionSlot)

        # Add widget to scroll area
        self.ScrollAreaLayout.addWidget(ConditionWidgetInstance)

    # This slot is executed when DeleteConditionSignal is emitted
    # It removes the deleted object from the list of condition widgets
    def DeleteConditionSlot(self, Sender):
        self.ConditionWidgetsList.remove(Sender)

    # This method is executed when the user presses SubmitBTN
    # It retrieves the user's inputs and constructs a SQL query from them
    def ConductSearch(self):
        # SQLStatement stores the string of the SQL query which will be executed
        SQLStatement = "SELECT * FROM " + self.OpenCollectionTable + " WHERE "
        # Iterate through each condition widget, retrieving the user's inputs and adding the new
          conditions to the query
        for widget in self.ConditionWidgetsList:
            # If the widget is the first widget in the scroll area, there won't be any OR or AND
              statement coming after it
            if widget.First:
                SQLStatement += "`" + widget.ConditionCB.currentText() + "`"
            else:
                # Append the user's choice of operator, then the choice of column to search,
                  then the search text to the string
                SQLStatement += widget.OperatorCB.currentText() + " `" +
                widget.ConditionCB.currentText() + "`"
            # Use the appropriate comparison operator based on the user's choice in the
              MatchType combobox
            if widget.MatchTypeCB.currentText() == "Exact Match":
                SQLStatement += " = '" + widget.ConditionTB.text() + "' "
            else:
                SQLStatement += " LIKE '%" + widget.ConditionTB.text() + "%' "
        # Execute SQL query
        self.MyCursor.execute(SQLStatement)
        MyResult2 = self.MyCursor.fetchall()
```

```
        # Filter the query results based on the user's rarity filter
        for result in MyResult2:
            # If the user wants results that are both rare and not rare, do nothing to the
              results
            if self.RareCB.currentText() == "Both":
                continue
            # If the user only wants rare results, delete all non-rare results
            elif self.RareCB.currentText() == "True":
                if not result[len(result) - 3]:
                    MyResult2.remove(result)
            # If the user only wants non-rare results, delete all rare results
            else:
                if result[len(result) - 3]:
                    MyResult2.remove(result)

        # Filter the query results based on the user's thumbnail
        for result in MyResult2:
            # If the user wants results that both have a thumbnail and don't, do nothing to the
              results
            if self.ThumbnailCB.currentText() == "Both":
                continue
            # If the user only wants results with a thumbnail, delete all results that don't
              have a thumbnail
            elif self.ThumbnailCB.currentText() == "True":
                if result[len(result) - 1] is None:
                    MyResult2.remove(result)
            # If the user only wants results without a thumbnail, delete all results that do
              have a thumbnail
            else:
                if result[len(result) - 1] is not None:
                    MyResult2.remove(result)

        # If the SQL query returns no results, display the error message
        if len(MyResult2) == 0:
            self.ErrorLBL.show()
            QTimer.singleShot(3000, self.ErrorLBL.hide)
        # If the search returns results, emit the corresponding signal
        else:
            # The signal takes the result of the query as a parameter
            self.SearchCompleteSignal.emit(MyResult2)
            self.close()

    # Inbuilt PyQt function
    # sizeHint is the preferred size of the widget
    # It has to be set so that the painter event can paint the outline image according to these
      dimensions
    def sizeHint(self):
        return QSize(725, 425)

    # This event runs when the window is first initialized
    # It draws the outline/background of the window as the image "AddItemOutline.png"
    # This allows me to create irregularly shaped windows
    def paintEvent(self, event):
        # Create painter
        Painter = QPainter()
        # Start painter, set painter to perform on AddItemWindow (self)
        Painter.begin(self)
        # Create pixmap to act as guide for painter
        Outline = QPixmap()
        Outline.load('Resources/AdvancedSearchOutline.png')
        # Paint window as Outline pixmap
        Painter.drawPixmap(QPoint(0, 0), Outline)
        # End paint event
        Painter.end()

# This class inherits from QWidget
# It is the widget which allows users to input search terms, select the column to be searched,
  selects the comparison operator and choose AND or OR
class ConditionWidget(QWidget):
```

```
    # This signal is emitted when the user chooses to delete the widget through its context menu
    DeleteConditionSignal = pyqtSignal(QObject)
    def __init__(self, Conditions, DataTypes, First):
        super(ConditionWidget, self).__init__()

        # Set arguments to attributes
        # Conditions is a list of all the columns in the collection's SQL table
        self.Conditions = Conditions
        # DataTypes is a list of each column's data type
        self.DataTypes = DataTypes
        # First is a boolean value conveying if the widget is the first condition widget or not
        self.First = First

        self.setObjectName("BorderlessWidget")

        # Initialise and set widget layout
        self.ConditionWidgetLayout = QHBoxLayout()
        self.setLayout(self.ConditionWidgetLayout)

        # If the widget isn't the first widget, it will contains an OperatorCB for selecting the
          condition's logical operator
        if not First:
            self.OperatorCB = QComboBox()
            self.OperatorCB.addItems(["or", "and"])
            self.OperatorCB.setFixedWidth(70)
            self.ConditionWidgetLayout.addWidget(self.OperatorCB)

        # This text box is for inputting the search term for this widget
        self.ConditionTB = QLineEdit()
        self.ConditionWidgetLayout.addWidget(self.ConditionTB)

        self.ConditionWidgetLayout.addWidget(QLabel("in"))

        # This combobox contains the name of each column in the table as an option to choose
          from
        # It is used to select the column the user would like to query
        self.ConditionCB = QComboBox()
        self.ConditionCB.addItems(self.Conditions)
        self.ConditionCB.currentTextChanged.connect(self.CurrentTextChangedSlot)
        self.ConditionCB.setFixedWidth(130)
        self.ConditionWidgetLayout.addWidget(self.ConditionCB)

        # The user has a choice between exact matches using the = operator, or partial matches
          using the LIKE operator
        self.MatchTypeCB = QComboBox()
        self.MatchTypeCB.addItems(["Exact Match", "Partial Match"])
        self.MatchTypeCB.setFixedWidth(130)
        self.ConditionWidgetLayout.addWidget(self.MatchTypeCB)

    # This method is executed when the user selects another item in the combobox
    def CurrentTextChangedSlot(self):
        # If the selected item's data type is integer, a validator is set for the input box to
          ensure only integers are inputted
        if self.DataTypes[self.ConditionCB.currentIndex()] == b'int':
            self.ConditionTB.setText("")
            self.ConditionTB.setValidator(QIntValidator())
        # If the datatype is anything else, the validator is not needed
        else:
            self.ConditionTB.setValidator(None)

    # This event occurs when the user right-clicks on the widget
    # It displays a contextMenu with the option to delete the condition widget
    def contextMenuEvent(self, event):
        # If the widget is the first one, it can't be deleted as each search query needs at
          least one condition to be successful
        if self.First:
            pass
        # Every other widget can be deleted
        else:
            # Initialise context menu
```

