

**Richard Jones**

A Level Computer Science Coursework

Music Centre Program

Centre Number – 52423

Candidate Number - 2389

Module - H446/03~04

Draft

Authors : Richard Jones

Date : 7 February 2017

Version : V0.1

E-mail : richardj99@sky.com

Contents

[1 Analysis 4](#_Toc482039187)

[1.1 Suitability for Solubility by computational Methods 4](#_Toc482039188)

[1.1.1 Description of Relevant Features of problem 4](#_Toc482039189)

[1.1.2 Why is this problem Suitable for solution by computational Methods? 4](#_Toc482039190)

[1.2 Stakeholder Management 5](#_Toc482039191)

[1.2.1 Who are the stakeholders and how were they chosen? 5](#_Toc482039192)

[1.2.2 How will they make use of the proposed solution? 5](#_Toc482039193)

[1.2.3 Why will the approach taken be suitable for stakeholder needs 6](#_Toc482039194)

[1.2.4 Research into Alternative Solutions 6](#_Toc482039195)

[1.2.5 Minimum Essential features required 8](#_Toc482039196)

[1.2.6 Limitation of Minimum Features 10](#_Toc482039197)

[1.2.7 Identification of proposed solution with reference to Research 10](#_Toc482039198)

[1.2.8 Hardware and software requirements for solution 10](#_Toc482039199)

[2 Design 12](#_Toc482039200)

[2.1 Breakdown of Problem 12](#_Toc482039201)

[2.1.1 Problem Elements suitable for computational solutions 12](#_Toc482039202)

[2.1.2 Justifying the process 12](#_Toc482039203)

[2.2 Detailed Structure of Solution to be developed 13](#_Toc482039204)

[2.2.1 User Interface Design 13](#_Toc482039205)

[2.2.2 Database Design 19](#_Toc482039206)

[2.2.3 Algorithms required in solution 20](#_Toc482039207)

[2.2.4 UML Diagram 20](#_Toc482039208)

[2.3 Approach to Testing 21](#_Toc482039209)

[2.3.1 Test Data 21](#_Toc482039210)

[2.3.2 Test Plan 21](#_Toc482039211)

[3 System Development 26](#_Toc482039212)

[3.1 Database Initialisation 26](#_Toc482039213)

[3.1.1 Problem Decomposition 26](#_Toc482039214)

[3.1.2 Creating the Tables 26](#_Toc482039215)

[3.1.3 Creating Relationships 29](#_Toc482039216)

[3.1.4 Loading the Test Data 30](#_Toc482039217)

[3.1.5 Final View 30](#_Toc482039218)

[3.2 Class-less code 31](#_Toc482039219)

[3.2.1 Imported Libraries 31](#_Toc482039220)

[3.2.2 Linking the Database with the code 31](#_Toc482039221)

[3.2.3 Connecting the User Interface Files 31](#_Toc482039222)

[3.3 Login Screen Class 31](#_Toc482039223)

[3.3.1 Project Decomposition 31](#_Toc482039224)

[3.3.2 1st Iteration 32](#_Toc482039225)

[3.3.3 Testing for Iteration 01 33](#_Toc482039226)

[3.3.4 2nd Iteration 38](#_Toc482039227)

[3.3.5 Testing for Iteration 02 39](#_Toc482039228)

[3.4 Create Account Window Class 41](#_Toc482039229)

[3.4.1 Project Decomposition 41](#_Toc482039230)

[3.4.2 Final Code 41](#_Toc482039231)

[3.5 Main Window Class 43](#_Toc482039232)

[3.5.1 Project Decomposition 43](#_Toc482039233)

[3.5.2 Iterative Testing 44](#_Toc482039234)

[3.5.3 Final Code 46](#_Toc482039235)

[3.6 Settings Window Class 52](#_Toc482039236)

[3.6.1 Project Decomposition 52](#_Toc482039237)

[3.6.2 Final Code 53](#_Toc482039238)

[3.7 Playlist Manager Window Class 56](#_Toc482039239)

[3.7.1 Project Decomposition 56](#_Toc482039240)

[3.7.2 Final Code 57](#_Toc482039241)

[3.8 Playlist Name Dialog Box Class 60](#_Toc482039242)

[3.8.1 Project Decomposition 60](#_Toc482039243)

[3.8.2 Final Code 60](#_Toc482039244)

[4 Evaluation 62](#_Toc482039245)

[4.1 Final Testing 62](#_Toc482039246)

[4.1.1 Database 62](#_Toc482039247)

[4.1.2 User Interface 64](#_Toc482039248)

[4.1.3 Login Screen 67](#_Toc482039249)

[4.1.4 Provide an input box where users can enter their username 67](#_Toc482039250)

# Analysis

## Suitability for Solubility by computational Methods

### Description of Relevant Features of problem

From the late 1800s, music has been able to be recorded and distributed through many different media. For most of the 20th century, music was mainly released on either vinyl or cassette formats. However, as computers began to become more and more commonplace in households and society as a whole, digital solutions for storing recorded music have been released. With this advancement, the music needs to be organised (similar to a music library of physical music). A physical music library is normally sorted by a category, such as the record’s artist name, album name or song name. Usually this order is done by hand and is a laborious task. This organisation is done to allow people to easily and quickly navigate their library in search for a specific record. Alongside their library, music listeners also keep a music player of some kind. This could be anything from a vinyl record turntable to a CD player. In order to cater for this feature of the problem, I will need to make sure that the solution can play this music.

However, the idea of a music player program is no new concept. Many other companies have managed to create monumentally successful programs that organise music libraries and play music. Later, I will conduct research into similar solutions to gain a further understanding of how I can make my proposed solution more complex and relatable to consumer requirements.

I have been approached by Samuel Barrett, one of my fellow students, who expressed his own opinions about a music player program. He felt that many of the other music playing programs on the market today have become too complex in hopes to achieve a wider target market. In doing so, the programs have become too confusing to use and are becoming over encumbering on the machines that they are being run on. From this conversation, I decided to aim to create a solution that is lightweight and simple in nature. I also made Samuel Barrett one of my end users for the solution and a stakeholder in the finished product.

Though I am taking ideas from several previous solutions to this problem. It is important to understand that I do not want this program to be a main competitor in this market. This solution is not meant to be a fully official and profitable attempt, it is just a prototyped solution that will aim to solve the basic problems mentioned earlier.

### Why is this problem Suitable for solution by computational Methods?

The solution can be solved by computational methods for a multitude of reasons:

First, the library organisation. Using a database and tag reading, information about the songs (including their artist, and album) can be stored. Then, using a selection of methods, which are yet to be decided on (such as whether to use search queries or iterative table searching), a view similar to a physical music library can be customised. In fact, using this method, the library can be improved upon. If somebody wanted to change the view of their library, say from sorted by artist to sorted by album, they would have to physically reorder their records which could be physically strenuous and time consuming. Using a computer, the exact same task can be done by clicking a button.

Second, searching for a song, album or artist would be even quicker. In a physical music library, the user would have to manually scan the library for the record. While this could be quickened with an index, it would not reach the speed that it could be searched for using a computer. Using either an iterative search function or an SQL query, the record could be found in a matter or milliseconds.

Third, music can be played using computational methods. Audio files exist for computers and can be played very easily. Many operating systems even come built in with integrated music players. By simply recording the location of each music file, the file can be played instantly if the correct codec is used. This would be very efficient compared to the physical solution, where the record would have to be taken from the library and manually placed in the CD player or turntable.

In conclusion, many improvements can be made through solving the problem by computational methods. Making the overall task of listening and organising music more enjoyable and less strenuous to the user.

## Stakeholder Management

### Who are the stakeholders and how were they chosen?

There could be many potential stakeholders for this solution as most of the population have a mutual enjoyment of music and would be interested in the media player program described.

Many members of my tutor group have expressed a deep interest in music and would be suitable stakeholders in this project. As I have already mentioned, Samuel Barrett has already expressed a deep interest in my program, and hopes to see it to be a successful, yet lightweight solution to the problem. I aim to choose stakeholders who are genuinely interested in the program’s wellbeing and who will give detailed yet constructive criticism to ensure its success.

To collate as wide a target market as possible, I have randomly selected six students to be my stakeholders. These people will originate from my tutor group, my school’s staff and my family. They will express their wants and needs out of a solution to this problem and will be consulted throughout the design, development and testing process of this program.

### How will they make use of the proposed solution?

Several studies (and my own personal experience) have shown that people tend to listen to music while they are working and relaxing, either at home or at a workplace. I aim for my solution to be used in these situations.

The solution will be in the form of an application that people can keep open on their desktop, allowing them to concentrate on work or relax while using the solution.

Furthermore, I have conducted several interviews with the stakeholders to create a more specific picture of what my solution needs to satisfy their needs. Here is the survey that I gave to all of them:

|  |
| --- |
| **Name:** |
| **Age:** |
| **Vocation:** |
| **Which program do you currently use to play music?** |
| **What, in your opinion, are the unique strengths of this solution?** |
| **What, in your opinion, are the limitations of this solution?** |
| **What feature would you personally want most out of a solution tailored to your needs?** |

The results from this survey were then collated to allow the data to be viewed and concluded more easily. Here are the results:

* **Which program do you currently use to play music?**
  + Windows Media Player **– 2**
  + ITunes – **3**
  + Other – **1**
* **What, in your opinion, are the unique strengths of this solution?**
  + Wide support for files – **2**
  + Online store – **1**
  + Video Playback – **1**
  + Smart playlists/Genius Playlists - 2
* **What, in your opinion, are the** **limitations of this solution?**
  + Slows down machine – **5**
  + Too many pointless features – **1**
* **What feature would you personally want most out of a solution tailored to your needs?**
  + Lightweight solution/won’t interfere with machine performance – **3**
  + Playlists, both automatically created – **3**

Conclusion

Some conclusions can be made from the data gathered in this interview. First, my selected stakeholders show a great care in how the solution will affect the performance of their machine when it is running.

### Why will the approach taken be suitable for stakeholder needs

As previously mentioned, my solution is going to be a lightweight, yet useful, attempt at the ‘music player’ problem. This will be suitable as many of the stakeholders mentioned how their preferred music players produced issues with the performance of their PC.

Furthermore, my stakeholder interviews have showed only a small number of users are interested in video playback. This could be because of the advancements in video streaming products such as Netflix and Amazon Prime Instant Video. These solutions do not require a player to play their files, it is all included in one system. However, my will to focus on music playback perfectly suits the stakeholders’ needs.

### Research into Alternative Solutions

The main competitors for this solution are Microsoft’s Windows Media Player and Apple’s ITunes.

|  |  |
| --- | --- |
| Product Name: | Windows Media Player |
| Features: | 1. Plays music (with music navigation buttons and progress bar). 2. Oorganises local media files into libraries 3. Allows video and photo playback 4. Allows connection to Network Accessible Devices 5. Provides plugins for cosmetic features such as player themes and visualizations for music. |
| Supported File Formats: | 1. Windows Media Formats (.asf, .wma, .wmv, .wm) 2. Windows Media Metafiles (.asx, .wax, .wvx, .wmx) 3. Windows Media Metafiles (.wpl) 4. Microsoft Digital Video Recording (.dvr-ms) 5. Windows Media Download Package (.wmd) 6. Audio Visual Interleave (.avi) 7. Moving Pictures Experts Group (.mpg, .mpeg, .m1v, .mp2, .mp3, .mpa, .mpe, .m3u) 8. Musical Instrument Digital Interface (.mid, .midi, .rmi) 9. Audio Interchange File Format (.aif, .aifc, .aiff) 10. Sun Microsystems and NeXT (.au, .snd) 11. Audio for Windows (.wav) 12. CD Audio Track (.cda) 13. Indeo Video Technology (.ivf) 14. Windows Media Player Skins (.wmz, .wms) 15. QuickTime Movie file (.mov) 16. MP4 Audio file (.m4a) 17. MP4 Video file (.mp4, .m4v, .mp4v, .3g2, .3gp2, .3gp, .3gpp) 18. Windows audio file (.aac, .adt, .adts) 19. MPEG-2 TS Video file (.m2ts) |
| Support: | 24/7 Help Centre provided by Microsoft |

|  |  |
| --- | --- |
| Product Name: | ITunes |
| Features: | Plays Music, organises local media files into libraries, allows video and photo playback, contains a store where users can buy music and video. |
| Supported Music File Formats: | .mp3, .aiff, .wav, .mpeg, .aac, .m4a, .mov, .aac, .ogg |
| Support: | 24/7 Help Centre provided by Apple |

Conclusion

While both of these solutions have strengths and weaknesses in areas, both provide exemplar solutions to the problem in discussion. The most prominent being Windows Media Player’s unprecedented selection of supported file types and ITune’s online store, providing original features to the problem in question.

However, through research, I discovered that both solutions have had many reports of slowing down the performance of machines, particularly ITunes. While the additional features that they provide are appreciated by many people, it seems they come at a cost in the usability of the program. The main point that my stakeholders brought to my attention in the interview is that they require a solution that has little to no impact on their machine. This means that some features of the solution (including some that are in these solutions) must be ignored to ensure its usability.

It is also appropriate to mention that both solutions are professional attempts, meant for a large-scale audience in a largely commercial environment. Because of this, these solutions are likely to have taken several years to perfect, requiring the help of a large team working on the solution at the same time. My solution is meant for a smaller audience and is scheduled to be done in less than a year, meaning that many of the features must be forgotten to meet the target by the required deadline.

It is also unlikely that my solution will be able to provide the 24/7 support, that these two solutions allow, in the immediate future. To address this, I have decided that the program will be open-source. This will have a very small effect on the solution in question, just that the program will have to be well structured with clear variable names to accommodate third-party development.

### Minimum Essential features required

#### Database

1. Form a normalized database to store information relating to the music.
2. Create a table for the songs.
3. Create a table for the albums.
4. Create a table for the artists.
5. Create a table for the playlists.
6. Create a table that stores information about the program’s users.

#### User Interface

1. Provide a login window
2. Provide a create account window
3. Provide a Media Player window
4. Provide a settings window
5. Provide a playlist manager screen
6. Provide a dialog box to change a playlist’s name.

#### Login Screen

1. Provide an input box where users can enter their username.
2. Provide an input box where users can enter their password.
3. Provide a button that users can use to log in to the program.
4. Validate the username and password against the program’s database.
5. Provide access to the main program if validation is successful.
6. Provide a button that allow the user to navigate to the create account window.
7. Provide a label that notifies the user of any errors and gives instructions.

#### Create Account Screen

1. Provide a text box that allows the user to input their desired username.
2. Provide a text box that allows the user to input their desired password.
3. Provide a button that allows the user to create their account.
4. Validate the username and password using regex to make sure the inputs are long enough, contain numbers and contain letters.
5. Provided the validation is successful, append the details to the database alongside a 0 for the administrator field. Then close the window.

#### Media Player Screen

1. Provide a table that allows the user to navigate the library.
2. When an artist is clicked in the table, show all of the albums by the respective artist in the table.
3. When an album is clicked in the table, show all of the song from the respective album in the table.
4. When a song is clicked, play the respective song.
5. Create a queue of the next songs in the table when a song is clicked.
6. When a playlist is clicked, display all the song from the respective playlist.
7. Provide a button that allows the user to change the view of the library to scrutinize artists.
8. Provide a button that allows the user to change the view of the library to scrutinize albums.
9. Provide a button that allows the user to change the view of the library to scrutinize songs.
10. Provide a button that allows the user to change the view of the library to scrutinize playlists. If the user isn’t an administrator, only show the playlists that are owned by the user or are ownerless (such as the ‘most played playlist’.
11. Provide a button that allows the user to navigate to the settings window.
12. Provide a button that allows the user to navigate to the playlist manager window.
13. Provide a text box that allows the user to search for an item in the database.
14. Provide a dropdown box that allows the user to select what item type they want to search for (e.g. artist, album, song).
15. Provide a button that allows the user to confirm the search.
16. Provide a button that allows the user to exit the application.
17. Provide a button that allows the user to pause the song that is currently being played.
18. Provide a button that allows the user to resume the song if a song is paused.
19. Provide a button that allows the user to skip to the next song in the queue when clicked.
20. Add the previously played song to a stack when said button is activated.
21. Provide a button that allows the user to play the previously played track when clicked.
22. Provide labels that notify the user what song is playing.
23. Provide labels that indicate what artist or album the user is currently viewing in the table.

#### Settings Screen

1. Provide an administrator section that only users with administrator permissions can see.
   1. Provide a table that shows all the program’s users (excluding the current user).
   2. Allows users to be selected from the table.
   3. Provide a label that displays the selected user at a given time.
   4. Provide a button that deletes the selected user from the Users table in the database.
   5. Provide a button that toggles the selected user’s administrator settings in the Users table.
2. Provide an import section:
   1. Provide a text box that allows the user to input a selected directory.
   2. Provide an import button that reads in the user’s input when clicked.
   3. Validate the input to make sure the directory exists.
   4. Scan through the inputted directory and catch any .mp3 files.
   5. Read the relevant information from the ID3 tags of each .mp3 file, e.g. Song name, artist name, album name, genre, song length.
   6. Append the relevant information from the tags into the database, namely the Songs, Artists and Albums tables.
   7. Input N/A if any of the tags are not found.
3. Provide a button that allows the user to exit the settings window when clicked.
4. Provide a button that allows the user to log out of the program, taking them back to the login screen.

#### Playlist Manager Window

1. Provide a table that will display all of the user’s playlists.
2. When a playlist in this table is clicked, songs from the respective playlist will be shown.
3. When a song in this table is clicked, the respective song will be removed from the playlist it belongs to (these changes will also be made in the database).
4. Provide a table that will display all of the songs in the database.
5. When a song in this table is clicked, add the song to the selected playlist provided one is selected (this change will also be made in the database).
6. Provide a button that will deselect the currently selected playlist and display all of the playlists in the playlist tables (see objective 52).
7. Provide a button that will open the new playlist dialog box when clicked.
8. Provide a button that will close this window when clicked.

#### Playlist Dialog Box

1. Provide a text box that will allow the user to input their desired name for the playlist.
2. Provide a button that will allow the user to create their playlist using their input when clicked.
3. Provide a button that will allow the user to close the dialog box when clicked.

### Limitation of Minimum Features

There are several limitations to my proposed solution:

Firstly, my solution will only cater for music playback and not photo or video. This decision was made to reduce the strain that the solution had over the machine it was running on. Better suiting the stakeholder’s requirements. I also feel that it would be an important decision to keep this project focused on music for the near future. This is to make sure that all efforts stays on perfecting the music playing problem and making a better all-round solution. I felt that this would be more suitable than diverting resources to incorporate phot and video.

Secondly, my solution will only support .mp3 files. MP3 files are the most used audio files to date, but there are many other widely used formats which could need support. However, many of these files do not contain the same tags as mp3 (ID3), meaning that my program will not be able to read them into the database. This decision was made to keep the solution as ‘cut down’ as possible as well as reducing the time of development to fit in with the deadline. Better suiting the stakeholder’s needs.

I have personally had thoughts of incorporating access to an internet database of music. This could be utilized in many ways, including finding missing information for the local music files as well as recommending new music to the user depending on what music they listen to more. However, given the time pressures that this project has, I feel that it would be inappropriate to take on this responsibility.

### Identification of proposed solution with reference to Research

Through the research into similar solutions, I discovered that both solutions have had many reports of slowing down the performance of machines, particularly ITunes. While the additional features that they provide are appreciated by many people, it seems they come at a cost in the usability of the program. The main point that my stakeholders brought to my attention in the interview is that they require a solution that has little to no impact on their machine. This means that some features of the solution (including some that are in these solutions) must be ignored to ensure its usability.

It is also appropriate to mention that both solutions are professional attempts, meant for a large-scale audience in a largely commercial environment. Because of this, these solutions are likely to have taken several years to perfect, requiring the help of a large team working on the solution at the same time. My solution is meant for a smaller audience and is scheduled to be done in less than a year, with minimal help from outside sources. This means that many of the features must be forgotten to meet the target by the required deadline.

### Hardware and software requirements for solution

This solution will be programmed in Python 3.4. Python was chosen because it is an object-oriented language, allowing the solution to use models, methods and classes. These constructs will aid the solution’s development because it will allow the problem to be split up into more solvable sections, reducing space and producing a more efficient solution.

Python was also chosen due to its large collection of libraries. Libraries will be used in this solution to read the ID3 tags and allow the playback of songs. The use of these libraries will reduce the amount of time that the solution will be developed, as well as decrease the amount of code in the core program.

# Design

## Breakdown of Problem

### Problem Elements suitable for computational solutions

I aim to decompose the problem down into smaller sub-problems in the hopes that this will make the problem easier to solve. These are the steps that will be taken:

1. First, the application’s User Interface will be designed. The UI will comprise of several windows each serving different purposes. The windows will be the login screen, the create account screen, the media player screen and the settings window. Later in this section, the initial design of these screens will be established by the stakeholders. The UI will be developed in PyQt Designer.
2. After the UI has been designed, the database will be established. This database will hold all the data used by the program, including data relating to the music files (songs, artists, albums and playlists) and data related to the users (usernames and passwords).
3. Once the infrastructure for the application has been implemented, the solution will be coded. This process will start with the coding for the login section. This section will accept account information from the user and validate it in the program’s database. If the details are accepted, the user is admitted into the main program.
4. After the login section, the import function will be programmed in the Settings Window. The function will accept a user specified directory and scan it for any mp3 files, inputting the details from the files into the database.
5. Then the view of the Music Player’s library will be coded. Buttons will be implemented to allow the user to view all the tables in the database (artists, songs, albums etc.) and allow the user to traverse the music library (selecting an artist will retrieve the albums by that artist, selecting an album will retrieve the songs by that artist and selecting a song will play it).
6. The music playback functions will then be coded. This will play a song when it is selected from the view. A pause button will also allow a song to be stopped momentarily. A play button will also be provided to resume a paused song.
7. The queue function will then be coded. The function will add all of the songs under a selected song to a queue. Once the selected song has finished playing, the next song in the queue will be retrieved. This will continue until the queue is empty or a new song is selected.
8. After the queue function has been implemented into the program, the skip buttons will be programmed to traverse the queue backwards and forwards.
9. After the song queueing function has been programmed, the search function will be programmed. The search function will accept a string and use it to search the Music Library for any similar fields in the records. All those that match the User’s input will be outputted in the view. The function will also be able to scrutinize what to search for (over artists, albums, songs and playlists).
10. Then the playlist feature will be coded. The feature will allow users to add new playlists into the database as well as access existing ones. A window can then be accessed that will allow the user to add and remove songs from the playlist.

### Justifying the process

As the previous section has shown. The problem has been decomposed into sub-problems. The process has been done this way to ensure that the problem can be solved in a more simplistic nature. Furthermore, decomposing this problem also shows how it can be solved using computational methods such as modularity and object oriented programming. If each of the screens referred to in the success criteria (e.g. login screen, create account screen, main screen, playlist manager screen and settings screen) was assigned a separate class and therefore a separate object, the solution would become more efficient for a multitude of reasons. Some code could become reusable through the methods of a class and the variables from one class can be kept separate from another, ensuring the that no errors occur by mixing up variables.

## Detailed Structure of Solution to be developed

### User Interface Design

After having a discussion with the stakeholders for this product. I drew up a plan of the interface that everyone could agree on. Slight changes are likely to be made to these in order to improve functionality.

#### **Login Window**

Username:

Password:

Log-In

Create Account

Please Login

|  |  |  |  |
| --- | --- | --- | --- |
| **Object** | **Number** | **Purpose** | **Example Input** |
| txt\_username | 1 | This is a textbox where the user can input their username. | “user1” 🡪 String |
| txt\_pass | 2 | This is a textbox where the user can input their password. | “password1” 🡪 String |
| btn\_login | 3 | This is a button that, once clicked, uses the information that the user has provided and attempts to gain access to the main program. | N/A |
| btn\_crtacc | 4 | This is a button that, once clicked, will take the user to the ‘create account’ window. The user can then proceed to create a new account. | N/A |
| lbl\_username | 5 | This label currently displays ‘Please Login’. However, if any errors are brought up when checking the user’s inputs (e.g. username doesn’t exist in database), a suitable message will be displayed. | N/A |
| lbl\_password | 6 | This label is used to indicate where the user’s password should be entered. | N/A |
| lbl\_info | 7 | This label is used to indicate where the user’s username should be entered. | N/A |

#### **Create Account Window**

Create an Account

Create Account

Password:

Username:

|  |  |  |  |
| --- | --- | --- | --- |
| **Object** | **Number** | **Purpose** | **Example Input** |
| txt\_username | 1 | This object allows the user to input their desired username. | “User1” 🡪 String |
| txt\_password | 2 | This object allows the user to input their desired password. | “Password1” 🡪 String |
| lbl\_username | 3 | This label indicates where the user should enter their username. | N/A |
| lbl\_password | 4 | This label indicates where the user should enter their password. | N/A |
| lbl\_info | 5 | This label initially displays “Create an Account”. However, if any errors are raised when validating the user’s inputs, the label will show a suitable message (e.g. Username has already been taken). | N/A |
| btn\_crtacc | 6 | This button will activate the procedure that takes the user’s inputs and creates an account using them (provided the inputs are successfully validated) | N/A |

#### **Media Player Window (Main Window)**

Songs View

Artists

Albums

Songs

Playlists

Settings

Playlist Manager



Now Playing

Search

|  |  |  |  |
| --- | --- | --- | --- |
| **Object** | **Number** | **Purpose** | **Example Input** |
| Btn\_back | 1 | When this button is activated, it will play the song at the top of the ‘previously playing’ stack. | N/A |
| Btn\_forward | 2 | When this button is activated, it will play the song at the front of the “next song” queue. | N/A |
| Btn\_pause | 3 | When this button is activated, the song that is currently playing will be paused. | N/A |
| Btn\_play | 4 | When this button is activated, any song that has been paused will be resumed. | N/A |
| Lbl\_nowplaying | 5 | Initially, this label will display: “Now Playing:”. However, when a song begins playback, the name of the song will be added to the end of the label. | N/A |
| Btn\_exit | 6 | When this button is activated. The entire program will shut down. | N/A |
| Btn\_search | 7 | When this button is activated, The input from the drop-down menu and the search box will be collected and the music library will then be searched (using these inputs). | N/A |
| Drp\_search | 8 | This drop-down menu allows the user to specify whether they want to search for Albums, Songs, Artists or Playlists | N/A |
| txt\_search | 9 | This object allows the users to input the term that they want to search for in the Music Library. | “ArtistName” 🡪String |
| Lbl\_album | 10 | This label will show which album (if any) is being viewed in the Song View | N/A |
| Lbl\_artist | 11 | This label will show which artist (if any) is being viewed in the Song View | N/A |
| Lbl\_nowviewing | 12 | This label is to help the user understand what the lbl\_artist and lbl\_album objects are denoting | N/A |
| Btn\_albums | 13 | When this button is activated, the song view (tbl\_songs) will show all the albums in the music library | N/A |
| Btn\_artists | 14 | When this button is activated, the song view (tbl\_songs) will show all the artists in the music library. | N/A |
| Btn\_playlists | 15 | When this button is activated, the song view (tbl\_songs) will show all the playlists in the music library. | N/A |
| Btn\_plylstmngr | 16 | When this button is activated, the ‘Playlist Manager’ window will be opened on top of the ‘Main’ Window. | N/A |
| Btn\_settings | 17 | When this button is activated, the ‘Settings’ Window will be opened on top of the ‘Main’ Window | N/A |
| Tbl\_songs | 18 | This table will show all the fields in the libraries database. At a basic level, it will show all of artists, artists, songs and playlists (depending on which button the user selects). However, it will also display albums belonging to a certain artist, songs belonging to a certain album and songs belonging to a certain playlist | N/A |

#### **Settings Window**

Import

Directory:

Users Table

Delete User

Toggle Admin.

Logout

Exit

**User: User1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Object** | **Number** | **Purpose** | **Example Input** |
| Btn\_exit | 1 | When this button is activated, this window will be closed. | N/A |
| Btn\_logout | 2 | When this button is activated, the user will be taken back to the login screen. Any variables that identify the user (e.g. UserID and UserName) will be set to a null value. | N/A |
| Btn\_admin | 3 | When this button is activated, the selected user (the field selected in tbl\_users) will be made an admin or, if they are already an admin, will be made an ordinary user. | N/A |
| Btn\_delUser | 4 | When this button is activated, the selected user (the field selected in tbl\_users) will be deleted from the program and the database. | N/A |
| Lbl\_user | 5 | This object has a null value of “User:”. However, when a user is selected from tbl\_users, the object will also show the name of the user. | N/A |
| Tbl\_users | 6 | This table will show all of the users (from the users table in the database), excluding the user who is currently logged in. | N/A |
| Btn\_import | 7 | When this button is activated, the user’s input (their selected directory) from txt\_dir is taken. The music in this directory is then imported into the database. | N/A |
| Lbl\_dir | 8 | This object will indicate where the user should enter their directory. | N/A |
| Txt\_dir | 9 | This object will allow the user to input the directory from where they want their music to be imported from. | “C:/Users/User/Music/” 🡪 String |

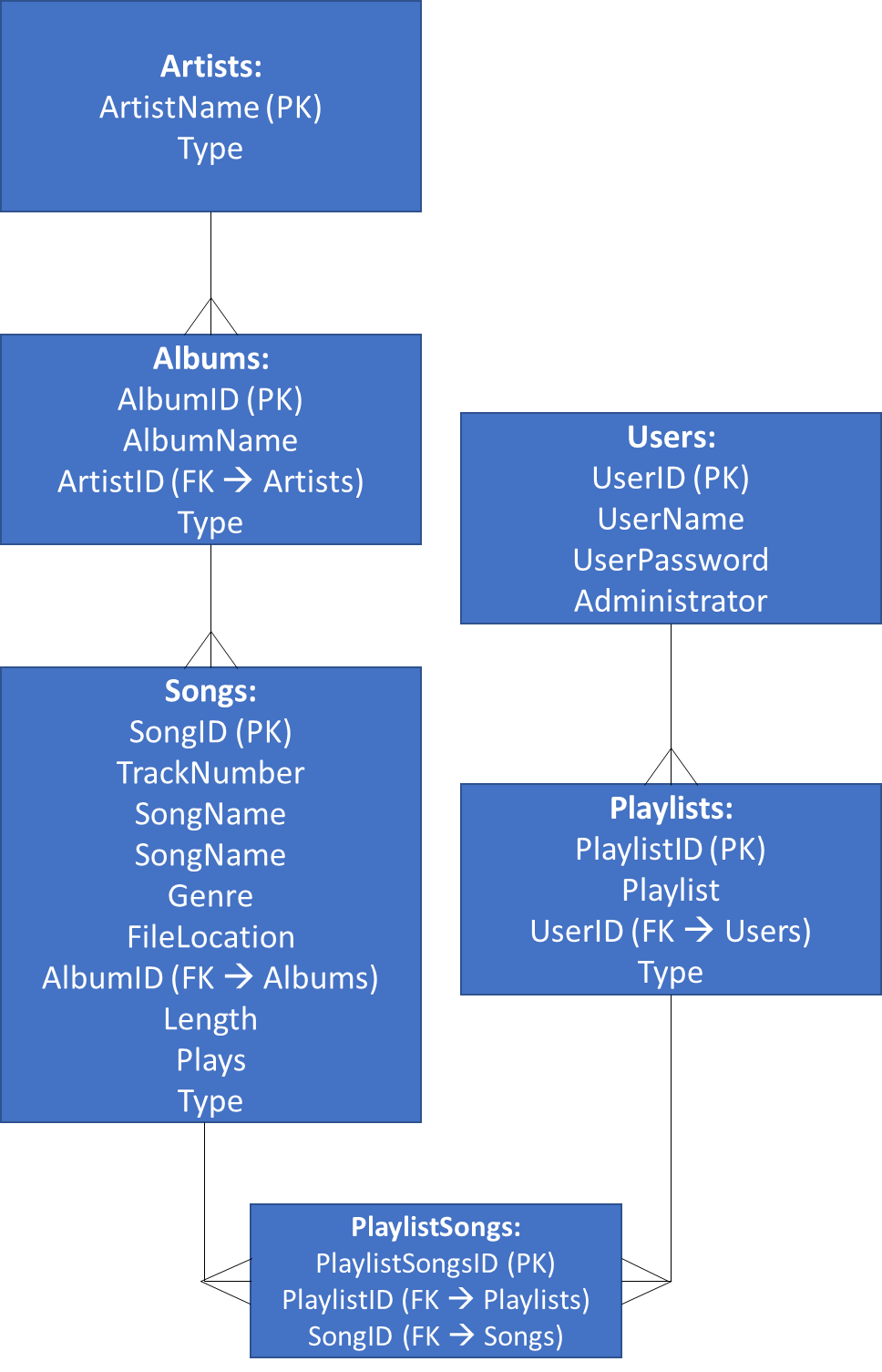
#### **Playlist Manager Window**

|  |  |  |  |
| --- | --- | --- | --- |
| **Object** | **Number** | **Purpose** | **Example Input** |
| Btn\_close | 1 | When this button is activated, the window is closed, taking the user back to the Main Window |  |
| Btn\_newplylst | 2 | When this button is activated, a dialog box will open. Here the user can input the new playlist’s name. |  |
| Btn\_plylstreset | 3 | When this button is activated, th |  |
| Tbl\_playlists | 4 | This table will display all of the information relating to playlist. This includes the names of the playlists and the songs that each playlist contain. |  |
| Tbl\_songs | 5 | This table will show all of the songs that are currently in the music library. |  |

### Database Design

In order to maintain complete and consistent efficiency in my program, I will need to have a normalised database, preferrably in Third Normal Form.