```
            ConditionWidgetContextMenu = QMenu(self)
            # Add action to context menu
            DeleteAction = ConditionWidgetContextMenu.addAction("Delete Condition")
            Action = ConditionWidgetContextMenu.exec_(self.mapToGlobal(event.pos()))
            # If the user selects the delete option from the context menu...
            if Action == DeleteAction:
                # Emit the delete signal which is received by the AdvancedSearch parent class
                self.DeleteConditionSignal.emit(self)
                # Setting the widget's parent to None deletes it
                self.setParent(None)
```

GraphWindow.py

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
from matplotlib import dates as matplotlibdates
from math import log10, floor
from datetime import datetime
import os

# This object is the window which displays the graphs generated from the user's collection
# It uses matplotlib to plot the graphs
class GraphWidget(QWidget):
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID, OpenCollectionTable,
    widget = None):
        super(GraphWidget, self).__init__()

        # Reassign arguments as attributes
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.ActiveUserID = ActiveUserID
        self.OpenCollectionID = OpenCollectionID
        self.OpenCollectionTable = OpenCollectionTable

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window attributes (same as AddItemWindow)
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setAttribute(Qt.WA_TranslucentBackground)
        self.setWindowModality(Qt.ApplicationModal)
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint | Qt.WindowStaysOnTopHint))

        # Initialise main window layout
        self.GraphWidgetVBL1 = QVBoxLayout()
        self.setLayout(self.GraphWidgetVBL1)

        # Retrieve column names from table
        self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
        self.GraphTypes = self.MyCursor.fetchall()
        # Remove thumbnails from list
        self.GraphTypes.pop(len(self.GraphTypes) - 1)
        # Add Collection Size category to list
        self.GraphTypes.append(("Collection Size", ))
        # Index keeps track of which graph is currently being displayed
        self.GraphTypesIndex = 1

        # Horizontal layout for displaying the graph/chart and its legend
        self.FigureHBL = QHBoxLayout()

        self.FigureWidget = QWidget()
        self.FigureWidget.setObjectName("BorderlessWidget")
        self.FigureWidgetLayout = QVBoxLayout()
        self.FigureWidgetLayout.setContentsMargins(0, 0, 0, 0)
```

```
        self.FigureWidget.setLayout(self.FigureWidgetLayout)
        # Create MatPlotLib figure for displaying the graph/chart
        self.Figure = Figure()
        # Create MatPlotLib canvas and pass figure in as argument
        self.Canvas = FigureCanvas(self.Figure)
        # Add canvas to layout
        self.FigureWidgetLayout.addWidget(self.Canvas)
        self.FigureHBL.addWidget(self.FigureWidget)
        self.GraphWidgetVBL1.addLayout(self.FigureHBL)

        # Create MatPlotLib figure for displaying the legend
        self.LegendFigure = Figure()
        # Create MatPlotLib canvas and pass figure in as argument
        self.LegendCanvas = FigureCanvas(self.LegendFigure)
        self.FigureHBL.addWidget(self.LegendCanvas)

        # Horizontal box layout for displaying the navigation and action buttons
        self.NavigationHBL = QHBoxLayout()
        # BackBTN is used to navigate to the previous graph
        self.BackBTN = QPushButton()
        self.BackBTN.setIcon(QIcon("Resources/LeftArrow2.png"))
        self.BackBTN.clicked.connect(self.PreviousGraph)
        self.BackBTN.setFixedWidth(40)
        # ForwardBTN is used to naviagte to the next graph
        self.ForwardBTN = QPushButton()
        self.ForwardBTN.setIcon(QIcon("Resources/RightArrow2.png"))
        self.ForwardBTN.clicked.connect(self.NextGraph)
        self.ForwardBTN.setFixedWidth(40)
        # CloseBTN closes the window
        self.CloseBTN = QPushButton("Close")
        self.CloseBTN.clicked.connect(self.close)
        self.CloseBTN.setFixedWidth(50)
        # ExportBTN is used to export the graph as a png
        self.ExportBTN = QPushButton("Save as Image")
        self.ExportBTN.clicked.connect(self.ExportGraph)
        self.ExportBTN.setFixedWidth(100)
        # Label displays message to user when graph is successfully exported as image
        self.ExportStatusLBL = QLabel("Successfully Exported graph as 'AnthologyExport.png'")
        self.ExportStatusLBL.hide()
        # Add buttons to layout
        self.NavigationHBL.addWidget(self.BackBTN)
        self.NavigationHBL.addWidget(self.ForwardBTN)
        self.NavigationHBL.addWidget(self.CloseBTN)
        self.NavigationHBL.addWidget(self.ExportBTN)
        self.NavigationHBL.addWidget(self.ExportStatusLBL)
        self.GraphWidgetVBL1.addLayout(self.NavigationHBL)
        # Push buttons to the left hand side of the layout
        self.NavigationHBL.addStretch()

        # Execute ChooseGraph method to display the first graph
        self.ChooseGraph(self.GraphTypes[self.GraphTypesIndex][0])

        # Position window below GenerateGraphsBTN using widget argument
        x = widget.mapToGlobal(QPoint(0, 0)).x() - 720
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        self.move(x, y)

        # Show window
        self.show()

    # Executed when the user clicks ExportBTN
    # It exports the graph currently being displayed as a png
    def ExportGraph(self):
        # Open file dialog and allow user to select a directory to export to
        # The file path of this directory is stored under the variable FilePath
        self.FilePath = QFileDialog.getExistingDirectory(None, "Select Export Directory",
        os.path.abspath(os.sep))
        # If the graph being exported is a line chart, the legend is not required
        if self.GraphTypes[self.GraphTypesIndex] == ("Collection Size",):
            # Save the image to the user-specified directory with the name AnthologyExport.png
```

```
            try:
                self.Figure.savefig(self.FilePath + "/AnthologyExport.png")
            except:
                pass
        # If the graph is a pie chart and has an index, QPainter is used to place the two images
          together and export them as a single image
        else:
            # Save graph and legend as separate images in Temp directory
            # They need to be joined together to produce the final export image
            self.Figure.savefig("Temp/Export1.png")
            self.LegendFigure.savefig("Temp/Export2.png")
            # Now load both images as QImage objects
            GraphImage = QImage("Temp/Export1.png")
            LegendImage = QImage("Temp/Export2")
            # Create a new QPixmap for the export
            # The width of the new image is the sum of the width of the graph image and the
              legend image
            FinalExport = QPixmap(QSize(GraphImage.width() + LegendImage.width(),
            GraphImage.height()))
            # Start painter, set painter to perform on FinalExport
            Painter = QPainter(FinalExport)
            # Draw the graph image starting at point (0, 0)
            Painter.drawImage(QPoint(0, 0), GraphImage)
            # Draw the legend image alongside the graph image
            Painter.drawImage(QPoint(GraphImage.width(), 0), LegendImage)
            # End painter
            Painter.end()
            # Save image
            FinalExport.save(self.FilePath + "/AnthologyExport.png")
            # Remove images from Temp directory
            os.remove("Temp/Export1.png")
            os.remove("Temp/Export2.png")

        # Display message to user informing them that the image has been successfully imported
        self.ExportStatusLBL.show()
        # Set message to disappear after 3 seconds
        QTimer.singleShot(3000, self.ExportStatusLBL.hide)

    # This method determines whether the program needs to draw a line graph or a pie chart
    def ChooseGraph(self, Type):
        if Type == "Collection Size":
            self.DrawLineGraph()
        else:
            self.DrawPieChart(Type)

    # This function plots a pie chart and creates a legend
    # It takes the parameter Type, which is the collection field that it is plotting
    def DrawPieChart(self, Type):
        self.FigureWidget.setFixedWidth(380)
        # Clear figure of previous graph
        self.Figure.clear()
        # Group values stored under the current column and return the number in each group
        self.MyCursor.execute("SELECT " + Type + ", COUNT(*) AS c FROM " +
        self.OpenCollectionTable + " GROUP BY " + Type)
        MyResult1 = self.MyCursor.fetchall()
        # Total is the total number of items being plotted (used to calculate percentages)
        self.Total = 0
        # GraphLabels is a list containing each category in the pie chart
        self.GraphLabels = []
        # Percentages is a list containing the percentage of each category
        self.Percentages = []
        # Count number of items by incrementing Total for each value in MyResult1
        for x in range(0, len(MyResult1)):
            self.Total += MyResult1[x][1]
        # Iterate through MyResult1, calculating the percentage of each category and adding
          category labels to the list
        for y in range(0, len(MyResult1)):
            # Calculate category's percentage of the Total
            self.Percentages.append((MyResult1[y][1] / self.Total) * 100)
            # If the category name is longer than 17 characters, cut it down and add an ellipses
```

```
            to the end
            if len(str(MyResult1[y][0])) > 17:
                Label = str(MyResult1[y][0][:17]) + "..."
            else:
                Label = str(MyResult1[y][0])
            # If we are plotting the rarity, we have to replace 1 and 0 with True and False fo
              our legend
            if Type == "Rare":
                if MyResult1[y][0]:
                    Label = "True"
                else:
                    Label = "False"
            # Add the label to the list, percentages are round to the nearest whole number
            self.GraphLabels.append(Label + " (" + str(round(self.Percentages[y],
            int(floor(log10(abs(self.Percentages[y])))))) + "%)")
        # Create axis object for plotting values
        self.Axis = self.Figure.add_axes((0, 0, 1, 1))
        # Set the background colour of the figure to white
        self.Figure.set_facecolor("#FFFFFF")
        # Plot pie chart using values stored in Percentages list
        self.patches, texts = self.Axis.pie(self.Percentages, startangle = 90)
        # Set the graph's aspect ration to "equal" to display the pie chart as a perfect circle
          rather than an ellipse
        self.Axis.axis("equal")

        # Clear figure of previous legend
        self.LegendFigure.clear()
        self.LegendFigure.tight_layout()
        # Create legend
        self.Legend = self.LegendFigure.legend(self.patches, self.GraphLabels, prop = {"size":
        7}, ncol = 2)
        # Set legend title to Type parameter
        self.Legend.set_title(Type, prop = {"size": 10})

        # Call draw() method on both canvases to display new graphs
        self.Canvas.draw()
        self.LegendCanvas.draw()
        self.LegendCanvas.show()

    # The method plots a line chart
    # It is called when the user wants to view the collection sizes chart
    def DrawLineGraph(self):
        self.FigureWidget.setFixedWidth(750)
        # The line chart doesn't need a legend so its canvas is hidden
        self.LegendCanvas.hide()
        # Clear figure of previous graph
        self.Figure.clear()
        # Get all entries into the sizes table where the foreign key corresponds with the
          primary key of the currently open collection
        self.MyCursor.execute("SELECT * FROM Sizes WHERE FK_Collections_Sizes = " +
        str(self.OpenCollectionID))
        MyResult2 = self.MyCursor.fetchall()
        # Dates is a list storing the date when each size was recorded
        self.Dates = []
        # Magnitudes stores the size of the collection at each date
        self.Magnitudes = []
        # Iterate through the results of SQL query and append them to the appropriate list
        for x in range(0, len(MyResult2)):
            self.Dates.append(MyResult2[x][1])
            self.Magnitudes.append(MyResult2[x][2])
        # Convert MySQL date values to python-compatible datetime objects
        self.Dates = [datetime.strptime(str(d), "%Y-%m-%d %H:%M:%S") for d in self.Dates]
        # Convert python datetime objects to MatPlotLib compatible values
        self.Dates = matplotlibdates.date2num(self.Dates)
        # Create a formatter for the x axis to correctly display the time values
        self.DatesFormat = matplotlibdates.DateFormatter("%Y-%m-%d\n%H:%M:%S")
        # Create axis object for plotting values
        self.Axis = self.Figure.add_subplot()
        self.Figure.set_facecolor("#FFFFFF")
        self.Figure.tight_layout()
```

```
        # Set the formatter of the x axis to the newly created formatter
        self.Axis.xaxis.set_major_formatter(self.DatesFormat)
        # Plot dates against sizes as a line chart
        self.Axis.plot(self.Dates, self.Magnitudes, marker = "x", markersize = 5, linewidth = 1)
        # Set the graph's title
        self.Axis.set_title("Collecton Size", size = 12)
        # Set the x axis label
        self.Axis.set_xlabel("Date", size = 10)
        # Set the y axis label
        self.Axis.set_ylabel("Size", size = 10)
        self.Axis.tick_params(axis = "both", labelsize = 6)
        self.Figure.tight_layout()

        # Call draw() method on both canvases to display new graphs
        self.LegendFigure.clear()
        self.Canvas.draw()
        self.LegendCanvas.draw()

    # Executed when ForwardBTN is clicked
    def NextGraph(self):
        # If the index is at the end of the list, set the index to the beginning of the list
        if self.GraphTypesIndex >= len(self.GraphTypes) - 1:
            self.GraphTypesIndex = 1
        # Otherwise, increment the index by 1
        else:
            self.GraphTypesIndex += 1
        # Draw the new graph
        self.ChooseGraph(self.GraphTypes[self.GraphTypesIndex][0])

    # Executed when BackBTN is clicked
    def PreviousGraph(self):
        # If the index is at the start of the list, set the index to the end of the list
        if self.GraphTypesIndex <= 1:
            self.GraphTypesIndex = len(self.GraphTypes) - 1
        # Otherwise, decrement the index by 1
        else:
            self.GraphTypesIndex -= 1
        # Draw the new graph
        self.ChooseGraph(self.GraphTypes[self.GraphTypesIndex][0])


    def sizeHint(self):
        return QSize(815, 415)

    def paintEvent(self, event):
        Painter = QPainter()
        Painter.begin(self)
        Outline = QPixmap()
        Outline.load('Resources/GraphOutline.png')
        Painter.drawPixmap(QPoint(0, 0), Outline)
        Painter.end()
```