In order to do this, I have decided to isolate the three most independent fields to describe the songs (The artist, album and title).



The diagram also shows that there will be a PlaylistSongs table. This table was used to create a relation between the Songs and Playlists tables in a more normalized format (making sure the two tables did not have a many to many relationship).

### Algorithms required in solution

#### Class-less Code

Public procedure retrieveData(tableName)

Database = openRead(database)

Selected\_table = database.read(tableName)

Return results

End procedure

#### Login Window Class

Class LoginScreen

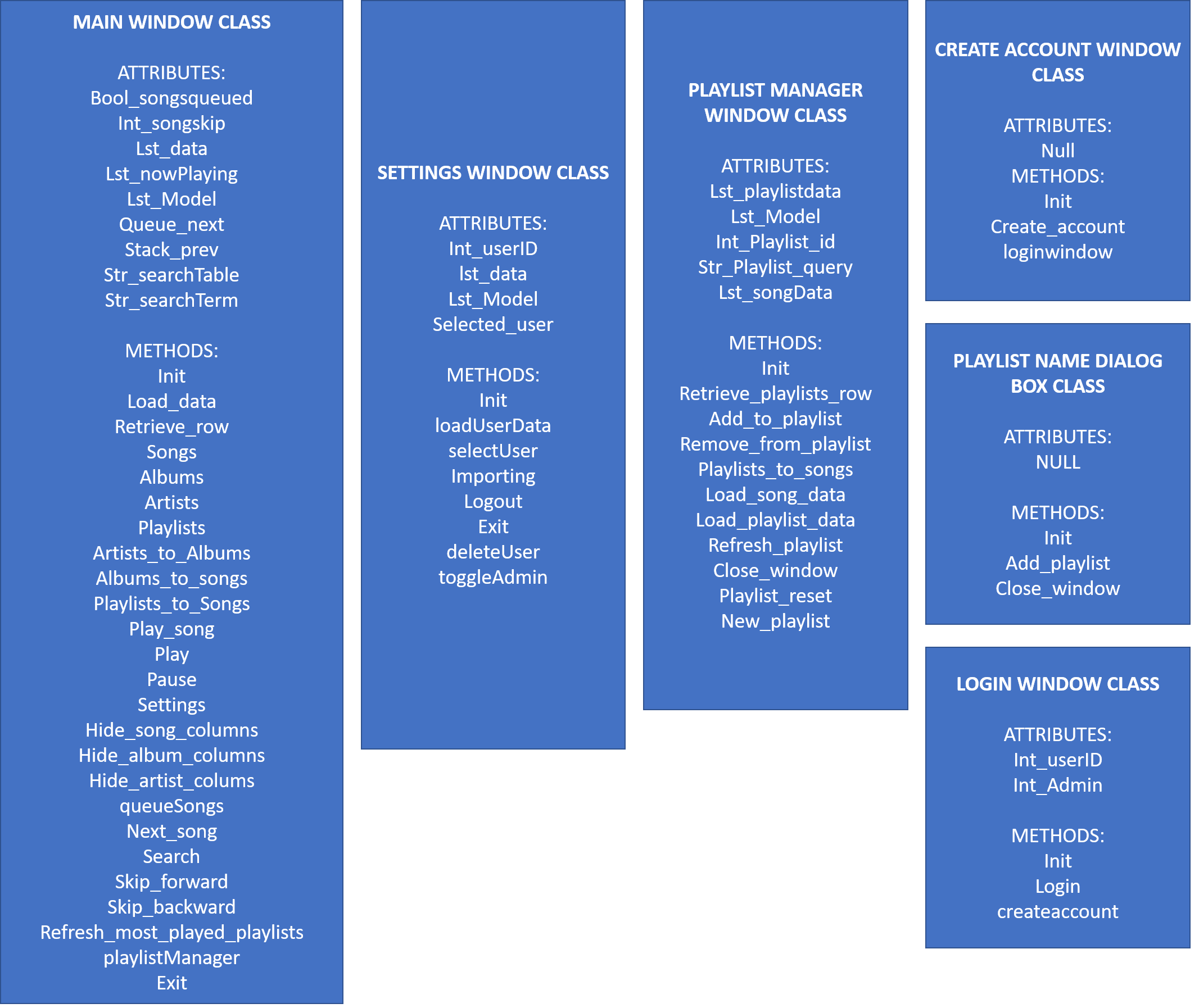
Public procedure

Public procedure login(str\_username, str\_password)

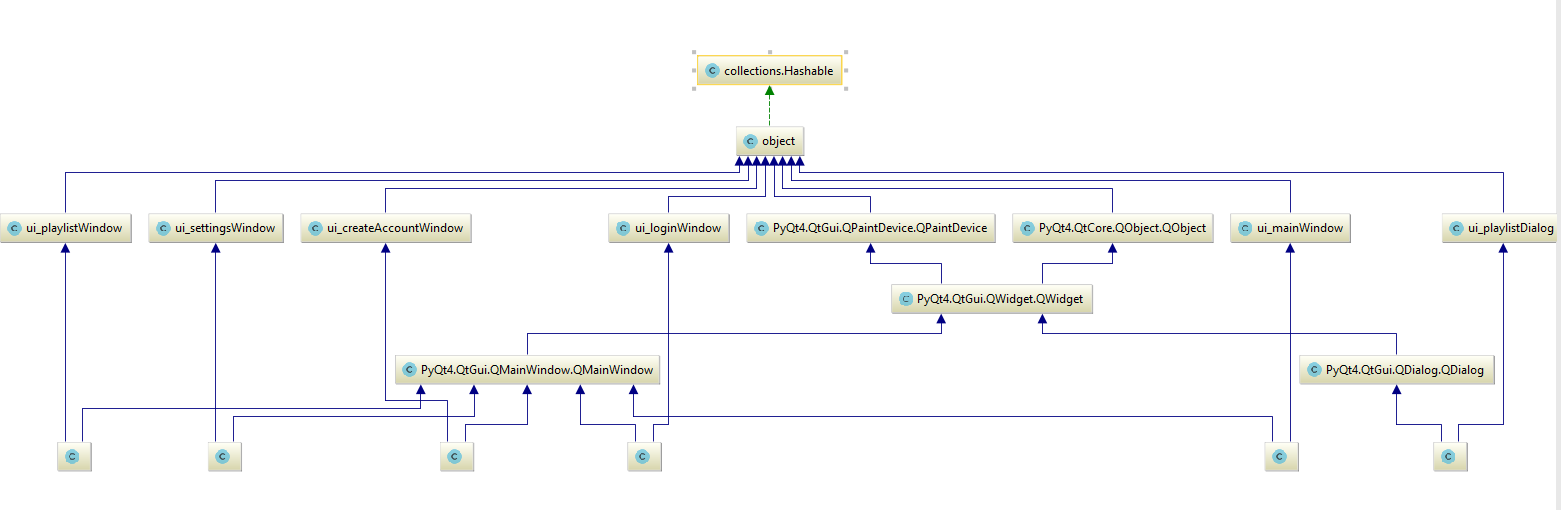
### UML Diagram

#### UML Class Diagram

Here are the diagrams showing the individual classes as well as their attributes and methods:



#### UML Program Stream Diagram

Here is a diagram showing the flow of the program stream and the connections between each of the classes:

## Approach to Testing

An iterative approach will be taken to test the programming solution. Meaning that after some major changes to the code, it will be tested by myself. Any problems or weaknesses that arrive from this testing will be improved in a revised iteration of the code.

### Test Data

This solution does not require much test data before testing begins because most of the data is inputted during the running of the program. For example, the data on the music files are appended when the physical music files are imported in the import function. Furthermore, user data can also be created in the create account screen of the program. The only data that must be inputted into the database is the Most Played Playlist record in the Playlist Table. However, an administrator user will also be added to the users table to ensure the ease of testing the solution.

However, for the solution to be tested, some mp3 files will need to exist. For my initial tests, I will use my own personal music library. However, in later sections of the program I will release my program to the initial stakeholders to gather their point of view and see how the code performs with different machines and music libraries.

### Test Plan

#### Database

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Success Criteria Tested** | **Input** | **Expected Output** |
| 1 | 1 | Open database file | All tables in the database will be related to one another and primary keys will exist for all tables. |
| 2 | 2 | Open database file | Songs table will exist |
| 3 | 3 | Open database file | Albums table will exist |
| 4 | 4 | Open database file | Artists table will exist |
| 5 | 5 | Open database file | Playlist table will exist |
| 6 | 6 | Open database file | Users table will exist |

#### User Interface

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Success Criteria Tested** | **Input** | **Expected Output** |
| 7 | 7, 13, 14, 15, 18, 19 | Open login window .ui file | User Interface will exist with all expected objects |
| 8 | 8, 20, 21, 22 | Open create account window .ui file | User Interface will exist with all expected objects |
| 9 | 9, 25, 31, 32, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 45, 46, 47 | Open Main Window .ui file | User Interface will exist with all expected objects |
| 10 | 10 | Open Settings Window.ui file | User Interface will exist with all expected objects |
| 11 | 11 | Open Playlist Manager .ui file | User Interface will exist with all expected objects |
| 12 | 12 | Open Playlist Name .ui file | User Interface will exist with all expected objects |

#### Login Screen

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Success Criteria Tested** | **Input** | **Branches** | **Expected Output** |
| 13 | 16, 17 | Login Button Pressed | 1. Correct username, incorrect password. 2. Incorrect username, correct password. 3. No details inputted. 4. Correct details inputted. | 1. Label will display: “Wrong Password” 2. Label will display: “Wrong Username” 3. Label will display: “Fields Empty” 4. User will be logged in & Main Window will be displayed. |

#### Create Account Window

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Success Criteria Tested** | **Input** | **Branches** | **Expected Output** |
| 14 | 23, 24 | Create Account Button pressed | 1. Password has no letters. 2. Password has no numbers 3. Password is not between 6 and 16 characters. 4. Username has already been used | 1. Label displays: “Please use letters in the password” 2. Label displays: “Please use numbers in the password” 3. Label displays: “Password must be 6-16 characters” |

#### Main Window

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Success Criteria Tested** | **Input** | **Branches** | **Expected Output** |
| 15 | 26 | Artist row is clicked in the table |  | All of the albums by artist in library are shown. |
| 16 | 27 | Album row clicked in table |  | All songs in album are shown |
| 17 | 28 | Song row clicked in table |  | Song begins playback |
| 18 | 29, 43 | Skip forward button press |  | Next song in queue will begin playback |
| 19 | 30 | Playlist row clicked in table |  | All song in playlist are shown |
| 20 | 31 | Artist button pressed |  | All artists in music library are shown |
| 21 | 32 | Albums button pressed |  | All albums in music library are shown |
| 22 | 33 | Songs button pressed |  | All songs in music library are shown |
| 23 | 34 | Playlists button pressed |  | All playlists owned by the user and the ‘Most Played’ playlist are shown |
| 24 | 35 | Settings button press |  | Settings window opens |
| 25 | 36 | Playlist Manager button pressed |  | Playlist Manager window opens |
| 26 | 39 | Search button pressed with inputs for the search term and its criteria |  | Matches with the term and criteria will be displayed. |
| 27 | 40 | Exit button pressed |  | The application will close |
| 28 | 41 | Pause button pressed |  | Music playback stops |
| 29 | 42 | Resume button pressed |  | Music playback **resumes** |
| 30 | 44, 45 | Skip backward button pressed |  | Previously played song begins playback |
| 31 | 46 | Song is played |  | Label changes to the title of the song being played. |
| 32 | 47 | Table is navigated |  | Labels change depending on the artist/album they are looking at |

#### Settings Screen

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Id** | **Success Criteria Tested** | **Input** | **Branches** | **Expected Output** |
| 33 | 48 | Settings button in Main Window pressed while logged in with an administrator account. | 1. Select user from the table 2. Delete user 3. Toggle admin of user | Administrator section is shown   1. User is selected and label changes to the name of the user. 2. User is deleted from database. 3. User’s administrator permissions are successfully toggled. |
| 34 | 49 | Directory Inputted and Import button pressed | 1. Invalid directory | All tagged files in folders are directly inputted into the library   1. Label will show: “Directory does not exist” |
| 35 | 50 | Exit button pressed |  | Settings window is closed |
| 36 | 51 | Logout button pressed |  | User is returned to the logout screen |

#### Playlist Manager Window

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Success Criteria Tested** | **Input** | **Branches** | **Expected Output** |
| 37 | 53 | Playlist row selected in playlists table |  | Songs from selected playlists are shown in playlists table |
| 38 | 54 | Song in playlists table is selected |  | Selected song is removed from the playlist it is in |
| 39 | 56 | Song is songs table is selected |  | Selected song is added to the selected playlist |
| 40 | 57 | Deselect Playlist button clicked |  | Playlist is deselected and all playlists are shown again in the playlists table |
| 41 | 58 | New Playlist button clicked |  | Playlist Name dialog box opens |
| 42 | 59 | Close Window button clicked |  | Playlist Manager window will close |

#### Playlist Dialog Box

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Success Criteria Tested** | **Input** | **Branches** | **Expected Output** |
| 43 | 61 | OK button pressed | 1. Playlist name is under 5 characters | Playlist Name dialog box closes and playlist is created.   1. Label shows “Playlist name is too short” |
| 44 | 62 | Cancel button pressed |  | Playlist Name dialog box is closed |

# System Development

## Database Initialisation

### Problem Decomposition

The database will have to hold all the data on the songs in the user’s library, particularly the artist’s name, the album’s name and the song’s name. On top of this, the database must also store all the information on the playlists created in the program and the information of all the users who have access to the program.

To maintain efficiency in the database (removing data redundancy and duplication), the database will be put in third normal form. Doing so will also prevent problems in data management by the program.

### Creating the Tables

As mentioned in the Design section of the documentation, the database will contain the following tables:

1. Albums
2. Artists
3. Songs
4. Playlists
5. Users
6. PlaylistSongs

The required columns for each of these tables will be created through SQLite code. The code will also have relations assigned to them through foreign keys. Primary keys will also be assigned to each table.

When the solution is distributed for the stakeholder’s use, the Albums, Artists. Songs, Users and PlaylistSongs tables will be empty, as these are filled with information specific to each user. The Playlists table will contain one field, for the most played playlist. For the testing of the program, test data will be loaded into the

First, the tables and primary keys will be established. Then, the foreign keys relating the databases will be established.

#### Albums

The albums table will hold all of the information specifically related to the albums in general:

1. AlbumID (int) **(PK)**
2. AlbumName (text)
3. ArtistID (int) **(FK)**

Any relations to artists table will be established after the creation of the tables.

Creation Code:

CREATE TABLE `Albums` (

`AlbumID` INTEGER UNIQUE,

`AlbumName` TEXT,

`ArtistID` INTEGER,

PRIMARY KEY(`AlbumID`),

);

#### Artists

The artists table will hold all the information specifically related to the artists in general:

1. ArtistsID (int) **(PK)**
2. ArtistName (text)

Creation Code:

CREATE TABLE `Artists` (

`ArtistID` INTEGER UNIQUE,

`ArtistName` TEXT,

PRIMARY KEY(`ArtistID`)

);

#### Songs

The songs table will hold all of the information specifically on the songs in general:

1. SongID (int)
2. SongName (text)
3. TrackNumber (int)
4. Genre (text)
5. FileLocation (str)
6. AlbumID (int) **(FK)**
7. Length (int)
8. Plays (int)

Any relations to the albums and artists tables will be created after the initial creation of the tables.

Creation Code:

CREATE TABLE `Songs` (

`SongID` INTEGER UNIQUE,

`TrackNumber` INTEGER,

`SongName` TEXT,

`Genre` TEXT,

`FileLocation` TEXT,

`AlbumID` INTEGER,

`Length` REAL,

`Plays` INTEGER,

PRIMARY KEY(`SongID`),

);

CREATE TABLE `Songs` (

`SongID` INTEGER UNIQUE,

`TrackNumber` INTEGER,

`SongName` TEXT,

`Genre` TEXT,

`FileLocation` TEXT,

`AlbumID` INTEGER,

`Length` REAL,

`Plays` INTEGER,

PRIMARY KEY(`SongID`),

);

#### Playlists

The playlists table will hold all the information related to the playlists in the program. This will include all user-created playlists and the ‘Most Played’ playlist that will already be in the table. The table will include the following values:

1. PlaylistID (int) **(PK)**
2. Playlist (text)
3. UserID (int) **(FK)**

Creation Code:

#### PlaylistSongs

CREATE TABLE `Playlists` (

`PlaylistID` INTEGER UNIQUE,

`Playlist` TEXT,

`UserID` INT,

PRIMARY KEY(`PlaylistID`),

);

In order to keep the database in third normal form, none of the tables can have a many-to-many relationship with another. This table is used as a go-between for the Playlists table and the Songs table. Instead of having a many-to-many relationship with one another, they now each have a one-to-many relationship with the PlaylistSongs table. The contents of this table are as follows:

1. PlaylistSongsID **(PK)**
2. PlaylistID **(FK)**
3. SongID **(FK)**

Creation Code:

CREATE TABLE `PlaylistSongs` (

`PlaylistSongsID` INTEGER,

`PlaylistID` INTEGER,

`SongID` INTEGER,

PRIMARY KEY(`PlaylistSongsID`),

);

#### Users

This table will hold all of the information about the Users. The fields in the table are as follows:

1. UserID (int) **(PK)**
2. UserName (text)
3. UserPassword (text)
4. Administrator (int)

Creation Code:

CREATE TABLE `Users` (

`UserID` INTEGER UNIQUE,

`UserName` TEXT,

`UserPassword` TEXT,

`Administrator` INT DEFAULT 0,

PRIMARY KEY(`UserID`)

);

### Creating Relationships

After the initial tables were created, the relationships between them were established through the foreign keys. The foreign keys used in this database are as follows:

1. Albums.ArtistID 🡪 Artists.ArtistID
2. Playlists.UserID 🡪 Users.UserID
3. PlaylistSongs.PlaylistID 🡪 Playlists.PlaylistID
4. PlaylistSongs.SongID 🡪 Songs.SongID
5. Songs.AlbumID 🡪 Albums.AlbumID

These relationships will be made in **DB Browser for SQLite** (the Database Management System).

#### Reformed Code

**Albums**

CREATE TABLE `Albums` (

`AlbumID` INTEGER UNIQUE,

`AlbumName` TEXT,

`ArtistID` INTEGER,

PRIMARY KEY(`AlbumID`),

FOREIGN KEY(`ArtistID`) REFERENCES `Artists`(`ArtistID`)

);

**Artists**

CREATE TABLE `Artists` (

`ArtistID` INTEGER UNIQUE,

`ArtistName` TEXT,

PRIMARY KEY(`ArtistID`)

);

**PlaylistSongs**

CREATE TABLE `PlaylistSongs` (

`PlaylistSongsID` INTEGER,

`PlaylistID` INTEGER,

`SongID` INTEGER,

PRIMARY KEY(`PlaylistSongsID`),

FOREIGN KEY(`PlaylistID`) REFERENCES `Playlists`(`PlaylistID`),

FOREIGN KEY(`SongID`) REFERENCES `Songs`(`SongID`)

);

**Playlists**

CREATE TABLE `Playlists` (

`PlaylistID` INTEGER UNIQUE,

`Playlist` TEXT,

`UserID` INT,

PRIMARY KEY(`PlaylistID`),

FOREIGN KEY(`UserID`) REFERENCES `Users`(`UserID`)

);

**Songs**

CREATE TABLE `Songs` (

`SongID` INTEGER UNIQUE,

`TrackNumber` INTEGER,

`SongName` TEXT,

`Genre` TEXT,

`FileLocation` TEXT,

`AlbumID` INTEGER,

`Length` REAL,

`Plays` INTEGER,

PRIMARY KEY(`SongID`),

FOREIGN KEY(`AlbumID`) REFERENCES `Albums`(`AlbumID`)

);

**Users**

CREATE TABLE `Users` (

`UserID` INTEGER UNIQUE,

`UserName` TEXT,

`UserPassword` TEXT,

`Administrator` INT DEFAULT 0,

PRIMARY KEY(`UserID`)

);

### Loading the Test Data

As mentioned previously, the artists, albums, songs and PlaylistSongs tables take their data as inputs from the user. These tables will be empty as a result. The Playlist table will only contain a field for the ‘Most Played’ playlist.

#### Most Played Playlist

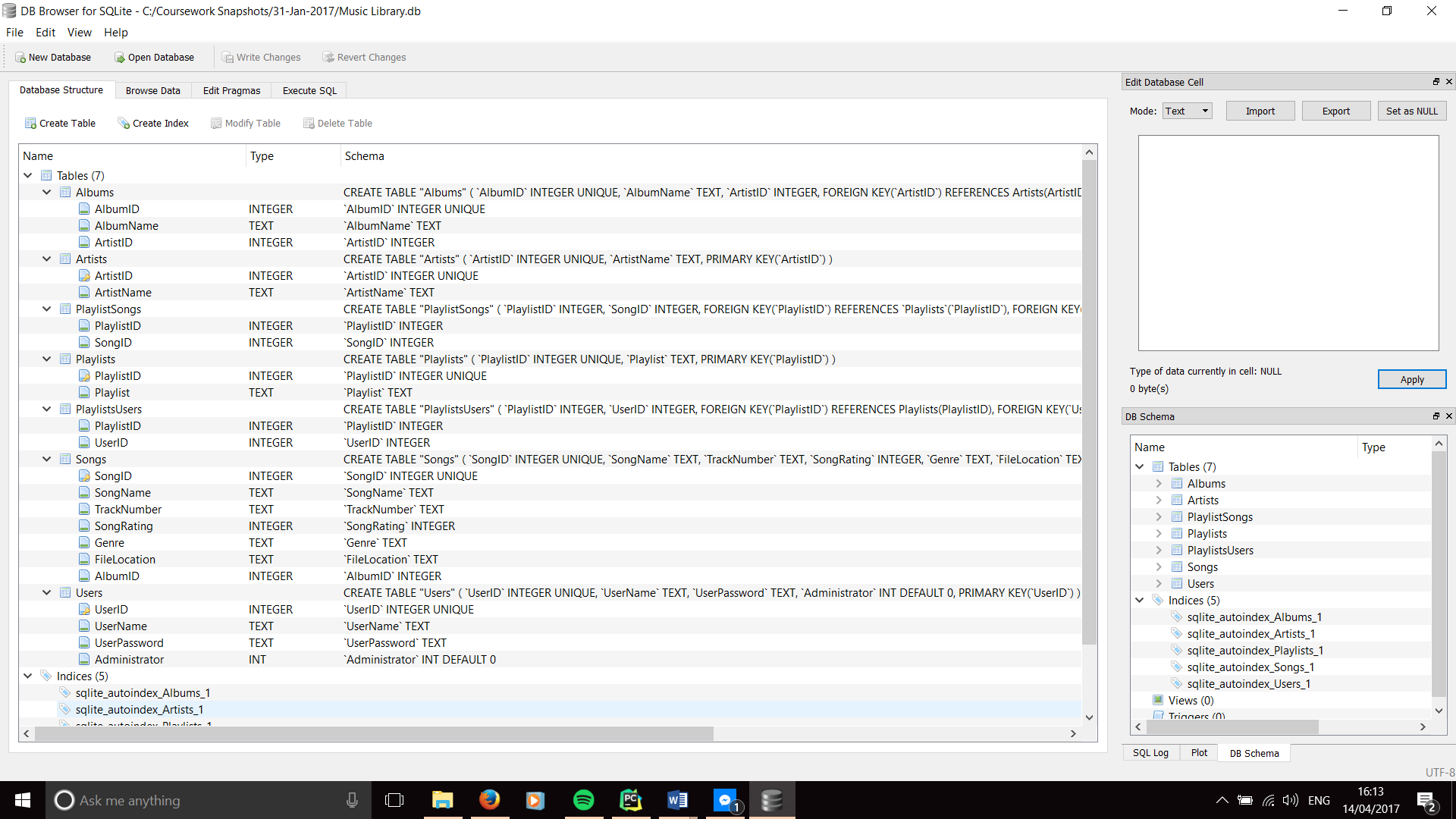
Here is the code used to insert ‘Most Played’ playlist into the Playlists database:

“INSERT INTO Playlists VALUES (1, “Most Played”, 0)”

The playlist is given a UserID of 0 as it is available for all users to use.

### Final View

Here is the final view of the database:



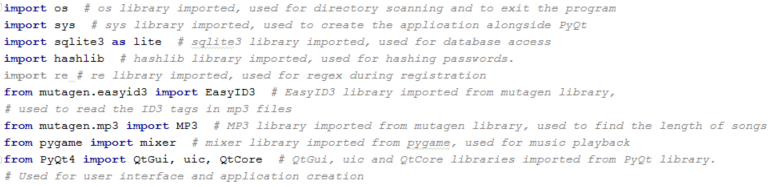
## Class-less code

Some of the code to be used in the program does not belong in any specific class. Most of these lines are to initialise components of the program (such as the database and .ui files) or to create objects.

All the imported libraries also belong in this section.

### Imported Libraries

Here is the code for the imported libraries that are used by the program. The lines have also been annotated to show what each library is used for:



### Linking the Database with the code

The database created in 4.1 must be linked to the code before it can be used by the program.

Here is the code used to connect the database:

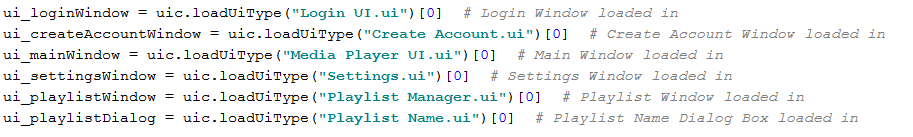


The first line connects the database file (Music Library.db) to the program, while the second line establishes the cursor to allow SQL queries to made for the database.

### Connecting the User Interface Files

The User Interface files are created in QtDesigner through .ui files. The files are then imported into the program.

As the files were added at different points in the development cycle, evidence of iterative development will be shown in later sections. However, the final code of the UI imports are as follows:



## Login Screen Class

### Project Decomposition

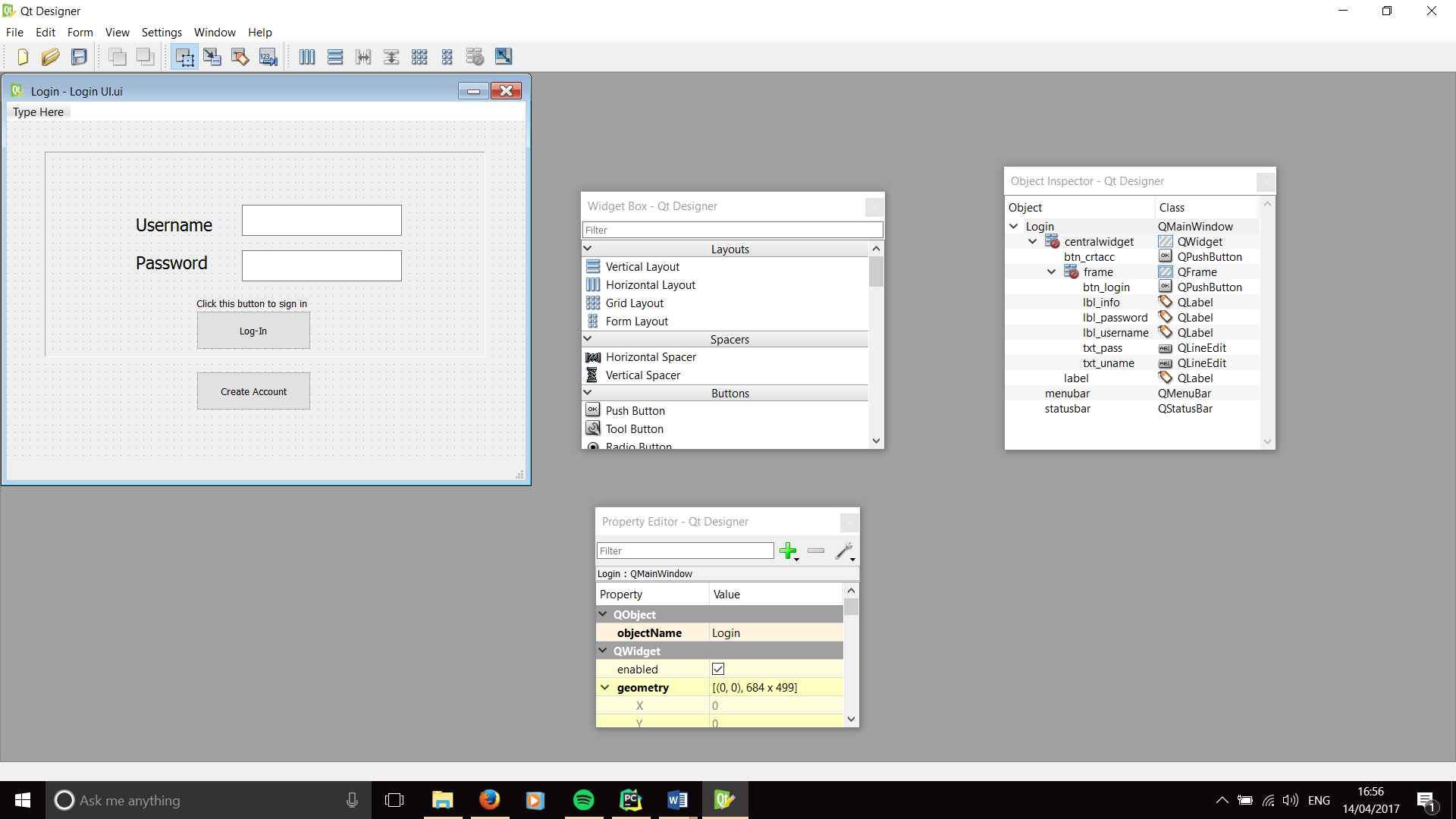
This class will hold all the code required for the ‘Log-In’ section of the program. It will be the first screen/class used by the user when the program starts.

The code will have to take two inputs (the user’s username and password) and validate them against the details in the Users table in the database. As the passwords in the database are required to be hashed, the users input must also be hashed before it is compared to the password in the database. If the validation is not successful, an appropriate message will be displayed to the user. Else, the Login window will close and the Main Window will be opened.