BarcodeSearch.py

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from bs4 import BeautifulSoup
import pyzbar
import cv2
import requests

# This class contains the window which will display the live video feed, an instruction label
  and a cancel button
# The two concurrent threads are initialised in this window
# The only argument it takes is the button widget clicked to create it
class BarcodeSearch(QWidget):
    def __init__(self, widget = None):
```

```
        super(BarcodeSearch, self).__init__()

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window attributes
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setAttribute(Qt.WA_TranslucentBackground)
        self.setWindowModality(Qt.ApplicationModal)
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint | Qt.WindowStaysOnTopHint))

        # Initialise main layout
        self.BarcodeSearchVBL1 = QVBoxLayout()
        self.setLayout(self.BarcodeSearchVBL1)

        # This label displays each frame captured by the webcam
        self.FeedLabel = QLabel()
        # Set the image displayed by the window to a generic placeholder image whilst the webcam
          feed is being set up
        self.FeedLabel.setPixmap(QPixmap("Resources/WebCamConnecting.png"))
        self.BarcodeSearchVBL1.addWidget(self.FeedLabel)

        # This label will be updated to relay messages to the user
        # Currently, it is telling the user how to scan a barcode using the webcam
        self.InstructionLabel = QLabel("Position barcode in camera view")
        self.BarcodeSearchVBL1.addWidget(self.InstructionLabel)

        # The user presses this button when they want to close the window and stop the video
          feed
        self.CancelBTN = QPushButton("Cancel")
        # The button's clicked signal is connected to the CancelFeed method
        self.CancelBTN.clicked.connect(self.CancelFeed)
        self.BarcodeSearchVBL1.addWidget(self.CancelBTN)

        # Initialise Thread ----------------------------------------------------
        self.Worker1 = VideoFeedWorker()
        # Connect the object's signals to slots/methods in this class
        self.Worker1.ImageUpdateSignal.connect(self.ImageUpdateSlot)
        self.Worker1.BarcodeDetectedSignal.connect(self.BarcodeDetectedSlot)
        # Start the thread
        self.Worker1.start()
        # ----------------------------------------------------------------------

        # Determine where window should be positioned based on the widget which is passed in as
          an argument
        x = widget.mapToGlobal(QPoint(0, 0)).x() - 600
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        # The window should appear on the screen just below the button which is clicked to
          create it
        self.move(x, y)

        self.show()

    # This method is executed when the user presses the cancel button
    def CancelFeed(self):
        # Stop the thread from executing
        # If this isn't carried out, the program would continue capturing video which would
          affect the performance of the program and raise privacy concerns
        self.Worker1.stop()
        # Close window
        self.close()

    # This method is executed when VideoFeedWorker emits BarcodeDetectedSignal
    # It initialises the second QThread object which performs the online barcode search
    def BarcodeDetectedSlot(self, Barcode):
        # Change the text of InstructionLabel to inform the user that the barcode is being
          searched for
        self.InstructionLabel.setText("Searching Barcode...")
        # Initialise new thread, passing in the detected barcode as an argument
```

```
        self.Worker2 = WebSearchWorker(Barcode = Barcode)
        # Connect the object's signals to slots/methods in the parent class
        self.Worker2.SearchCompleteSignal.connect(self.SearchCompleteSlot)
        self.Worker2.SearchFailedSignal.connect(self.SearchFailedSlot)
        # Start the thread
        self.Worker2.start()

    # This method is executed when VideoFeedWorker emits ImageUpdateSignal
    # It sets the image displayed in FeedLabel to the image emitted by the thread
    def ImageUpdateSlot(self, Frame):
        self.FeedLabel.setPixmap(QPixmap.fromImage(Frame))

    # This method is executed when WebSearchWorker emits SearchCompleteSignal
    # It relays the results of the barcode search to the user
    def SearchCompleteSlot(self, Result):
        # Create an instance of the popup, passing the results of the barcode search and
          FeedLabel object in as arguments
        self.SearchResultPopupInstance = SearchResultPopup(Result = Result, FeedLabel =
        self.FeedLabel)
        # Connect signals to slots
        self.SearchResultPopupInstance.CopiedSignal.connect(self.CopiedSlot)
        self.SearchResultPopupInstance.NewSearchSignal.connect(self.NewSearchSlot)
        # Create image effect object and apply to FeedLabel
        Effect = QGraphicsColorizeEffect(self.FeedLabel)
        # This effect will change the saturation of the image displayed in the label
        Effect.setStrength(0.7)
        Effect.setColor(QColor("silver"))
        self.FeedLabel.setGraphicsEffect(Effect)

    # This method is executed when SearchResultPopupInstance emits CopiedSignal
    # It changes the text in InstructionLabel to inform the user that they have successfully
      copied the search results to their clipboard
    def CopiedSlot(self):
        self.InstructionLabel.setText("The search result was copied to your clipboard")
        # Message should display for 6 seconds before feedLabel returns to its usual state
        QTimer.singleShot(6000, self.RevertInstructionLabel)

    # This method is executed when SearchResultPopupInstance emits NewSearchSignal
    # It restarts the Worker1 thread, allowing the user to scan a new barcode
    def NewSearchSlot(self):
        self.FeedLabel.setGraphicsEffect(None)
        # Set FeedLabel's image to a generic placeholder image whilst the webcam feed is being
          restarted
        self.FeedLabel.setPixmap(QPixmap("Resources/WebCamConnecting2.png"))
        self.Worker1.start()

    # This method is executed when WebSearchWorker emits SearchFailedSignal
    def SearchFailedSlot(self):
        # Sets the text in InstructionLabel to an error message
        self.InstructionLabel.setText("We couldn't find that specific barcode")
        self.InstructionLabel.setStyleSheet("color: red")
        QTimer.singleShot(6000, self.RevertInstructionLabel)
        # Restart video feed
        self.FeedLabel.setPixmap(QPixmap("Resources/WebCamConnecting2.png"))
        self.Worker1.start()

    # This method reverts the InstructionLabel back to its original text
    # It is executed in the QTimer events
    def RevertInstructionLabel(self):
        self.InstructionLabel.setText("Position barcode in camera view")
        self.InstructionLabel.setStyleSheet("color: black")