### 1st Iteration

#### User Interface

Before I started to code the class, I created the User Interface for the Login Screen. Changes will be made throughout the development to accommodate more features. In this iteration, I aimed to only cater for the basic login section of this class. Here is the initial look of the login screen:



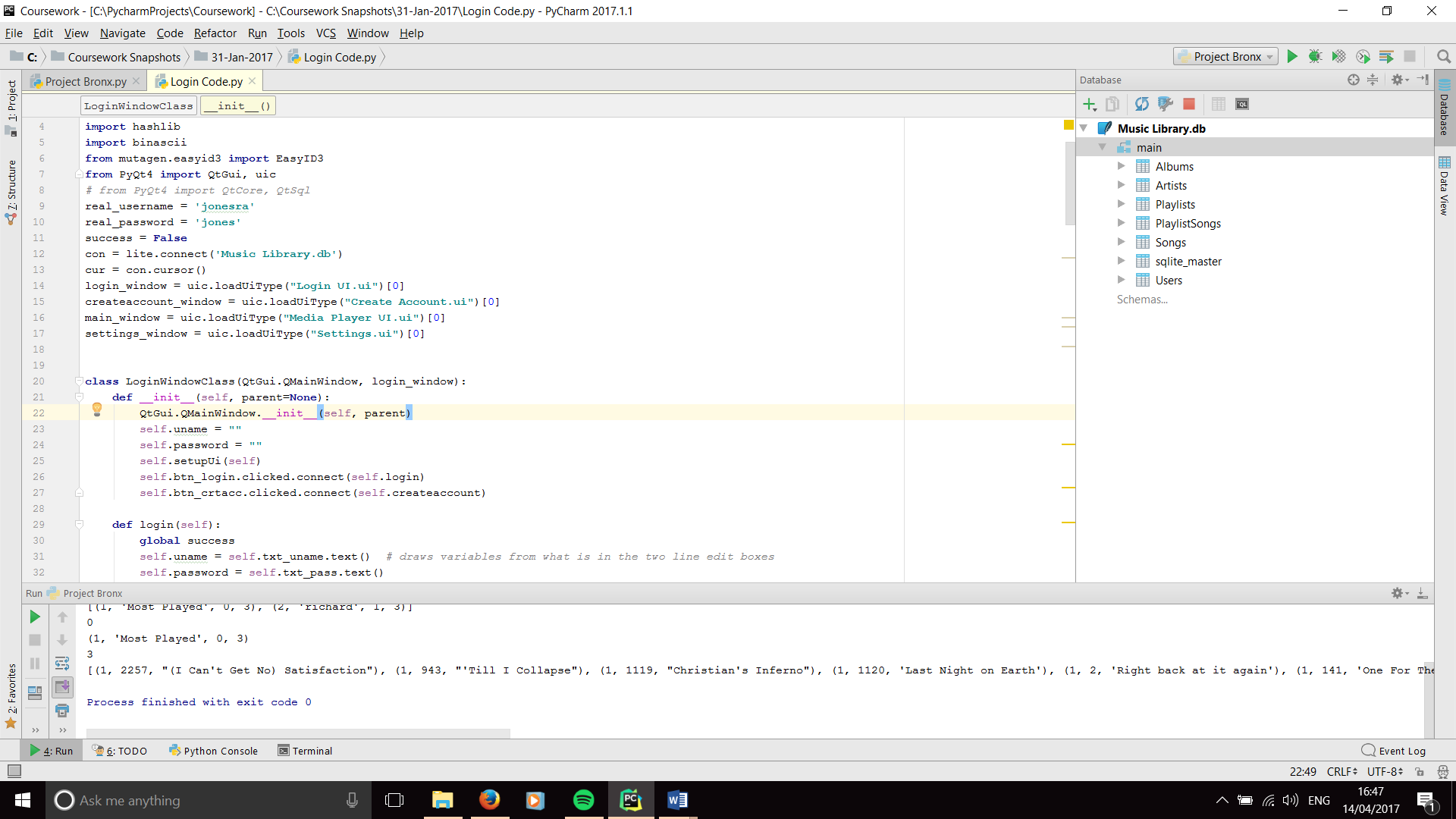
The Window uses several objects from the Qt Framework:

1. The textboxes use QLineEdit objects to allow the users to input their usernames and passwords.
2. QPushButton objects are used for the Log-In and Create Account buttons.
3. QLabel objects are used for the Username and Password labels as well as the sign-in instruction.

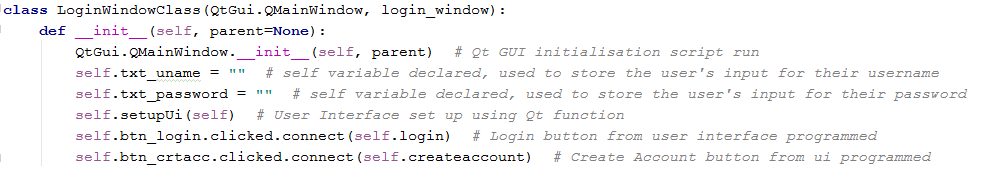
#### Code

For the first attempt at writing this class, I focused on writing just the validation process, with the hopes to implement the database and hashing function later. Here is the first iteration of the code.

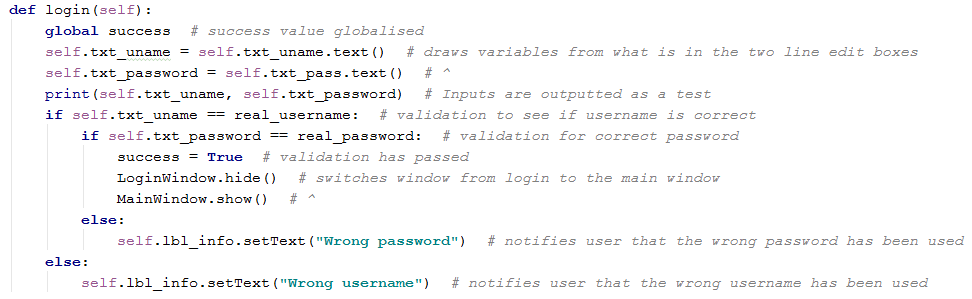
First, the correct username and password were declared outside of the class, as well as the ‘success’ variable for the validation:



Then the shell of the class was created and the initialisation function was written. The initialisation functions in each class will hold all of the definitions of attributes (including the objects in the ui) and the initialisation code for the GUI:



Then the algorithm for the login validation was written. The code retrieves the user’s inputs and validates them against the ‘real username’ and the ‘real password’ that have already been declared. If the usernames or passwords don’t match, an appropriate message will be displayed. Else, the current screen will be closed and the next screen will be opened:



The code has been written using a **Hungarian notation** standard. This is to support maintainability of code.

This code does not represent the full functionality of the program. However, it will still need to be tested.

### Testing for Iteration 01

#### Inputting the correct User Details

For this section, the inputs used will be:

Username = “jonesra”

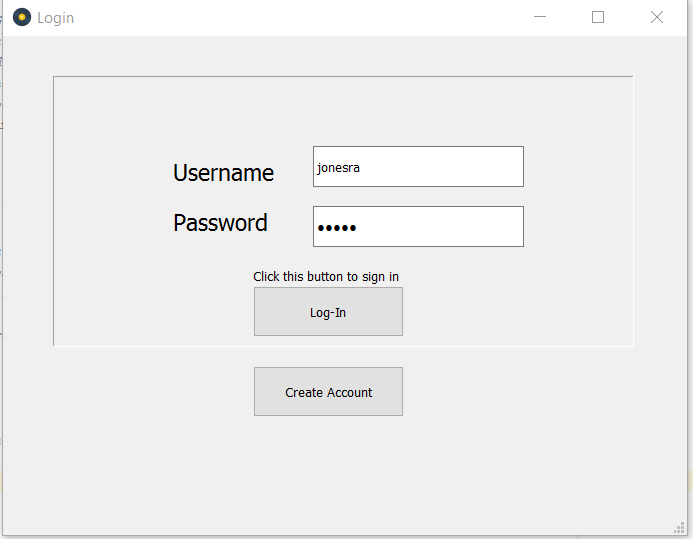
Password = “jones”

And the Login button is activated

As the username and password variables match the ‘real username’ and ‘real password’ variables, the expected output will be that the login window will close and the media player window will open.

**Evidence of testing:**

**Inputs:**



**Output:**



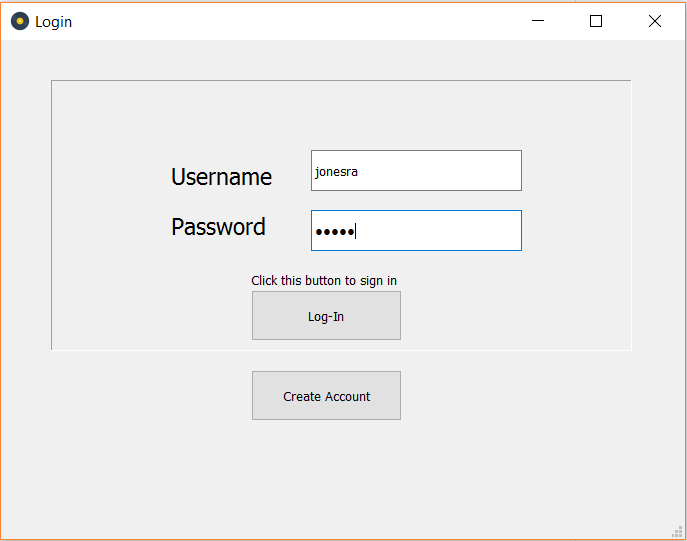


The two pictures show the inputs being inputted and the result of it. The fact that the window changed proves that the test is **Successful**

#### Inputting the correct Username and the incorrect Password

For this test, the username input will be “jonesra” and the password input will be “wrong”. The expected result is that the message “wrong password” will be displayed on the screen.

**Inputs:**



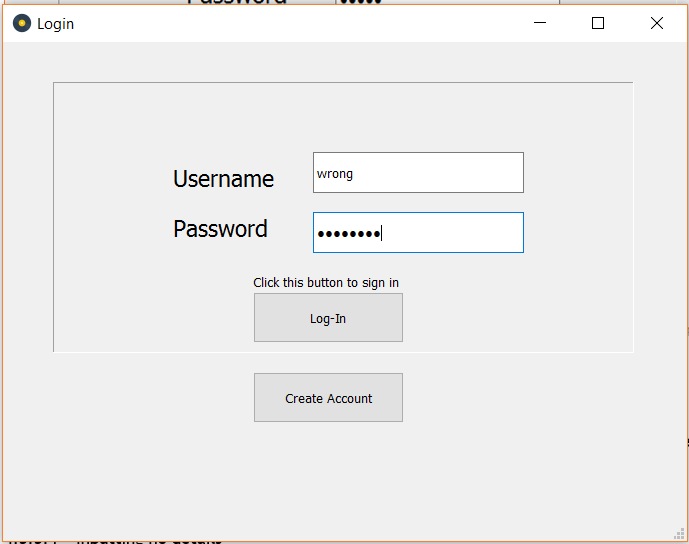
**Outputs:**

Note that a message has been displayed saying “wrong password”, showing the test to be **Successful**.

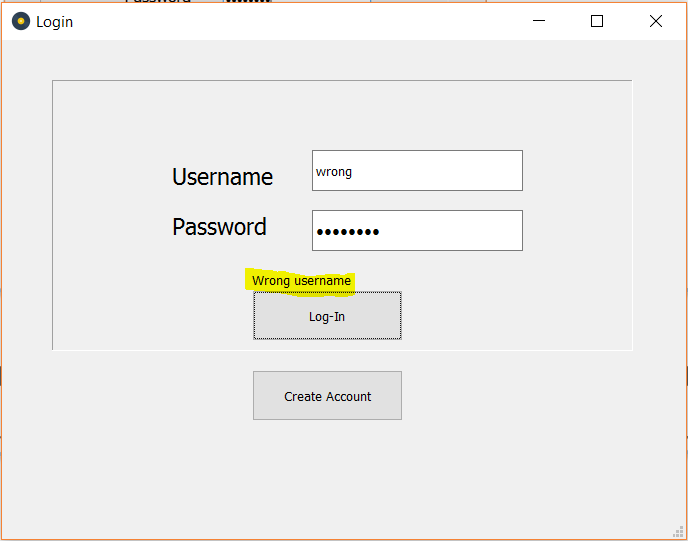
#### Inputting the correct Password and the incorrect Username

For this test, the username input will be “wrong” and the password will be “baritone”. The expected result will be that the screen will display the “wrong username” message.

**Inputs:**



**Outputs:**

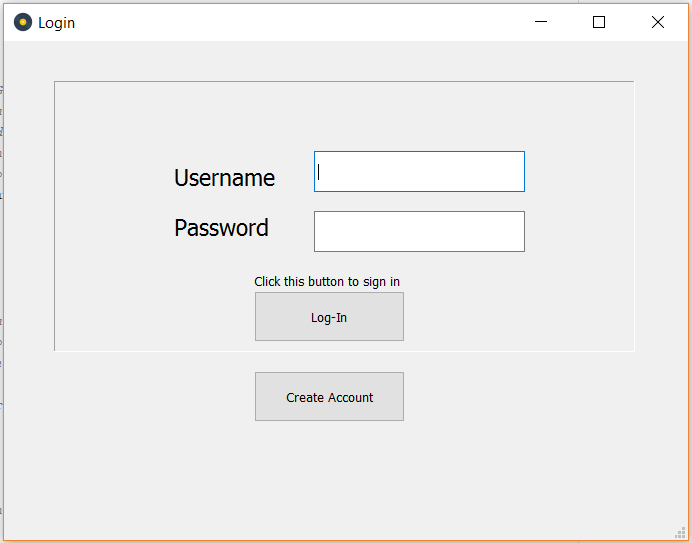


As the screenshot shows, the message “wrong username” was displayed, showing the test to be a **Success**

#### No username or password is inputted

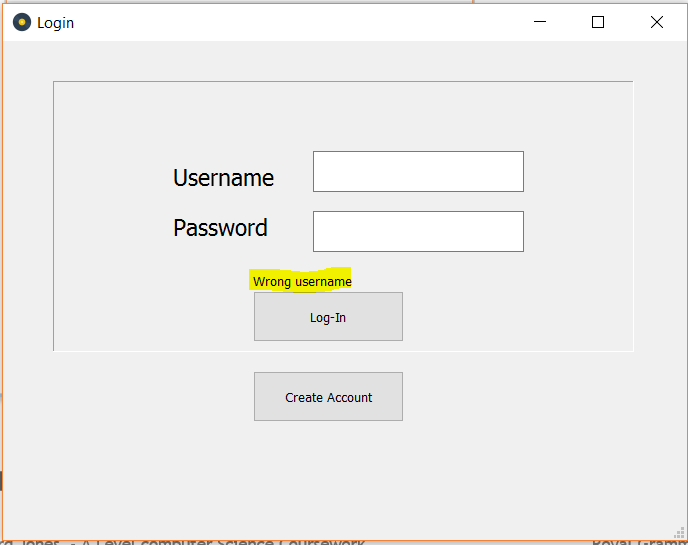
In this test, no username or password will be inputted. Just the login button will be activated. This section of the problem has not been coded yet, so the test is likely to show an error and no output. This will be fixed in the later iteration.

**Inputs:**



As the screenshot shows, there are no inputs in the text boxes.

**Outputs:**



Note that the “wrong username” message is still displayed.

However, there was no error message shown:



This test was obviously a **Failure.** Though no error was shown, a specific message will need to be shown in the final solution to meet all the objectives.

#### Conclusion

These tests show that the login problem is taking shape. However, the code still needs to check with details in the database and hash the password inputs. Furthermore, the testing showed that a specific message will need to be shown if the user doesn’t input anything in the login screen. These changes will be addressed in the second iteration of the code.

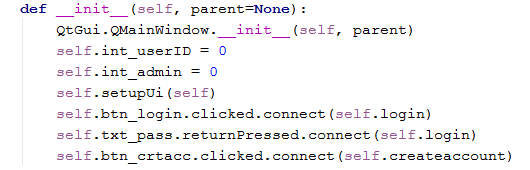
### 2nd Iteration

The changes mentioned in the conclusion for the 1st Iteration need to be made in this iteration. These mainly focus around:

1. Checking the login inputs against the users table in the database, hashing the passwords to achieve this.
2. Display a specific message if the user doesn’t input a username or password.

On top of this, the User’s ID, admin permission, and username will have to be stored as variables to be accessed by later sections of code.

#### Code



The initialisation section of the class has been slightly improved. First, the username and password sections have been changed from ‘self’ variables to local variables in the login procedure. This was because the variables were not needed throughout the class or the rest of the code. Furthermore, the int\_userID and int\_admin variables were declared. These variables will store the id and administrator permissions of the user who is logged in for each session. The listener for the create account button was also set up to call the ‘createaccount’ procedure when it is clicked.



This code has been extended to meet the changes mentioned earlier. As the annotations show, the code checks the user’s login details against all of the login details in the database using SQL queries. The user’s inputs are validated for three pieces of criteria: that the user has inputted both a username and a password, that the username is in the database and that the user’s password corresponding to the inputted username is correct (using the hashing function to do so).

Furthermore, the code for opening the create account screen was coded:



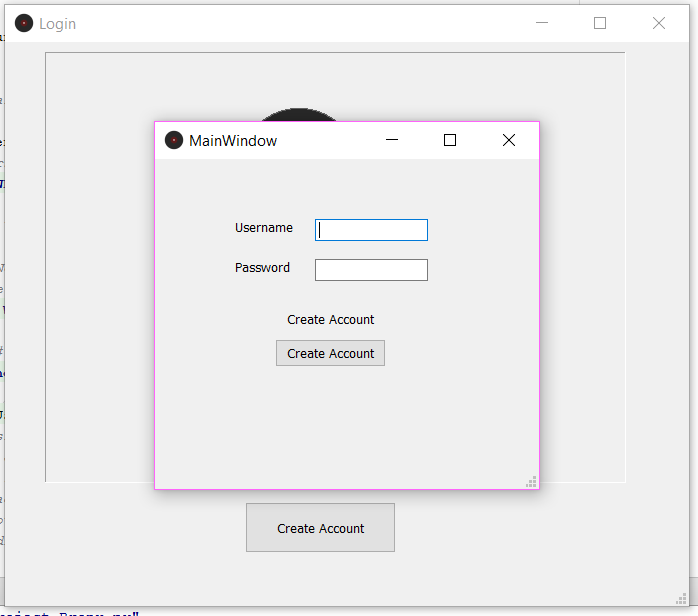
This procedure will simply open the create account window when it is run.

### Testing for Iteration 02

#### Create Account Window

For this test, the create account button was activated. The expected output is that the create account window will show.

**Output:**

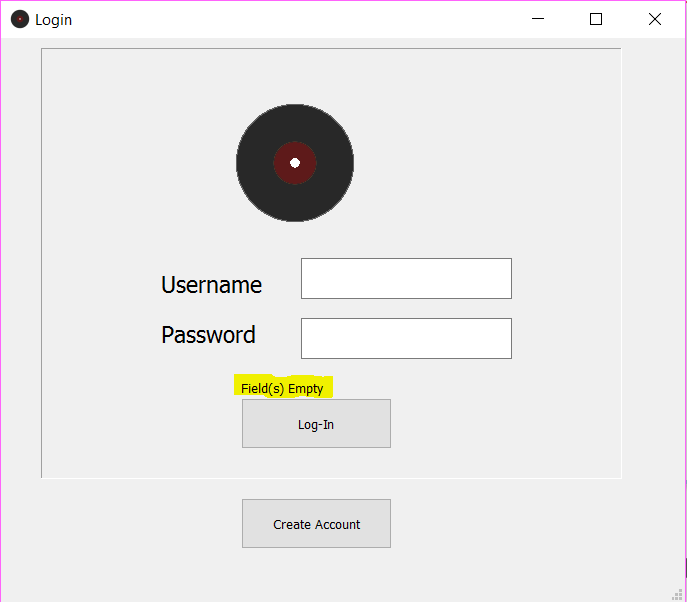


This screenshot shows that the create account window was opened. Therefore, the test was successful.

#### No username or password is inputted

For this test, the login button was activated without any inputs for the username or password. The expected output is that the message “Field(s) Empty” should be displayed in the lbl\_info object.

**Output:**



As the screenshot shows, the message was shown with the given inputs. Therefore, the test was successful.

## Create Account Window Class

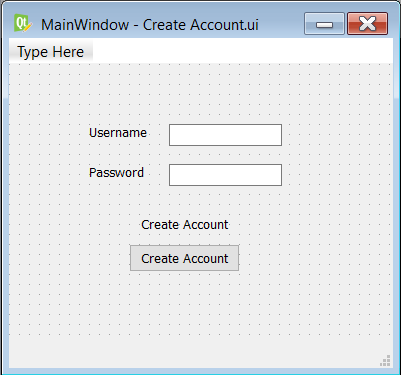
### Project Decomposition

For this section of the code, the program will need to allow the user to input the username and password that they want for their new account. Then, through the click of a button, they can have their details validated for several criteria (length, whether or not it had letters and numbers in and whether the username already exists in the database). If the validation is passed, then the new account will be created in the database and the user will be taken back to the Login page. If the validation fails, an appropriate message will be shown.

### Final Code

#### User Interface

Here is the final design of the ‘Create Account’ Window’s user interface:



The interface contains all of the objects mentioned in the original mock-up of the design as well as some labels (for the username and password boxes) to increase accessibility to the user.

#### Initialisation Code

As the annotations show, this initialisation script has three main functions. First, it runs initialisation script for PyQt. Second, it sets up the previously specified user interface file and third, it creates a listener for the create account button.

#### Create Account Function



This code is the main part of the create account class. The function takes the two inputs from the user and assigns them to variables in Hungarian notation. Then the password is passed through a set of validation conditions using regular expressions (regex). The validation checks to make sure the password is secure enough by making sure it contained letters and numbers and was between 6 and 16 characters. After the validation for the password, a new userID is created by incrementing the highest userID in the database by 1. Then the username is also validated to make sure there is not an identical username already in the database. This is done by counting number of fields in the Users tables where the username matches the user’s input. If the outputted result is not 0, then an appropriate message is displayed.

## Main Window Class

### Project Decomposition

This class has a lot of different functions:

#### Songs Table

The code will need to provide a table object that can have multiple pieces of data placed into it. The nature of this data will be specified by the user (e.g. albums, artists, songs, playlists) using a button. The table will allow data to be selected and activated, an appropriate action will occur. If an album record is clicked, the songs in the album will be displayed. If an artist is clicked, the albums by that artists will be displayed. If a song is clicked, the song will be played.

Headings will also be used to help users to understand what they are looking at.

#### Playing Songs

When a song is selected from the table. The data of that song will be retrieved and assigned to separate variables. A label will assign the name of the song to itself to notify the user of the song that is currently playing. The plays number of the song will also be incremented in the database. The next songs in the table were then queued.

#### Playbar Buttons

When a user activates the pause button, the song that is currently playing will be paused. When a user activates the skip forward button, the next song in the queue will be played. When a user activates the skip backward button, the song that was previously playing in the queue will be played.

#### Search Function

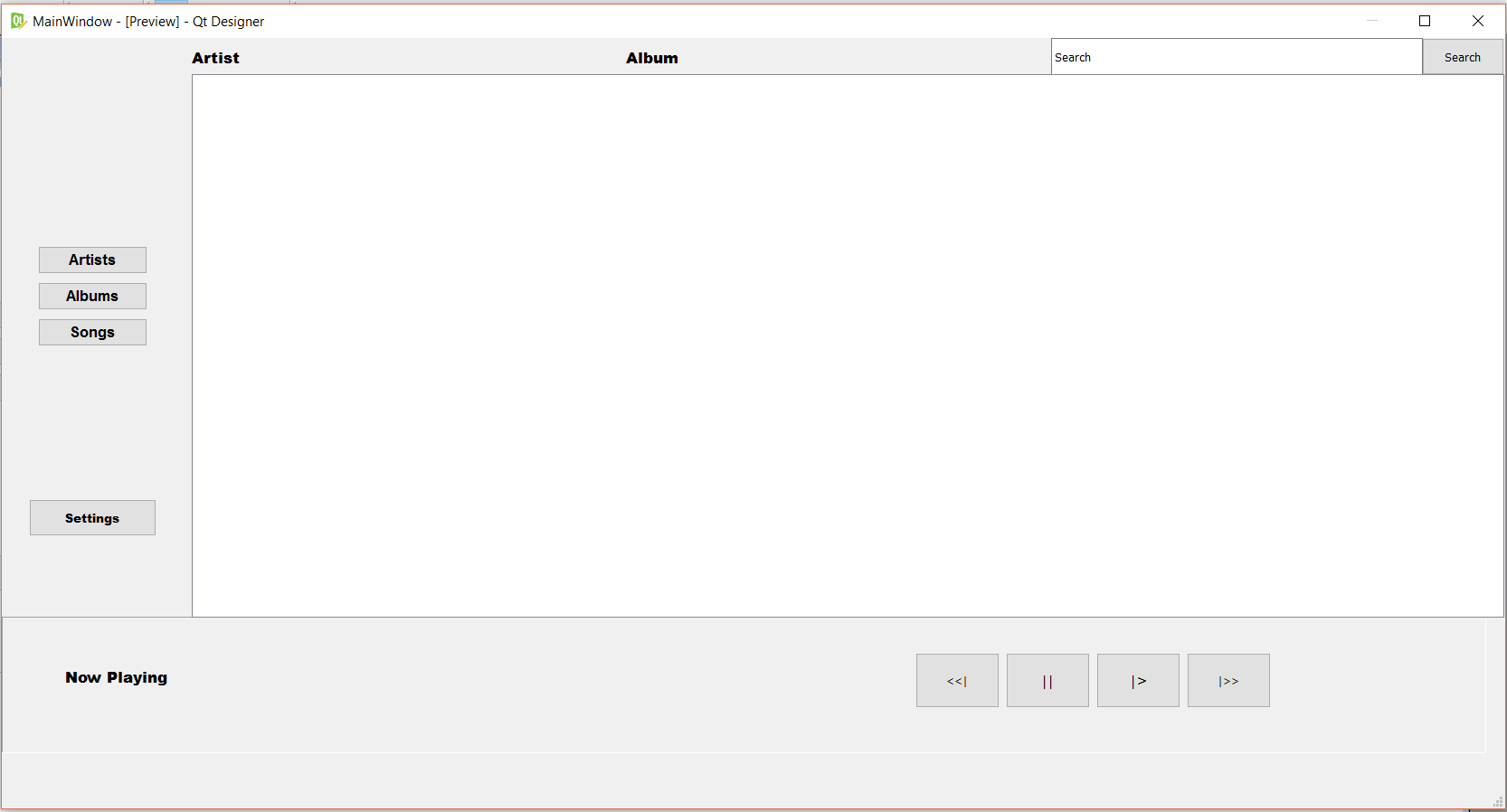
For this section. The user will be able to input a term that they want to search by and select what criteria they want to search (e.g. search by album, artist, song or playlist). Similar results should then be outputted in the song table.

### Iterative Development

#### Table Navigation

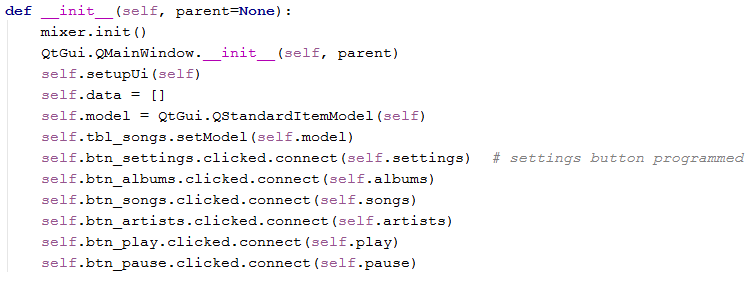
To start the coding of the Main Window class, I attempted to implement the table navigation functionality. The table would need to be loaded with data from many different inputs, including when one of the navigation buttons were clicked by the user or when a record in the table was clicked. Using SQL, this problem can be simplified. Here is what the first few steps of the code looked like:

User Interface



Here is the first iteration of the user interface. The interface has been set up to include some of the objects required for the final solution; such as the library navigation buttons (labelled artists, albums, songs), the settings buttons, the playbar buttons and the library table. The final code will hopefully also have a more stylish design that would be more enticing to the end user.

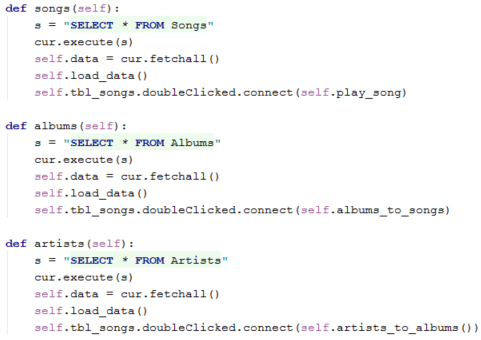
Initialisation Code



The initialisation code is similar to the other two classes I have already programmed. The code includes the initialisation of the user interface, the declaring of self variables and the setting up of listeners for the buttons of the user interface. The table navigation utilises the albums, songs and artists button listeners.

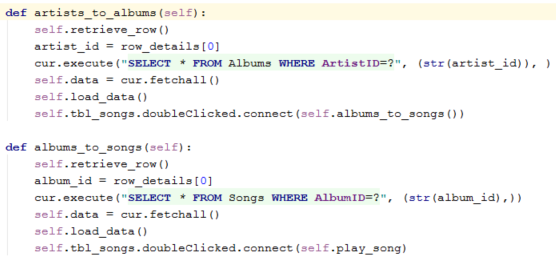
Table Navigation Buttons

Before the methods for the table navigation buttons could be implemented, the load\_data method



These three methods are each called when their respective buttons are clicked by the user. Each method starts by select all of the records from their respective tables through the use of an SQL query. The query is executed and the results are fetched and loaded into the songs table using the load data method. Then, each method contains a listener for the songs table in case its fields are double clicked. When a record is clicked, if the user is looking at songs (meaning they have clicked the song button and run the songs method), the play song method is run from that listener. If the user is looking at albums (meaning they have clicked the albums buttons and run the albums method), the albums to songs method is run.

This is continued by the albums\_to\_songs and artists\_to\_albums methods:



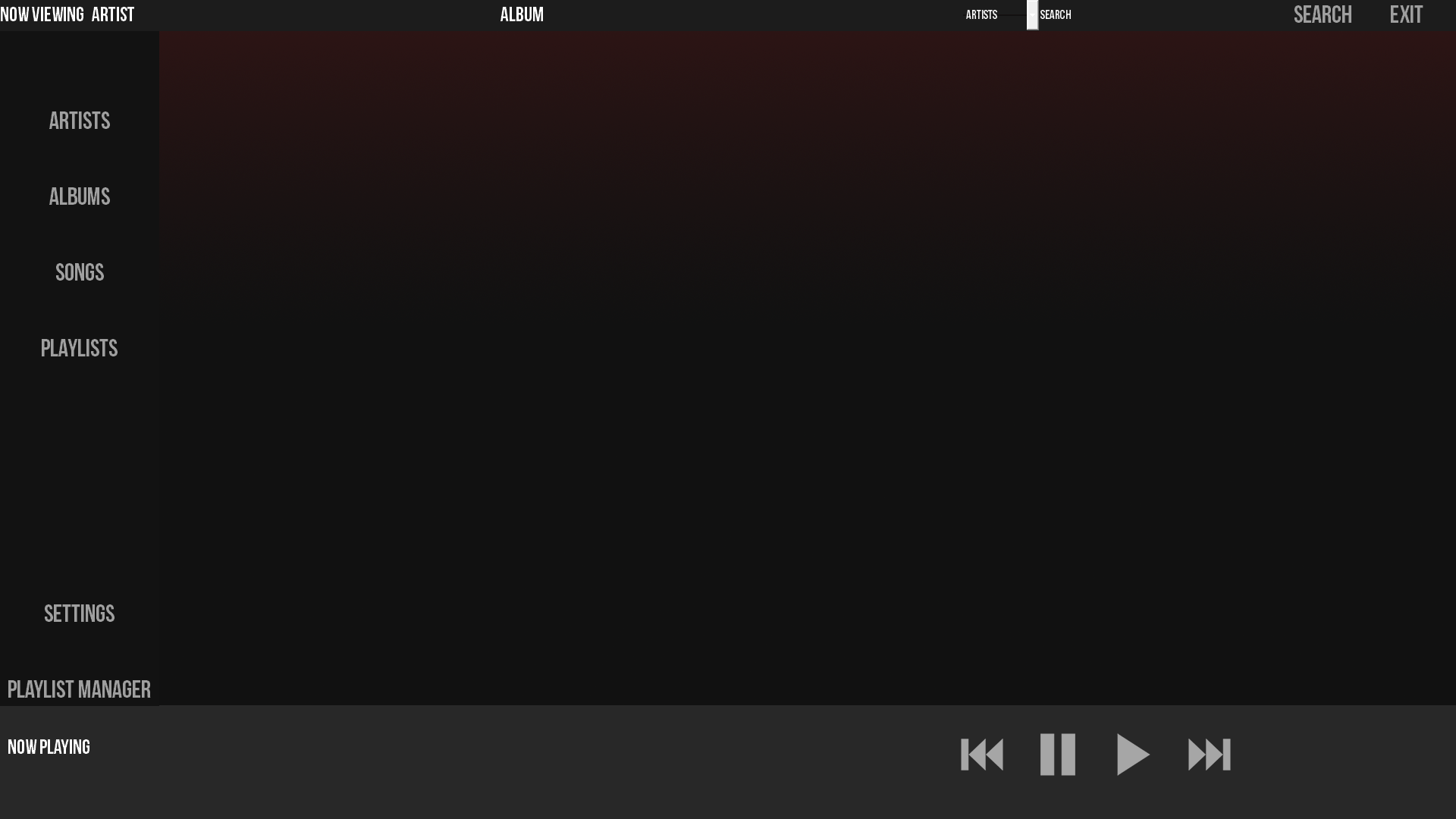
These two methods start by retrieving the row that has been selected by the user. In the case of the artists\_to\_albums method, the albums by the artist selected by the user will be retrieved through the execution of an SQL query. The results of the query are (once again) loaded into the songs table. Separate listeners for interaction with the songs table.