    # This method is executed when there is an error establishing a connection with the webcam
      (e.g. the user has no webcam or the webcam is broken)
    def VideoCaptureErrorSlot(self):
        # Display error message
        self.InstructionLabel.setText("There was an issue capturing video, please try
        relaunching this window")
        self.FeedLabel.setPixmap(QPixmap("Resources/WebcamError.png"))
        self.InstructionLabel.setStyleSheet("color: red")
```

```
    # Same paint event as AddItemWindow
    def sizeHint(self):
        return QSize(690, 465)

    def paintEvent(self, event):
        Painter = QPainter()
        Painter.begin(self)
        Outline = QPixmap()
        Outline.load('Resources/BarcodeSearchOutline.png')
        Painter.drawPixmap(QPoint(0, 0), Outline)
        Painter.end()
# This object is the thread which handles displaying the openCV video feed on the user's screen
# It inherits from QThread
class VideoFeedWorker(QThread):
    # Signal for when a new frame has been captured
    # it takes the captured frame image as a parameter
    ImageUpdateSignal = pyqtSignal(QImage)
    # Signal for when a barcode is detected in the frame
    # It takes the upc of the barcode as a parameter
    BarcodeDetectedSignal = pyqtSignal(str)
    # Signal for when there is an error capturing frames
    VideoCaptureErrorSignal = pyqtSignal()
    # Method executed when start() is called on the object
    # Initialises the thread loop
    def run(self):
        # Counter keeps track of how many frames have been recorded
        Counter = 0
        # Boolean value used to signal whether the while loop capturing each frame should
          continue executing or not
        # This value is set to False when the user presses the cancel button
        self.ThreadActive = True
        # Initialise connection with webcam, the 0 parameter tells the program to select the
          first webcam connection if the computer has multiple camera inputs
        Capture = cv2.VideoCapture(0)
        # This while loop captures each frame, processes it, scans it for barcodes and sends it
          back to the main program loop to be displayed in FeedLabel
        # It keeps iterating until boolean is set to False
        # Because it is executing in a concurrent thread, the while loop won't freeze the main
          program loop
        while self.ThreadActive:
            # Capture frame
            Ret, Frame = Capture.read()
            # If the frame is successfully captured...
            if Ret:
                # Convert frame to object
                Image = cv2.cvtColor(Frame, cv2.COLOR_BGR2RGB)
                # Flip image (OpenCV doesn't automatically mirror images)
                FlippedImage = cv2.flip(Image, 1)
                # The program scans for barcodes every 5 frames
                if Counter % 5 == 0:
                    # Use pyzbar to scan frame for barcodes
                    Barcodes = pyzbar.decode(Frame)
                    # If a barcode is present
                    if Barcodes:
                        # Get the dimensions of the barcode
                        x, y, w, h = Barcodes[0].rect
                        # Draw a red rectangle around the barcode
                        RectImage = cv2.rectangle(FlippedImage, (x, y), (x + w, y + h), (255, 0,
                        0), 5)
                        # Convert the frame to a format that is compatible with PyQt
                        ConvertToQtFormat = QImage(RectImage.data, RectImage.shape[1],
                        RectImage.shape[0], QImage.Format_RGB888)
                        # Scale image
                        FinalImage = ConvertToQtFormat.scaled(640, 480, Qt.KeepAspectRatio)
                        # Emit signal with the captured frame passed in as a parameter
                        # When the parent object receives its signal, it will set the image
                          displayed in FeedLabel to the frame emitted with the signal
                        self.ImageUpdateSignal.emit(FinalImage)
                        # Emit signal with the upc of the scanned barcode passed in as a
```

```
                        parameter
                        self.BarcodeDetectedSignal.emit(Barcodes[0].data.decode("utf-8"))
                        # Stop the video feed thread
                        self.stop()
                    # If a barcode isn't detected in the frame...
                    else:
                        # Convert frame to PyQt compatible format
                        ConvertToQtFormat = QImage(FlippedImage.data, FlippedImage.shape[1],
                        FlippedImage.shape[0], QImage.Format_RGB888)
                        # Scale image
                        FinalImage = ConvertToQtFormat.scaled(640, 480, Qt.KeepAspectRatio)
                        # Emit signal
                        self.ImageUpdateSignal.emit(FinalImage)
                # This is the code executed when the number of frames isn't a multiple of 5
                else:
                    # Convert frame to PyQt compatible format
                    ConvertToQtFormat = QImage(FlippedImage.data, FlippedImage.shape[1],
                    FlippedImage.shape[0], QImage.Format_RGB888)
                    # Scale image
                    FinalImage = ConvertToQtFormat.scaled(640, 480, Qt.KeepAspectRatio)
                    # Emit signal
                    self.ImageUpdateSignal.emit(FinalImage)
                # Increment counter
                Counter += 1
            # if there is an error capturing the frame, emit the error signal
            else:
                self.VideoCaptureErrorSignal.emit()

    # This method stops the thread from executing
    def stop(self):
        self.ThreadActive = False
        self.quit()

# This thread handles the webscraping barcode search
# Searches for the barcode in two different web databases: upcscavenger.com and upcitemdb.com
# It aslo inherits from QThread
class WebSearchWorker(QThread):
    # Signal for when the search has been successfully completed
    # It takes the string result of the search as a parameter
    SearchCompleteSignal = pyqtSignal(str)
    # Signal for when the search fails to find a result
    SearchFailedSignal = pyqtSignal()

    # Thread constructor
    def __init__(self, Barcode, parent = None):
        QThread.__init__(self, parent)
        self.Barcode = Barcode

    # Start thread
    def run(self):
        # requests.get makes a connection with the website
        # Format the barcode into the url to search for it in their database
        Response = requests.get("http://www.upcscavenger.com/barcode/" + self.Barcode +
        "/#page=barcode")
        # Create a parser object, passing in the HTML file from the requests.get()
        Soup = BeautifulSoup(Response.text, "html.parser")
        # Parse the file for the specific class which contains the result text, then strip any
          line breaks from the result
        Result = Soup.find(class_ = "us2329735521 us2445479844").find(class_ =
        "us1881050501").get_text().replace("\n", "")
        # If the search on the first website does not return a result, try searching the second
          website
        if Result == self.Barcode:
            try:
                # Connect with second website
                Response2 = requests.get("https://www.upcitemdb.com/upc/" + self.Barcode, verify
                = False)
                # Create second parser object
                Soup2 = BeautifulSoup(Response2.text, "html.parser")
                # Locate HTML class
```

```
                Result2 = Soup2.find(class_ = "num").find("li").get_text()
                # Emit signal
                self.SearchCompleteSignal.emit(Result2)
            # If the second search also doesn't return a result, emit SearchFailedSignal
            except:
                self.SearchFailedSignal.emit()
        # If the first search is successful, emit SearchCompleteSignal, passing in the search
          result as a parameter
        else:
            self.SearchCompleteSignal.emit(Result)

# This object is the window which relays the results of the barcode search to the user
# It takes the result of the search and the FeedLabel object as arguments
class SearchResultPopup(QWidget):
    # Signal for when the search result has been copied to the clipboard
    CopiedSignal = pyqtSignal()
    # Signal for when the user wants to scan a new barcode
    NewSearchSignal = pyqtSignal()
    def __init__(self, Result, FeedLabel):
        super(SearchResultPopup, self).__init__()

        # Assign arguments as attributes
        self.Result = Result
        self.FeedLabel = FeedLabel

        # Create clipboard object
        self.Clipboard = QApplication.clipboard()
        self.Clipboard.clear(mode=self.Clipboard.Clipboard)

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Set window attributes
        self.setWindowModality(Qt.ApplicationModal)
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint | Qt.WindowStaysOnTopHint))

        # Initialise main layout
        self.VBL = QVBoxLayout()
        self.setLayout(self.VBL)

        InfoLabel = QLabel("Your Search Result Is:")
        InfoLabel.setStyleSheet("font-size: 14px;")
        self.VBL.addWidget(InfoLabel)

        # Label for displaying the result of the search
        ResultLabel = QLabel(self.Result)
        ResultLabel.setStyleSheet("font-size: 18px; color: red; padding: 5px;")
        # The label will be displayed in a scroll area in case the label is longer than the
          width of the window
        self.ResultScrollArea = QScrollArea()
        self.ResultScrollArea.setWidget(ResultLabel)
        self.ResultScrollArea.setStyleSheet("QScrollBar:horizontal{border-bottom: 0px; border-
        left: 0px; border-right: 0px;}QScrollBar::handle{border-left: 0px; border-right: 0px;}")
        self.ResultScrollArea.setFixedHeight(50)
        self.VBL.addWidget(self.ResultScrollArea)

        self.ActionBTNHBL = QHBoxLayout()
        # Button for copying the search result to the clipboard
        self.CopyClickboardBTN = QPushButton("Copy to clipboard")
        self.CopyClickboardBTN.clicked.connect(self.CopyToClipboard)
        self.ActionBTNHBL.addWidget(self.CopyClickboardBTN)
        # Button for starting a new barcode search
        self.ScanAgain = QPushButton("Scan a new barcode")
        self.ScanAgain.clicked.connect(self.StartNewSearch)
        self.ActionBTNHBL.addWidget(self.ScanAgain)
        self.VBL.addLayout(self.ActionBTNHBL)

        self.setFixedSize(400, 140)
```

```
        # Position widget in the middle of FeedLabel
        x = self.FeedLabel.mapToGlobal(QPoint(0, 0)).x() + (self.FeedLabel.width() / 2) -
        (self.width() / 2)
        y = self.FeedLabel.mapToGlobal(QPoint(0, 0)).y() + (self.FeedLabel.height() / 2) -
        (self.height() / 2)
        self.move(x, y)

        self.show()

    # When the user wants to start a new barcode search, the appropriate signal is emitted and
      the popup window is closed
    def StartNewSearch(self):
        self.NewSearchSignal.emit()
        self.close()

    # When the user wants to copy the result, it is copied to their clipboard and the
      appropriate signal is emitted
    def CopyToClipboard(self):
        self.Clipboard.setText(self.Result)
        self.CopiedSignal.emit()
```

LoanItem.py

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

# This class is the window for managing loaned items
class LoanItemWindow(QWidget):
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID, OpenCollectionTable,
    widget = None):
        super(LoanItemWindow, self).__init__()

        # Reassign arguments as attributes
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.ActiveUserID = ActiveUserID
        self.OpenCollectionID = OpenCollectionID
        self.OpenCollectionTable = OpenCollectionTable

        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Initialise main layout
        self.LoanItemVBL1 = QVBoxLayout()
        self.setLayout(self.LoanItemVBL1)
        self.LoanItemHBL1 = QHBoxLayout()
        self.LoanItemVBL1.addLayout(self.LoanItemHBL1)

        # Set window attributes
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setAttribute(Qt.WA_TranslucentBackground)
        self.setWindowModality(Qt.ApplicationModal)
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint | Qt.WindowStaysOnTopHint))

        self.InitUI()

        # Calculate window position
        x = widget.mapToGlobal(QPoint(0, 0)).x() - 225
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        self.move(x, y)

        self.show()
        self.setFixedWidth(905)

    # Initialise widgets
```

```
    def InitUI(self):
        # The window is split into two columns; one for items that are available to loan and one
          for items which have already been loaned out
        self.ToLoanListLayout = QVBoxLayout()
        self.ToLoanListLayout.addWidget(QLabel("Items Available to Loan:"))
        # ToLoanList displays all the items in the collection which haven't been loaned out yet
        self.ToLoanList = QListWidget()
        # Style widget
        self.ToLoanList.setStyleSheet("selection-color: black;")
        self.ToLoanList.verticalScrollBar().setStyleSheet("QScrollBar { border-right: 0px;
        border-top: 0px; border-bottom: 0px; } QScrollBar::handle { border-top: 0px; border-
        bottom: 0px; }")
        # Connect the signal emitted when the list's selection is changed to a slot
        self.ToLoanList.itemSelectionChanged.connect(self.ItemSelected)
        # Don't display focus rectangle when item is selected
        self.ToLoanList.setAttribute(Qt.WA_MacShowFocusRect, 0)
        # Add list to layout
        self.ToLoanListLayout.addWidget(self.ToLoanList)
        self.LoanItemHBL1.addLayout(self.ToLoanListLayout)
        self.ToLoanList.setFixedWidth(300)

        self.AlreadyLoanedListLayout = QVBoxLayout()
        self.AlreadyLoanedListLayout.addWidget(QLabel("Loaned Items:"))
        # AlreadyLoanedList displays all the items that have already been loaned out
        self.AlreadyLoanedList = QListWidget()
        # Style widget
        self.AlreadyLoanedList.setStyleSheet("selection-color: black;")
        self.AlreadyLoanedList.verticalScrollBar().setStyleSheet("QScrollBar { border-right:
        0px; border-top: 0px; border-bottom: 0px; } QScrollBar::handle { border-top: 0px;
        border-bottom: 0px; }")
        # Don't show focus rectangle
        self.AlreadyLoanedList.setAttribute(Qt.WA_MacShowFocusRect, 0)
        # Add to layout
        self.AlreadyLoanedListLayout.addWidget(self.AlreadyLoanedList)
        self.LoanItemHBL1.addLayout(self.AlreadyLoanedListLayout)

        # Run PopulateLists method to fill both lists with items
        self.PopulateLists()

        # QDateEdit object used to input the due date of the selected item when loaning it out
        self.PickDueDate = QDateEdit()
        # Set the minimum date the user can input to the current date
        self.PickDueDate.setMinimumDate(QDate.currentDate())
        self.ToLoanListLayout.addWidget(self.PickDueDate)
        # The widget is hidden until the user selects an item in ToLoanList
        self.PickDueDate.hide()

        # OptionsWidget displays two checkboxes
        self.OptionsWidget = QWidget()
        self.OptionsWidget.setObjectName("BorderlessWidget")
        self.OptionsHBL = QHBoxLayout()
        self.OptionsHBL.setSpacing(0)
        self.OptionsHBL.setContentsMargins(0, 0, 0, 0)
        # This checkbox allows the user to choose if they would like to receive push
          notifications when an item is due
        self.PushNotificationCB = QCheckBox()
        self.PushNotificationCB.setFixedWidth(15)
        self.PushNotificationCB.setStyleSheet("padding-right: 0px;")
        self.PushNotificationCB.setChecked(True)
        self.OptionsHBL.addWidget(self.PushNotificationCB)
        PushLBL = QLabel("Push Notifications")
        PushLBL.setStyleSheet("padding-bottom: 1px; padding-left: 0px;")
        self.OptionsHBL.addWidget(PushLBL)
        # This checkbox allows the user to choose if they would like to receive email
          notifications when an item is due
        self.EmailNotificationCB = QCheckBox()
        self.EmailNotificationCB.setFixedWidth(15)
        self.EmailNotificationCB.setStyleSheet("padding-right: 0px;")
        self.EmailNotificationCB.setChecked(True)
        self.OptionsHBL.addWidget(self.EmailNotificationCB)
```

```
        EmailLBL = QLabel("Email Notifications")
        EmailLBL.setStyleSheet("padding-bottom: 1px; padding-left: 0px;")
        self.OptionsHBL.addWidget(EmailLBL)
        self.OptionsWidget.setLayout(self.OptionsHBL)
        self.ToLoanListLayout.addWidget(self.OptionsWidget)
        # These checkboxes are hidden until the user selects an item in ToLoanList
        self.OptionsWidget.hide()

        # This button allows the user to loan out the selected item in ToLoanList with the due
          date specified in PickDueDate
        self.SubmitLoanBTN = QPushButton("Loan Item")
        # Bind button to function
        self.SubmitLoanBTN.clicked.connect(self.SubmitLoanMethod)
        self.ToLoanListLayout.addWidget(self.SubmitLoanBTN)
        # Hide button until an item is selected in ToLoanList
        self.SubmitLoanBTN.hide()

        # Button for closing the window
        self.CloseWindowBTN = QPushButton("Close")
        self.CloseWindowBTN.setFixedWidth(50)
        self.CloseWindowBTN.clicked.connect(self.close)
        self.LoanItemVBL1.addWidget(self.CloseWindowBTN)

    # This method populates ToLoanList with items which haven't been loaned out yet and fills
      AlreadyLoanedList with the items in that collection which have already been loaned out
    def PopulateLists(self):
        # Clear both lists of their contents before repopulating them
        self.ToLoanList.clear()
        self.AlreadyLoanedList.clear()
        # Get the primary key of every item in the current collection's table
        self.MyCursor.execute("SELECT PK_" + self.OpenCollectionTable + " FROM " +
        self.OpenCollectionTable)
        MyResult1 = self.MyCursor.fetchall()
        # Presence check: if there are no items in the collection, do not proceed
        if len(MyResult1) == 0:
            pass
        # If there are items in the collection, go on to populate the lists
        else:
            # Get every entry into Loans which corresponds with the primary key of the current
              collection
            self.MyCursor.execute("SELECT KeyInCorrespondingTable FROM Loans WHERE
            FK_Collections_Loans = " + str(self.OpenCollectionID))
            MyResult2 = self.MyCursor.fetchall()

            # Remove all items from MyResult1 which are also in MyResult2
            # This should leave us with all the loaned items in MyResult2 and all the unloaned
              items in MyResult1
            MyResult1 = list(set(MyResult1) - set(MyResult2))
            # Repopulate tuple as list
            MyResult1 = [item[0] for item in MyResult1]
            # Sort items into ascending order
            MyResult1.sort()
            # If all the items in the collection have already been loaned out, don't proceed
              with populating ToLoanList
            if len(MyResult1) == 0:
                pass
            else:
                # Construct a query to select all the items in the SQL table with primary keys
                  in MyResult1
                if len(MyResult1) > 1:
                    SQLStatement = "SELECT * FROM " + self.OpenCollectionTable + " WHERE "
                    for x in range(0, len(MyResult1) - 1):
                        SQLStatement += "PK_" + self.OpenCollectionTable + " = " +
                        str(MyResult1[x]) + " OR "
                    SQLStatement += "PK_" + self.OpenCollectionTable + " = " +
                    str(MyResult1[len(MyResult1) - 1])
                    self.MyCursor.execute(SQLStatement)
                else:
                    self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " WHERE
```

```
                PK_" + self.OpenCollectionTable + " = " + str(MyResult1[0]))
            # Execute query to get a tuple of all the items which need to go into ToLoanList
            MyResult3 = self.MyCursor.fetchall()

            # Populate ToLoanList with the results of the SQL query
            for item in MyResult3:
                ListItem = QListWidgetItem(str(item[1]))
                # setData attaches a hidden value to each list item which can be accessed by
                  calling data() on the item object
                # In this case, the program attaches the primary key of each item to its
                  corresponding list item
                ListItem.setData(Qt.UserRole, item[0])
                self.ToLoanList.addItem(ListItem)

        # Repopulate tuple as list
        MyResult2 = [item[0] for item in MyResult2]
        # If no items have been loaned out, don't proceed with populating AlreadyLoanedList
        if len(MyResult2) == 0:
            pass
        else:
            # Get all the entries into the Loans table which correspond with the current
              collection
            self.MyCursor.execute("SELECT * FROM Loans WHERE FK_Collections_Loans = " +
            str(self.OpenCollectionID))
            MyResult5 = self.MyCursor.fetchall()
            # Iterate through query result
            for x in range(0, len(MyResult5)):
                # For each item in the query result, get the corresponding entry into the
                  collection's table
                self.MyCursor.execute("SELECT * FROM " + self.OpenCollectionTable + " WHERE
                PK_" + self.OpenCollectionTable + " = " + str(MyResult5[x][3]))
                MyResult6 = self.MyCursor.fetchall()
                # Create a list item to enclose the custom list widget
                ListItem = QListWidgetItem(self.AlreadyLoanedList)
                # Create an instance of CustomListItem, passing in the results of the SQL
                  query as parameters
                ListWidgetContents = CustomListItem(Name = MyResult6[0][1], DueDate =
                MyResult5[x][1].strftime("%d/%m/%Y"), ID = MyResult5[x][0])
                # Connect the CancelLoanSignal emitted by CustomListItem when CancelLoanBTN
                  is pressed to the corresponding slot
                ListWidgetContents.CancelLoanSignal.connect(self.CancelLoanSlot)
                ListItem.setSizeHint(ListWidgetContents.minimumSizeHint())
                # Set the layout of the container ListItem to the instance of CustomListItem
                self.AlreadyLoanedList.setItemWidget(ListItem, ListWidgetContents)

    # This slot is executed when CancelLoanSignal is emitted
    # It takes the parameter Sender which is the instance of CustomListItem which emitted the
      signal
    def CancelLoanSlot(self, Sender):
        # Delete the corresponding item from the Loans SQL table
        self.MyCursor.execute("DELETE FROM Loans WHERE PK_Loans = " + str(Sender.ID))
        # Save changes to database
        self.MyDB.commit()
        # Repopulate lists
        self.PopulateLists()

    # This slot is executed when an item is selected in ToLoanList
    def ItemSelected(self):
        # If no items are selected, hide PickDueDate, OptionsWidget and SubmitLoanBTN
        if len(self.ToLoanList.selectedItems()) == 0:
            self.PickDueDate.hide()
            self.SubmitLoanBTN.hide()
            self.OptionsWidget.hide()
        # If an item in the list has been selected, show these widgets
        else:
            self.PickDueDate.show()
            self.SubmitLoanBTN.show()
            self.OptionsWidget.show()

    # This method is executed when the user presses SubmitLoanBTN
```

```
    # It writes the new loan to the Loans table
    def SubmitLoanMethod(self):
        # Get the primary key corresponding with the item selected in ToLoanList
        SelectedItemID = self.ToLoanList.selectedItems()[0].data(Qt.UserRole)
        # Get the due date the user has inputted
        DueDate = self.PickDueDate.date().toPyDate()
        # Insert new entry into Loans table
        self.MyCursor.execute("INSERT INTO Loans (DueDate, FK_Collections_Loans,
KeyInCorrespondingTable, Email, Push) VALUES ('" + str(DueDate) + "', " +
str(self.OpenCollectionID) + ", " + str(SelectedItemID) + ", " +
str(self.EmailNotificationCB.isChecked()) + ", " + str(self.PushNotificationCB.isChecked()) +
")")
        # Save changes to database
        self.MyDB.commit()
        # Repopulate lists
        self.PopulateLists()

    def sizeHint(self):
        return QSize(910, 505)

    def paintEvent(self, event):
        Painter = QPainter()
        Painter.begin(self)
        Outline = QPixmap()
        Outline.load('Resources/LoanOutline.png')
        Painter.drawPixmap(QPoint(0, 0), Outline)
        Painter.end()

# This object is used to display items in AlreadyLoanedList
# Unlike standard list items, this widget can display multiple lables and buttons
class CustomListItem(QWidget):
    # This signal is emitted when CancelLoanBTN is pressed
    CancelLoanSignal = pyqtSignal(QObject)
    def __init__(self, Name, DueDate, ID):
        super(CustomListItem, self).__init__()

        # Reassign arguments as attribute
        # ID is the primary key of the item
        self.ID = ID
        # Name is the name of the item
        self.Name = Name
        # DueDate is the due date of the item
        self.DueDate = DueDate

        # Initialise horizontal layout
        self.RowHBL = QHBoxLayout()
        self.RowHBL.setContentsMargins(0, 0, 0, 1)
        self.setLayout(self.RowHBL)

        # If the length of the item's name is greater than 26 characters, shorten it and place
          ellipses at the end of the string
        if len(Name) >= 26:
            self.Name = self.Name[:26] + "..."

        # Create a label displaying the item's name and add it to the layout
        self.NameLBL = QLabel(self.Name)
        self.NameLBL.setFixedWidth(213)
        self.NameLBL.setContentsMargins(5, 0, 0, 0)
        self.RowHBL.addWidget(self.NameLBL)

        # Create a label displaying the item's due date and add it to the layout
        DateLBL = QLabel(self.DueDate)
        DateLBL.setFixedWidth(213)
        DateLBL.setContentsMargins(0, 0, 0, 0)
        self.RowHBL.addWidget(DateLBL)

        # CancelLoanBTN is used to delete loans
        self.CancelLoanBTN = QPushButton("Cancel Loan")
        self.CancelLoanBTN.setContentsMargins(0, 0, 0, 0)
        # Style button
```

```
        self.CancelLoanBTN.setStyleSheet("""
        QPushButton:pressed { color: black; }
        QPushButton:hover:!pressed { color: #D5E8D4; }
        QPushButton { border: 0px; background-color: rgba(0, 0, 0, 0); font-size: 13px; color:
        #96BA8A }
        """)
        ButtonFont = QFont()
        ButtonFont.setUnderline(True)
        self.CancelLoanBTN.setFont(ButtonFont)
        # Connect button's clicked signal so that it emits CancelLoanSignal when pressed
        self.CancelLoanBTN.clicked.connect(lambda: self.CancelLoanSignal.emit(self))
        self.RowHBL.addWidget(self.CancelLoanBTN)

        self.RowHBL.addStretch()
```