In conclusion, the methods constantly retrieve new data for the rows from SQL queries and set up specific listeners for interaction with the songs table depending on the user’s inputs. This functionality will now be thoroughly tested

### Iterative Testing

### Final Code

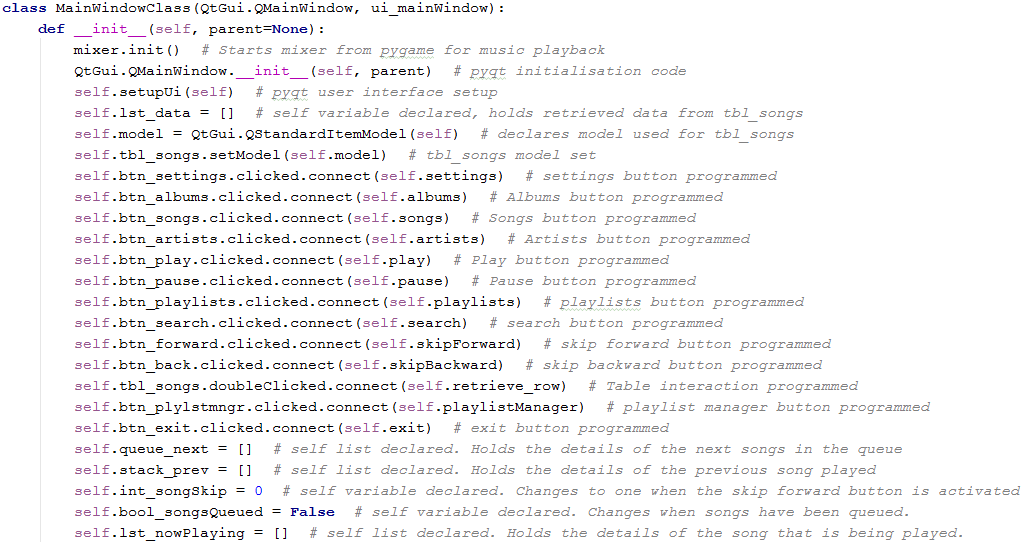
#### User Interface



In order to make the solution more attractive and comparable to an End-User product. I took some advice on the design of Main Window Screen from an Art Designer (Oliver Munby). I was advised to consider flat textured buttons with shades of grey and red. He also recommended I include some gradient effects for the background. I made an attempt to include this in the final design.

#### Initialisation Code

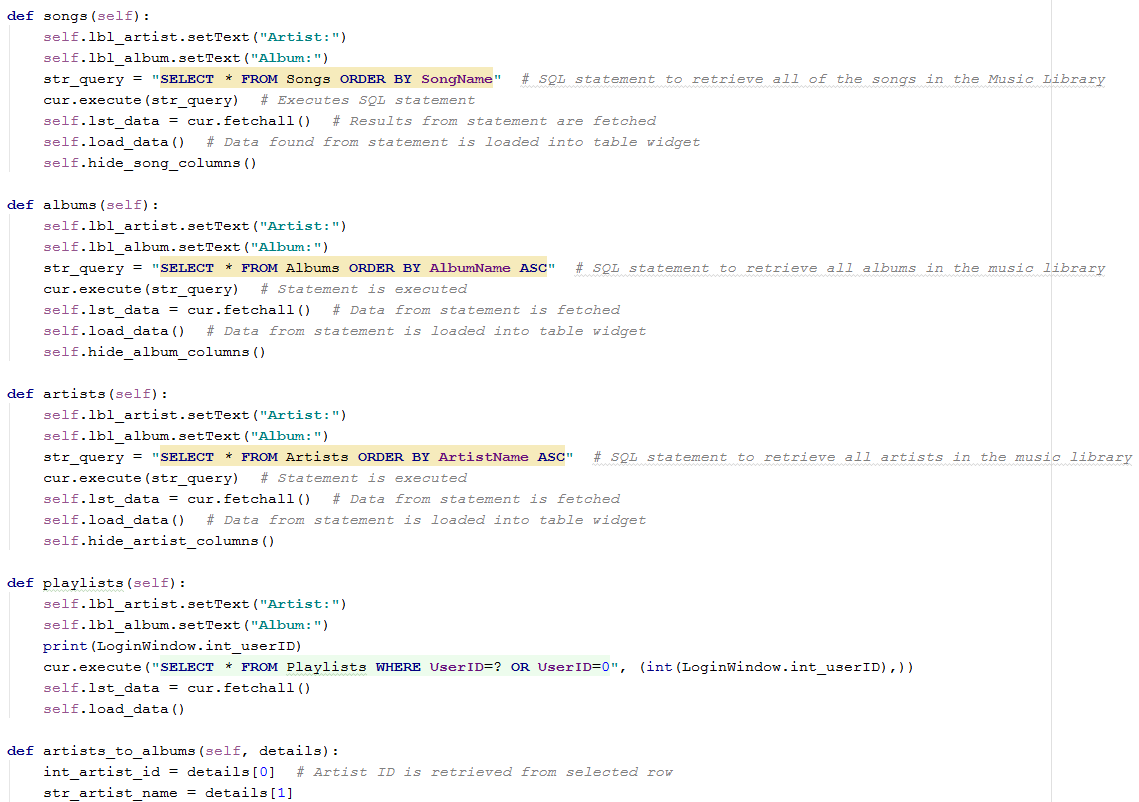
Here is the final look of the initialisation code:



#### Songs Table

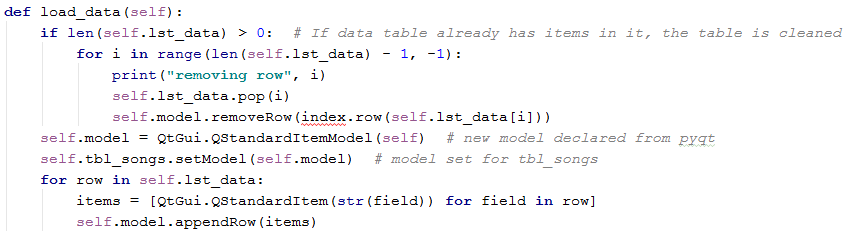
For the table selection code, this code has been amended to fix an error found in the iterative testing. The error happened because there were too many tblsongs.doubleClicked commands which had different responses. To fix this, I introduced a new field in the artists, albums, songs and playlists table. A different value was held for each individual table. Now, only one connect function can be used with several if statements discriminating this new field. Here is the code:

First, the album, songs and artist buttons were programmed:

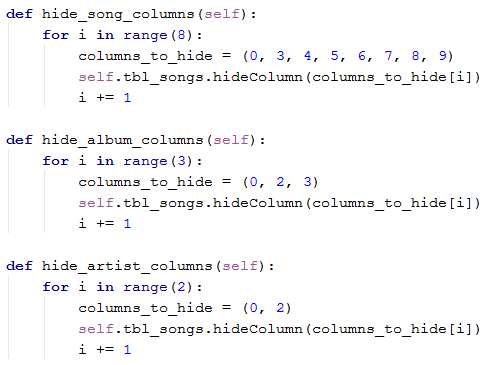


Each of these procedures follow the same formula. First, the labels used to assist user navigation were reset. Then, a suitable SQL query was used to retrieve all of the items in each table. The data from the queries were then fetched assigned to a variable. The load\_data function was then called. The hide\_?\_columns function was also used to tidy up the output to be seen by the user.

Here is the load\_data procedure:



And here are the hide\_?\_column procedures:



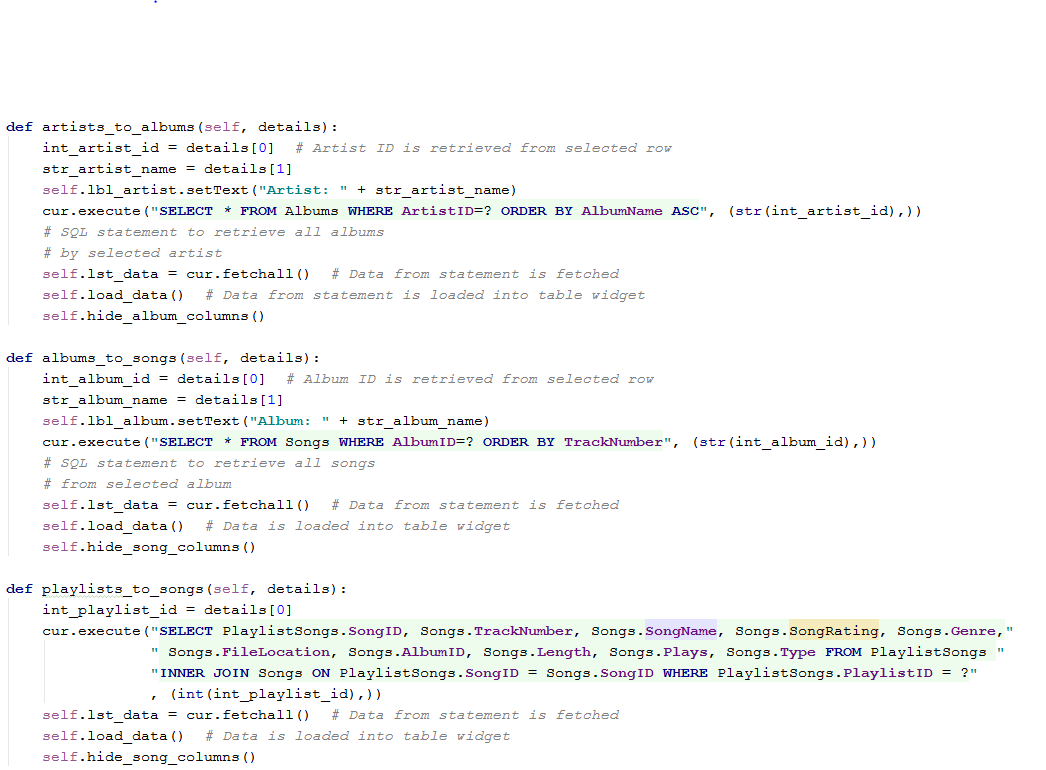
The procedures use iteration to hide all of the columns that the user does not need to see (such as the directory that the song are in). The specific columns are placed in an array.

The code must also allow the user to select certain fields in the table and perform a specific action, e.g. when an album is clicked in the table, the table will show all of the song in that album. This first piece of code is the first procedure used when the user double-clicks a field in the table:



This is the procedure which shows the new adjustments that were made to this class.

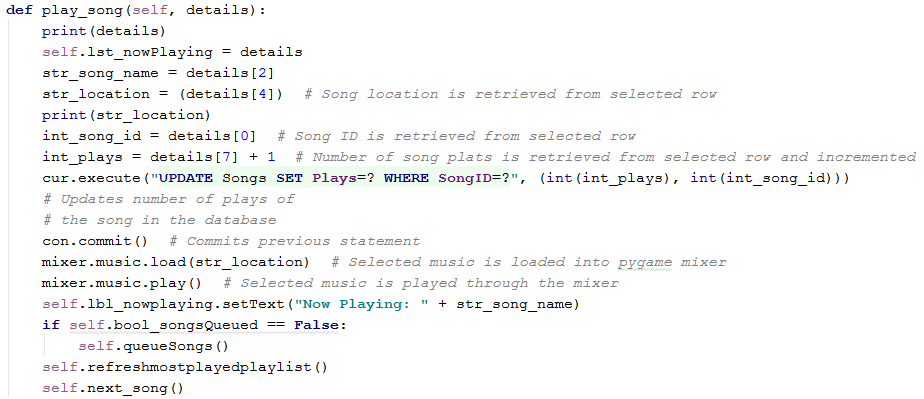
This code also points to four functions, artists\_to\_albums, albums\_to\_songs, playlist\_to\_songs and play\_song, here are the first three:



Once again, the functions follow the same formula to one another. First, the relevant information of the selected row is retrieved, typically the ID. For the first two functions, the navigation are labels are changed to either the artist or album name. Then an SQL query is used to fetch the relevant data where the ID number is identical. For the playlist\_to\_songs function, the sql query uses an inner join query to achieve a result similar to the other two functions. The data is then loaded into the table.

#### Playing Songs

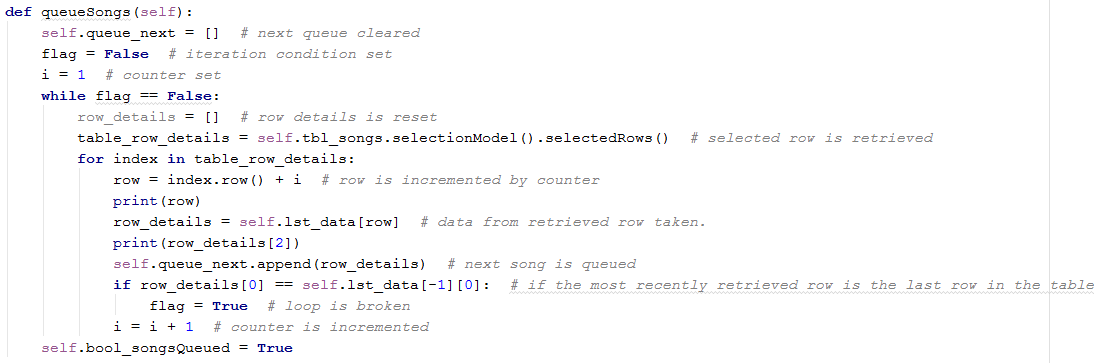
The fourth function, play\_song, differs slightly:



This function retrieves the relative details from its parent procedures, including the song name, its location in storage, its ID and its number of plays. The song’s number of plays is incremented and the record is updated with this new value. After this update query has been committed, the song’s location is used to load the song into the mixer and then player. Then, the songsQueued variable is checked. If the songs have not been queued, the queuesongs procedure will be run.

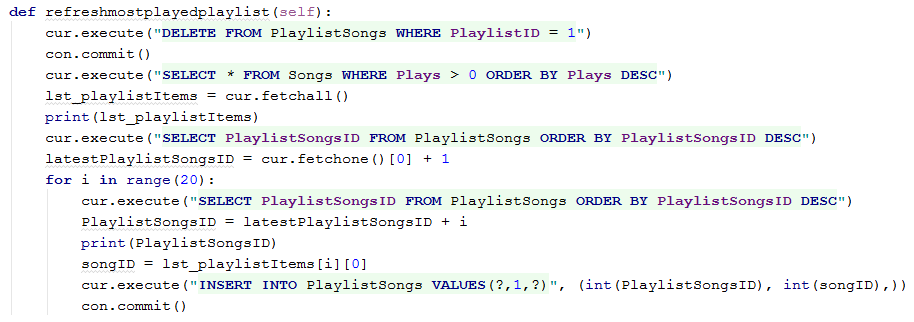
After this if statement, the most played playlist is refreshed to include the updated statistics. Then the next song procedure is run.

Here is the code for the queuesongs procedure:



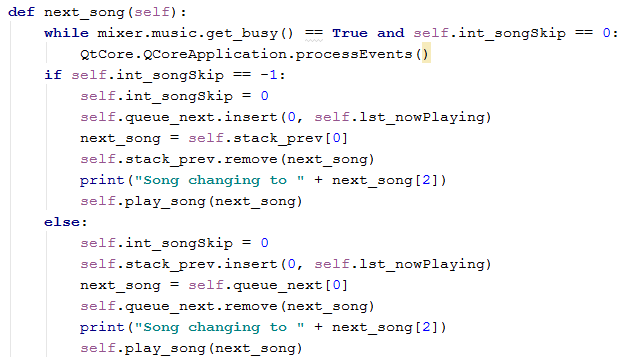
This section of code is reused from the retrieve row procedure, with some changes. Iteration is used to retrieve every row under the row that the user clicked.