LoanChecker.py

```
import smtplib, ssl

try:
    MyDB = mysql.connector.connect(
        host="localhost",
        user="root",
        password="B4tm4nisbae",
        database="Anthology"
    )
    MyCursor = MyDB.cursor()
    Connected = True
except:
    sys.exit(1)

# Get current date
CurrentTime = datetime.date.today()
# List for storing all loans which are overdue
LoansRequiringAction = []
# Get all entries into Loans table
MyCursor.execute("SELECT * FROM Loans")
MyResult1 = MyCursor.fetchall()
# Iterate through query results
for item in MyResult1:
    # If an item's due date is older than or the same as the current date, add it to the list
    if item[1] <= CurrentTime:
        LoansRequiringAction.append(item)

# Iterate through the list of overdue loans
for item in LoansRequiringAction:
    # Get conents of table that current item belongs to
    MyCursor.execute("SELECT * FROM Table" + str(item[2]) + " WHERE PK_Table" + str(item[2]) + "
    = " + str(item[3]))
    MyResult2 = MyCursor.fetchall()
    # Get entry into Collections table of collection which current item belongs to
    MyCursor.execute("SELECT * FROM Collections WHERE PK_Collections = " + str(item[2]))
    MyResult3 = MyCursor.fetchall()
    # Get the entry into the Users table for the user which the collection belongs to
    MyCursor.execute("SELECT * FROM Users WHERE PK_Users = " + str(MyResult3[0][3]))
    Recipient = MyCursor.fetchall()[0][1]
    # If the boolean value for Push notifications is set to true...
    if item[5]:
        # Display a notification, formatting the appropriate data in
        NotificationString = str(MyResult2[0][1]) + " is due today"
        os.system("""
        osascript -e 'display notification "{0}" with title "{1}"'
        """.format(NotificationString, "Anthology"))
    # If the boolean value for Email notifications if set to true...
    if item[4]:
        # The password for the dev email which sends emails to the user is encoded in base64 so
          people can't get the credentials for the account by looking at the source code
        EncodedCredential = b'c0B0ZWxpdGUyMDAz'
```

```
        # Decode base64
        # Begin by converting encoded byte object into utf-8
        ToString = EncodedCredential.decode("utf-8")
        # Then encode string into ascii
        Base64Bytes = ToString.encode('ascii')
        # Then decode string to get a byte object
        CredentialBytes = base64.b64decode(Base64Bytes)
        # Finally, decode binary object into ascii
        Credential = CredentialBytes.decode('ascii')

        # This text will make up the body of the email
        EmailText = """Subject: Item Due

The item '{0}' which you loaned out from your {1} collection is due back today

From the Anthology team
        """.format(str(MyResult2[0][1]), str(MyResult3[0][1]))

        # Start email server instance, using port 465
        with smtplib.SMTP_SSL("smtp.gmail.com", port = 465, context =
        ssl.create_default_context()) as Server:
            # Login to dev email account
            Server.login("sami.development2003@gmail.com", Credential)
            # Send email
            Server.sendmail("sami.development2003@gmail.com", Recipient, EmailText)
            # Quit server instance
            Server.quit()

    # Delete entry into Loans item once notification has been sent
    MyCursor.execute("DELETE FROM Loans WHERE PK_Loans = " + str(item[0]))
    # Save changes to database
    MyDB.commit()
```

ExportCollection.py

```
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
import os
from tabulate import tabulate
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive

# This object creates a submenu containing two new tool buttons
# One is for exporting the current collection as a TXT file, the other is for uploading the
  collection to Google Drive
class ExportSubMenu(QWidget):
    # This signal is emitted when the program is unable to export the collection
    Failed = pyqtSignal()
    # This signal is emitted when the program successfully exports the collection
    Success = pyqtSignal()
    def __init__(self, MyCursor, MyDB, ActiveUserID, OpenCollectionID, OpenCollectionTable,
    widget = None):
        super(ExportSubMenu, self).__init__()

        # Reassign arguments as attributes
        self.ActiveUserID = ActiveUserID
        self.MyCursor = MyCursor
        self.MyDB = MyDB
        self.OpenCollectionID = OpenCollectionID
        self.OpenCollectionTable = OpenCollectionTable

        # Set window attributes
        self.setWindowFlags(Qt.WindowFlags(Qt.FramelessWindowHint | Qt.WindowStaysOnTopHint |
        Qt.Popup))
        self.setAttribute(Qt.WA_QuitOnClose, False)
        self.setAttribute(Qt.WA_TranslucentBackground)
```

```
        # Apply stylesheet to window
        with open("Stylesheet/Stylesheet.txt", "r") as ss:
            self.setStyleSheet(ss.read())

        # Initialise main layout
        self.ExportHBL1 = QHBoxLayout()
        self.setLayout(self.ExportHBL1)

        # Create button for exporting as TXT file
        self.ExportToTxtBTN = QToolButton()
        self.ExportToTxtBTN.setText("Export as TXT\nfile")
        # Set button icon
        self.ExportToTxtBTN.setIcon(QIcon("Resources/ExportTXT.png"))
        self.ExportToTxtBTN.setIconSize(QSize(45, 45))
        self.ExportToTxtBTN.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)
        self.ExportToTxtBTN.setFixedWidth(90)
        self.ExportHBL1.addWidget(self.ExportToTxtBTN)
        # Bind button to function with the string parameter "Local" to indicate the user wants a
          local export rather than a Google Drive export
        self.ExportToTxtBTN.clicked.connect(lambda: self.ExportToTXT(type = "Local"))

        # Create button for uploading collection to Google Drive
        self.ExportGDriveBTN = QToolButton()
        self.ExportGDriveBTN.setText("Upload to \nGoogle Drive")
        # Set button icon
        self.ExportGDriveBTN.setIcon(QIcon("Resources/GDriveIcon.png"))
        self.ExportGDriveBTN.setIconSize(QSize(45, 45))
        self.ExportGDriveBTN.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)
        self.ExportGDriveBTN.setFixedWidth(90)
        self.ExportHBL1.addWidget(self.ExportGDriveBTN)
        # Bind button to function with string parameter "GDrive" to indicate the user wants to
          upload the export to their Google Drive
        self.ExportGDriveBTN.clicked.connect(lambda: self.ExportToTXT(type = "GDrive"))

        # Calculate window position
        x = widget.mapToGlobal(QPoint(0, 0)).x()
        y = widget.mapToGlobal(QPoint(0, 0)).y() + widget.height()
        self.move(x, y)

        # Show window
        self.show()

    # This method is called when either of the buttons are pressed
    # The parameter type tells the program whether the user wants to create a local export or
      upload the export to their Google Drive
    def ExportToTXT(self, type):
        # Get the collection's column names
        self.MyCursor.execute("SHOW FIELDS FROM " + self.OpenCollectionTable)
        MyResult1 = self.MyCursor.fetchall()
        MyResult1 = [item[0] for item in MyResult1]
        # Remove first and last elements from list to get rid of primary key and thumbnails
          columns
        MyResult1 = MyResult1[:-1]
        del MyResult1[0]

        # Construct a SQL statement for selecting the data under all columns in the table except
          for the primary key and thumbnails
        SQLStatement = "SELECT "
        for x in range(0, len(MyResult1) - 1):
            SQLStatement += MyResult1[x] + ", "
        SQLStatement += MyResult1[len(MyResult1) - 1] + " FROM " + self.OpenCollectionTable
        # Execute SQL command
        self.MyCursor.execute(SQLStatement)
        MyResult2 = self.MyCursor.fetchall()
        # Use tabulate library to format query results as an ascii table
        ExportTable = tabulate(MyResult2, headers = MyResult1, tablefmt = "fancy_grid")

        # If the user wants to export locally (they pressed ExportToTxtBTN)...
        if type == "Local":
            # Create a file dialog instance, allowing the user to select the director they would
```

```
            like to export to
            self.DirectoryName = QFileDialog.getExistingDirectory(None, "Select Export
            Directory", os.path.abspath(os.sep))
            # Try/Except clause provides error checking
            try:
                # Create a new file in the user-specified directory called "AnthologyExport.txt"
                ExportFile = open(self.DirectoryName + "/AnthologyExport.txt", "w")
                # Write the contents of ExportTable to the new file
                ExportFile.write(ExportTable)
                # Close the file to prevent corruption
                ExportFile.close()
                # Emit signal, this should be received by the parent class
                  (OpenCollectionWindow) and should display a message in the status bar
                  informing the user that the export was successful
                self.Success.emit()
            except:
                # If there is an issue during the export, emit the Failed signal which will also
                  display a message in the StatusBar when received
                self.Failed.emit()
        # If the user wants to upload the collection to their Google Drive (they pressed
          ExportGDriveBTN)...
        else:
            # Try/Except clause once again provides error checking
            try:
                # Create an object to handle authentication
                GoogleLogin = GoogleAuth()
                # Opens browser and displays login screen
                GoogleLogin.LocalWebserverAuth()
                # Create Google Drive object to handle creating and uploading files
                Drive = GoogleDrive(GoogleLogin)

                # Create a file in the Temp directory
                ExportFile = open("Temp/AnthologyExport.txt", "w")
                # Write the contents of ExportTable to the new file
                ExportFile.write(ExportTable)
                ExportFile.close()

                # Open the file in the Temp directory
                with open("Temp/AnthologyExport.txt", "r") as file:
                    # Create a new drive file object
                    FileForUpload = Drive.CreateFile({"title":os.path.basename(file.name)})
                    # Set the contents of the drive file to the contents of the file in the Temp
                      directory
                    FileForUpload.SetContentString(ExportTable)
                    # Upload the file to Google Drive
                    FileForUpload.Upload()

                # delete the file in the Temp directory as it is no longer needed
                os.remove("Temp/AnthologyExport.txt")
                # Emit Success signal
                self.Success.emit()
            # If there is an issue whilst uploading the file to the cloud, execute the following
              code
            except:
                # Delete the file in the temp directory if it exists
                try:
                    os.remove("Temp/AnthologyExport.txt")
                except:
                    pass
                # Emit Failed signal
                self.Failed.emit()

    def sizeHint(self):
        return QSize(232, 132)

    def paintEvent(self, event):
        Painter = QPainter()
        Painter.begin(self)
        Outline = QPixmap()
        Outline.load('Resources/ExportOutline.png')
```

```
        Painter.drawPixmap(QPoint(0, 0), Outline)
        Painter.end()
```

Stylesheet

```
* {
        background: #FFFFFF;
        color: #000000;
        border: 1px solid #5A5A5A;
}

QWidget#BorderlessWidget {
    border: 0px;
}

QWidget::item:selected {
        background: #D5E8D4;
}

QCheckBox, QRadioButton {
        border: none;
}

QCheckBox::indicator:checked {
    image: url(Stylesheet/Tick.png);
}

QRadioButton::indicator:checked {
    image: url(Stylesheet/Radio.png)
}

QGroupBox {
        margin-top: 6px;
}

QGroupBox::title {
        top: -7px;
        left: 7px;
}

QScrollBar {
        border: 1px solid #5A5A5A;
        background: #FFFFFF;
}

QScrollBar:horizontal {
        height: 10px;
        margin: 0px;
}

QScrollBar:vertical {
        width: 10px;
        margin: 0px;
}

QScrollBar::handle {
        background: #D5E8D4;
        border: 1px solid #5A5A5A;
}

QScrollBar::handle:horizontal {
        border-width: 0px 1px 0px 1px;
}

QScrollBar::handle:vertical {
        border-width: 1px 0px 1px 0px;
}

QScrollBar::handle:horizontal {
        min-width: 20px;
}

QScrollBar::handle:vertical {
        min-height: 20px;
}

QScrollBar::add-line:horizontal {
        height: 0px;
        width: 0px;
    border: none;
        background: none;
}

QScrollBar::add-line:vertical {
        height: 0px;
        width: 0px;
        border: none;
        background: none;
}

QScrollBar::sub-line:horizontal {
        height: 0px;
        width: 0px;
    border: none;
        background: none;
}

QScrollBar::sub-line:vertical {
        height: 0px;
        width: 0px;
    border: none;
        background: none;
}

QScrollBar::add-page, QScrollBar::sub-page {
        background: none;
        height: 0px;
}

QAbstractButton:hover {
        background: #96BA8A;
}

QAbstractButton:pressed {
        background: #ECECEC;
}

QAbstractItemView {
        show-decoration-selected: 1;
        selection-background-color: #D5E8D4;
        selection-color: #D5E8D4;
        alternate-background-color: #D5E8D4;
}
QRadioButton::indicator, QCheckBox::indicator {
        width: 13px;
        height: 13px;
}
QRadioButton::indicator::unchecked,
QCheckBox::indicator::unchecked {
        border: 1px solid #5A5A5A;
        background: none;
}
QRadioButton::indicator:unchecked:hover,
QCheckBox::indicator:unchecked:hover {
        border: 1px solid #000000;
}
QRadioButton::indicator::checked,
QCheckBox::indicator::checked {
        border: 1px solid #5A5A5A;
        background: #D5E8D4;
}
```

```
QHeaderView {
        border: 1px solid #5A5A5A;
        border-radius: 0px;
        border-bottom: 0px;
        border-left: 0px;
        border-top: 0px;
        font-size: 12pt;
}

QHeaderView::section {
        background: #FFFFFF;
        border: 1px solid #5A5A5A;
        border-right: 0px;
}

QHeaderView#hHeader::section {
    border-top: 2px solid rgb(90,90,90);
        border-right: 0px;
        border-bottom: 2px solid rgb(90,90,90);
        font-weight: bold;
        background: #D5E8D4;
}

QHeaderView#vHeader::section{
    border-left: 0px;
    border-bottom: 0px;
        border-right: 0px;
        padding-top: 5px;
}

QTableView {
        gridline-color: #5A5A5A;
        selection-color: #D5E8D4;
        selection-background-color: #D5E8D4;
        border-top: 0px;
        border-right: 0px;
}

QTableView QScrollBar {
    border-bottom: 0px;
}

QTableView QTableCornerButton::section {
    background: white;
}

QTableWidget::item {
    border-bottom: 1px solid #d6d9dc;
    padding-left: 5px;
}

QTableView::QScrollBar:vertical{
    background: #FFFFFF;
    border-bottom: 0px;
    border-top: 0px;
}

QTabBar {
        margin-left: 2px;
}

QTabBar::tab {
        border-radius: 0px;
        padding: 4px;
        margin: 4px;
}

QTabBar::tab:selected {
        background: #D5E8D4;
}

QAbstractSpinBox {
        padding-right: 15px;
}

QAbstractSpinBox::up-button, QAbstractSpinBox::down-
button {
        border: 1px solid #5A5A5A;
        background: #D5E8D4;
        subcontrol-origin: border;
}

QAbstractSpinBox::up-arrow {
        width: 7px;
        height: 7px;
        border: 0px solid #5A5A5A;
        image: url(Stylesheet/Up.png);
}

QAbstractSpinBox::down-arrow {
        width: 7px;
        height: 7px;
        border: 0px solid #5A5A5A;
        image: url(Stylesheet/Down.png);
}

QSlider {
        border: none;
}

QSlider::groove:horizontal {
    border: 1px solid;
        height: 2px;
        margin: 0px;
}

QSlider::groove:vertical {
        width: 5px;
        margin: 0px 4px 0px 4px;
}

QSlider::handle {
        border: 1px solid #5A5A5A;
        background: #D5E8D4;
}

QSlider::handle:horizontal {
        width: 10px;
        height: 40px;
        margin: -5px 0px;
}

QSlider::handle:vertical {
        height: 15px;
        margin: 0px -4px 0px -4px;
}

QSlider::add-page:vertical, QSlider::sub-page:horizontal
{
        background: #D5E8D4;
}

QSlider::sub-page:vertical, QSlider::add-page:horizontal
{
        background: #FFFFFF;
}

QLabel {
        border: none;
        background-color: rgba(0,0,0,0%);
}

QProgressBar {
        text-align: center;
}

QProgressBar::chunk {
        width: 1px;
        background-color: #D5E8D4;
}
```

```
QMenu::separator {
        background: #D5E8D4;
}

QPushButton {
    background: #D5E8D4
}

QToolButton {
    background: #D5E8D4
}

QComboBox::down-arrow {
        border: 0px solid #5A5A5A;
        background: #D5E8D4;
    width: 7px;
        height: 7px;
        image: url(Stylesheet/Down.png);
}

QComboBox::drop-down {
        border: 1px solid #5A5A5A;
        background: #D5E8D4;
}

QComboBox {
    margin-top: 0px;
    padding-left: 5px;
    padding-bottom: 0px;
}

QCalendarWidget QToolButton {
        height: 20px;
        width: 150px;
        color: black;
        font-size: 14px;
        icon-size: 30px, 30px;
        background-color: #D5E8D4;
        border-bottom: 0px;
}

QCalendarWidget QMenu {
    width: 150px;
    left: 20px;
    color: black;
    font-size: 14px;
    background-color: #FFFFFF;
}

QCalendarWidget QSpinBox {
    width: 110px;
    font-size: 14px;
    color: black;
    background-color: #D5E8D4;
    selection-background-color: rgb(136, 136, 136);
    selection-color: rgb(255, 255, 255);
}

QCalendarWidget QWidget {
    alternate-background-color: #ECECEC;
}

QCalendarWidget QAbstractItemView:enabled {
    font-size: 14px;
    color: black;
    background-color: white;
    selection-background-color: #D5E8D4;
    selection-color: white;
}

QCalendarWidget QWidget#qt_calendar_navigationbar {
    background-color: #D5E8D4;
    border-bottom: 0px;
}

QCalendarWidget QAbstractItemView:disabled{
    color: #a9a9a9;
}

QMessageBox QPushButton {
    width: 80px;
}

QAbstractButton#CollectionButton {
    width: 140px;
    height: 100px;
    border-radius: 10%;
    font-size: 14px;
}

QAbstractScrollArea#TopBorderRemoved {
    border-top: 0px;
}

QWidget#ContainerWidget {
    background-color: #D5E8D4;
    border-top: 1px solid #5A5A5A;
    border-left: 1px solid #5A5A5A;
    border-right: 1px solid #5A5A5A;
    border-bottom: 0px;
}
```

Bibliography

[1] C. Jarrett, “Why do we collect things? Love, anxiety or desire | Life and style | The Guardian,” 2014. [Online]. Available: https://www.theguardian.com/lifeandstyle/2014/nov/09/why-do-we-collect-things-love-anxiety-or-desire.

[2] B. A. Lafferty, E. Matulich and M. X. Liu, "Exploring Worldwide Collecting Consumption Behaviours," 2013. [Online]. Available: https://www.aabri.com/manuscripts/131634.pdf.

[3] D. Sachnev, “Debian Bug report logs - #787085,” 28 May 2015. [Online]. Available: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=787085.

[4] “Trustpilot Collectorz Reviews,” [Online]. Available: https://uk.trustpilot.com/review/www.collectorz.com?languages=en.

[5] 2018. [Online]. Available: https://github.com/BlueBrain/neurocurator/issues/56.

[6] 2017. [Online]. Available: https://lists.qt-project.org/pipermail/interest/2017-January/025727.html.

[7] 2018. [Online]. Available: https://github.com/marinecoin/marinecore/issues/3.