Here is the code to refresh the most played playlist:



To start this procedure, the songs currently related to the most played playlist must be deleted. An SQL query is used to remove all fields in the PlaylistSongs link table where the playlist ID is that of the Most Played Playlist (which is 1). After the query is committed, another query is executed to select all songs that have been played (where plays>0), ordered by plays highest to lowest. The results of this query are then set to the lst\_playlistItems variable. Then, the highest current PlaylistSongsID is retrieved from the database. An iterative loop is then performed to insert the first 20 fields in the lst\_playlistItems. The new PlaylistSongsID for each iteration are created by adding the previously highest playlistsongID to the counter for the iteration. At each iteration, the changes are committed.

Here is the code for the nextsong procedure. This procedure is used to detect when the next song must be loaded into the mixer:

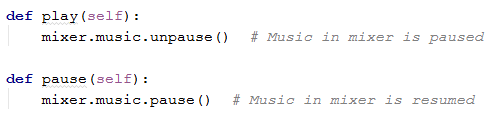


Firstly, this procedure includes a constant while loop, meaning that the code will keep repeating until the mixer is no longer playing a song or until the user skips the song. As this loop spends most of its time activated, a line of code had to be used to allow the PyQt interface to function. This is because python is an interpretive language and (normally) only processes one line at a time. Without the line “QtCore.QCoreApplication.processEvents()”, the program would crash as soon as a PyQt object was clicked or selected.

#### Playbar Buttons

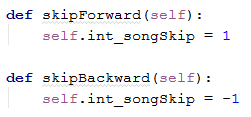
As the initialisation code shows; the play, pause and skip buttons have been programmed. Here are the procedures that they lead to:

Pause and Play



These two simple procedures allow the music to be played or paused, depending on which button is activated.

Skip Forward & Backward



These two procedures allow either the next or previous song to be played, depending on which button was pressed. The procedures change the state of the songSkip variable. This change is detected by the nextsong procedure (which have already been mentioned) which actually skips the song backwards or forwards.

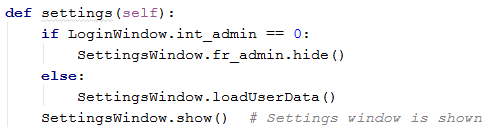
#### Search Function



This code starts by retrieving two of the user’s inputs: first, the user’s search term, in string and second, the search criteria (e.g. album, artist, song or playlist) from a dropdown box. Multiple if statements are then used depending on the inputted search criteria, each statement uses a query to retrieve items similar to the user’s search term in the table corresponding to the user’s search criteria. This procedure uses multiple wildcards in the LIKE queries to retrieve an estimated result.

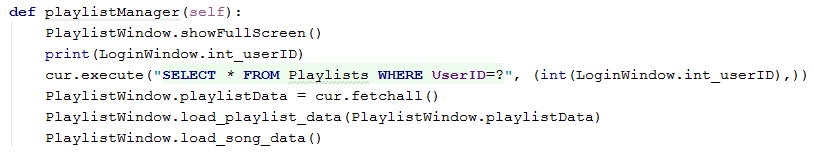
#### Exit/Transition Codes

Settings Window



This section of code performs two main tasks. First it validates whether the user who is currently using the program is an admin, using encapsulation from the LoginWindow object. If the user is an admin, the admin section of the settings window is shown and its user data is loaded. Otherwise, the admin section is hidden. After this validation, the Settings Window is shown to the user.

Playlist Manager Window



This section of code’s main purpose is to show the playlist manager window. It set’s the window’s behaviour to take up all of the screen. Then, a query is performed to retrieve all of the playlists that the user owns (where the userID in the playlist table is identical to the current user’s ID). The results of this query are then fetched and loaded into the playlist table.A procedure is then called to load the song data into the songs table.

Program Exit Code



This procedure is activated when the exit button is clicked (see the initialisation code for this class) and simply closes down the entire python program. This decision was made for two reasons: first, the program was made full screen and this button was required to allow the user to close the program and second, if the program was closed without any exit code, any music playing would keep playing. Closing the window only closes the Qt or UI section of the program, the mixer must also be closed to fully end the program.

## Settings Window Class

### Project Decomposition

#### Import Settings

For this section of the class. The program must accept an directory as a string from the user. This input is then validated to make sure that the directory actually exists. If the validation If the validation is successful, then the selected directory will be scanned and all of the mp3 files in the directory will be located. For each of these files, the ID3 tags will be read and inputted into the database. If a new artist is found, it is added to the artists table, is a new album is found, it is added to the albums table and when a new song is found, it is added to the songs table. If any of the information cannot be found, it is added as a null value into the record.

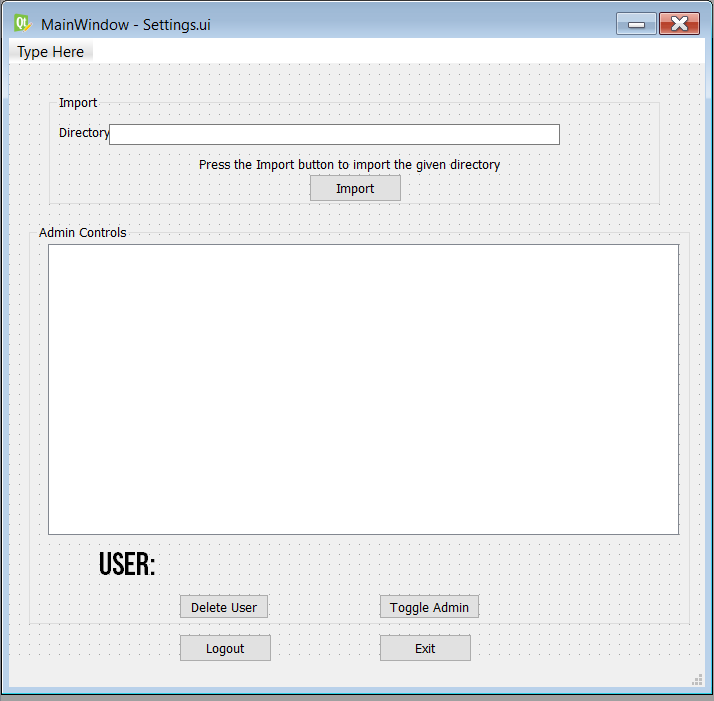
#### Admin Settings

For this section of the class. The code will provide a table that will contain all of the users (excluding the user currently using the program). The user will have the ability to select fields from this table and perform tasks. These tasks include: toggling the selected user’s administrator privileges and deleting a selected user’s account.

#### Other operations

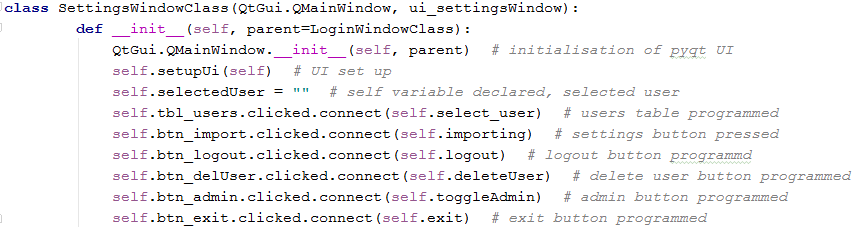
### Final Code

#### User Interface



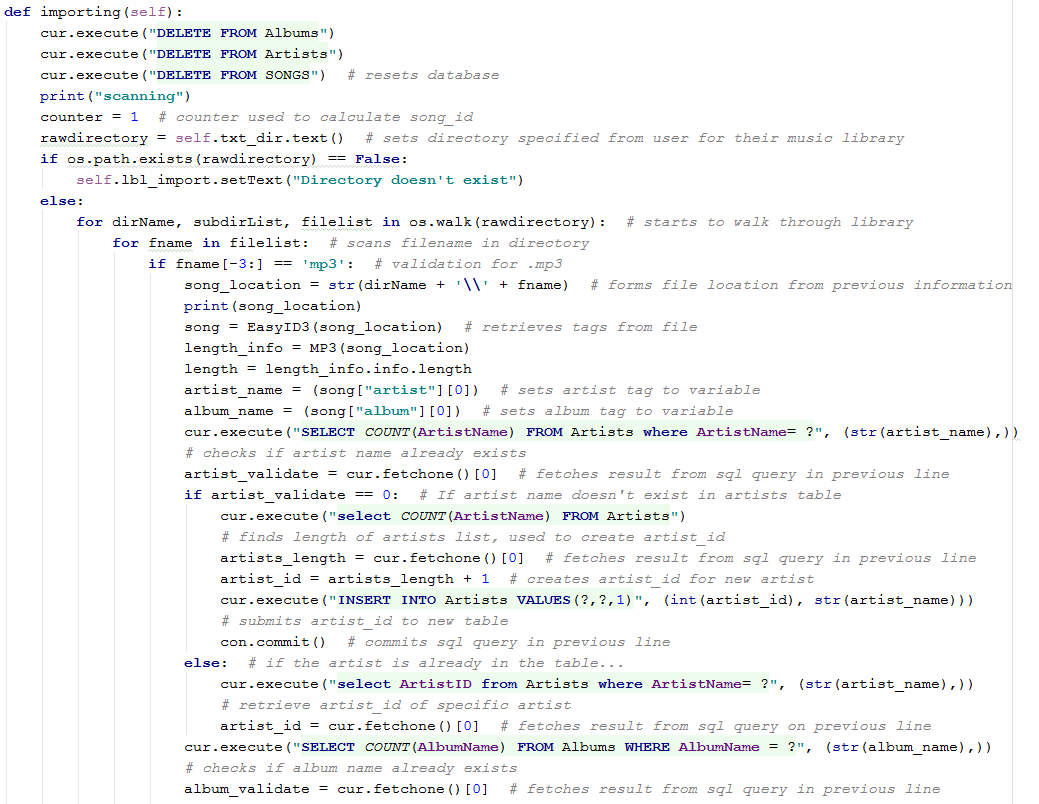
#### Initialisation Code

Here is the initialisation code of the Settings Class:



#### Import Code

This code has been amended to validate that the directory inputted by the user actually exists. The code now also checks the format of the track number, if the track number is in the form “01/07”, only the first number in the fraction will be noted:

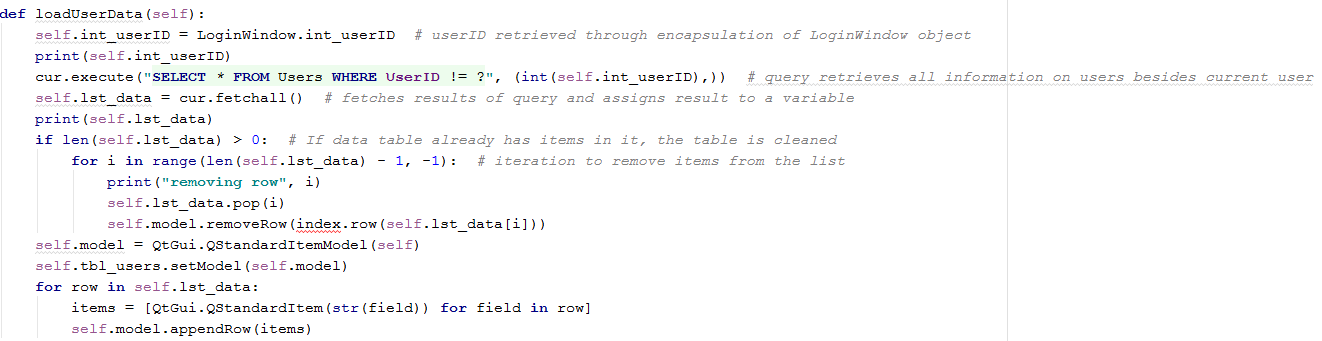




The annotations for this code give a step by step guide through how the code works. The key line to point out is the first if statement. This statement is the first piece of validation and validates whether the directory actually exists. Furthermore, after the track number has been read from the ID3 tags, the location of the slash (if one exists) is found and the string before it is separated so it is only number. The track number variable is then set to an integer so that it can be inputted into the database.

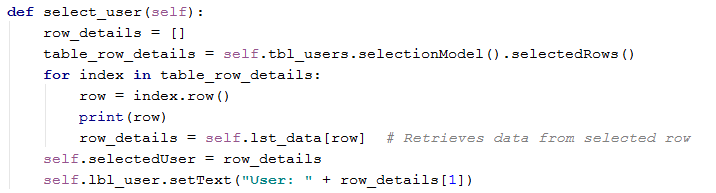
#### Admin Settings

Here is the procedure that loads the user data into the users table:



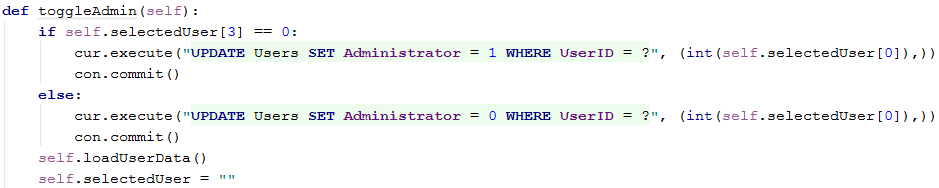
This method shows the use of polymorphism. This code is a slight manipulation of a previous procedure to load data into the songs table in the Main Window Class. The only differences are the query that is used and the name of the table that is being read to.

Then the code for when a user is selected in the table was written, this procedure is activated by the tbl\_users.clicked.connect line in the initialisation:



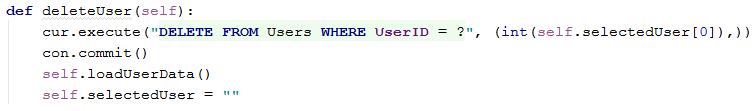
Once again, this code has been reused to better fit the problem to be dealt with. This procedure is identical to the retrieve row method in the Main Window class. However, this method finds the row in the users table rather than the songs table. Furthermore, after the row has been retrieved, the user’s name is displayed on a label so that the User knows which user he is about to edit.

After the select user method, the toggle user method was written:



This procedure starts by validating the details of the selected user. If the selected user is already an administrator (e.g. Administrator = 0), then their record in the Users table is updated to have an administrator value of 1. Else, the record is updated to have an administrator value of 0, the changes are then committed. After this, the user data is loaded again to update the information in the table. The selectedUser variable is also reset to avoid any mistakes that could be made by the user.

Then, the delete user method was written:



This method starts by using an SQL query to delete the user record corresponding to the selected user’s userID. The query is committed and the user data is reloaded to show the new change that has been made. The selectedUser variable is then reset to avoid any accidental actions that the user could make.

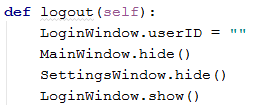
#### Exit Codes

The first exit code to be programmed was the exit method. The purpose of the code is to close the settings window. The method is activated when the exit button in the Settings Screen UI is clicked:



The code simply hides the settings window, leaving only the Main Window Screen open.

The next exit code is the logout method. The purpose of the code is to log the user out of the program and return to the login screen:



This method starts by resetting the userID variable from the LoginWindow object. Then, all of the windows that are currently open (e.g. the settings window and the main window) are closed. Then the login window is shown.

## Playlist Manager Window Class

### Project Decomposition

#### User Interface

The user interface will include two tables. One for holding playlist data and one for holding song data. There will also be buttons to change the playlist that the user is viewing, to create a new playlist and to exit the Playlist Manager window.

#### Initialisation Code

The initialisation code for this class will have to perform several tasks. Similar to all of the other classes in this program, the user interface will be set up. Then, the code will have ‘connect’ functions added to it for the buttons on the screen. This means that when the buttons are activated, their respective methods will be run.

#### Loading data into the Playlist and Songs table

Before any other actions can be taken with this class, the tables on the screen must have data in them. Both methods will have to retrieve results from an SQL query:

1. The load playlists method’s query will have already been executed in the playlistManager method of the Main Window Class. The query will retrieve all of the user’s playlists.
2. The ‘load songs’ method’s query will actually exist inside the method. The query will select all of the songs from the database.

The methods will then insert all of the data fetched from the database into their respective tables.

#### Retrieve Playlist Row

This method will need to be able to retrieve the contents of a row selected by the user in the Playlists table. Then, the code must check whether the row is a playlist or a song. If the row is a playlist, the playlist table will show all of the songs in that playlist. Else, the song will be removed from the selected playlist.

#### Add Song to Playlist

This method will work similarly to the retrieve playlist row section. However, once the row/song has been retrieved, the selected song is added to the selected playlist.

#### New Playlist

This method is activated when the new playlist button is pressed. The method must open the ‘new playlist dialog box’ screen.

#### Close Window

This method is activated when the close window button is pressed. This method closes the Playlist Manager window, leaving only the main window up.

#### Refresh Playlist

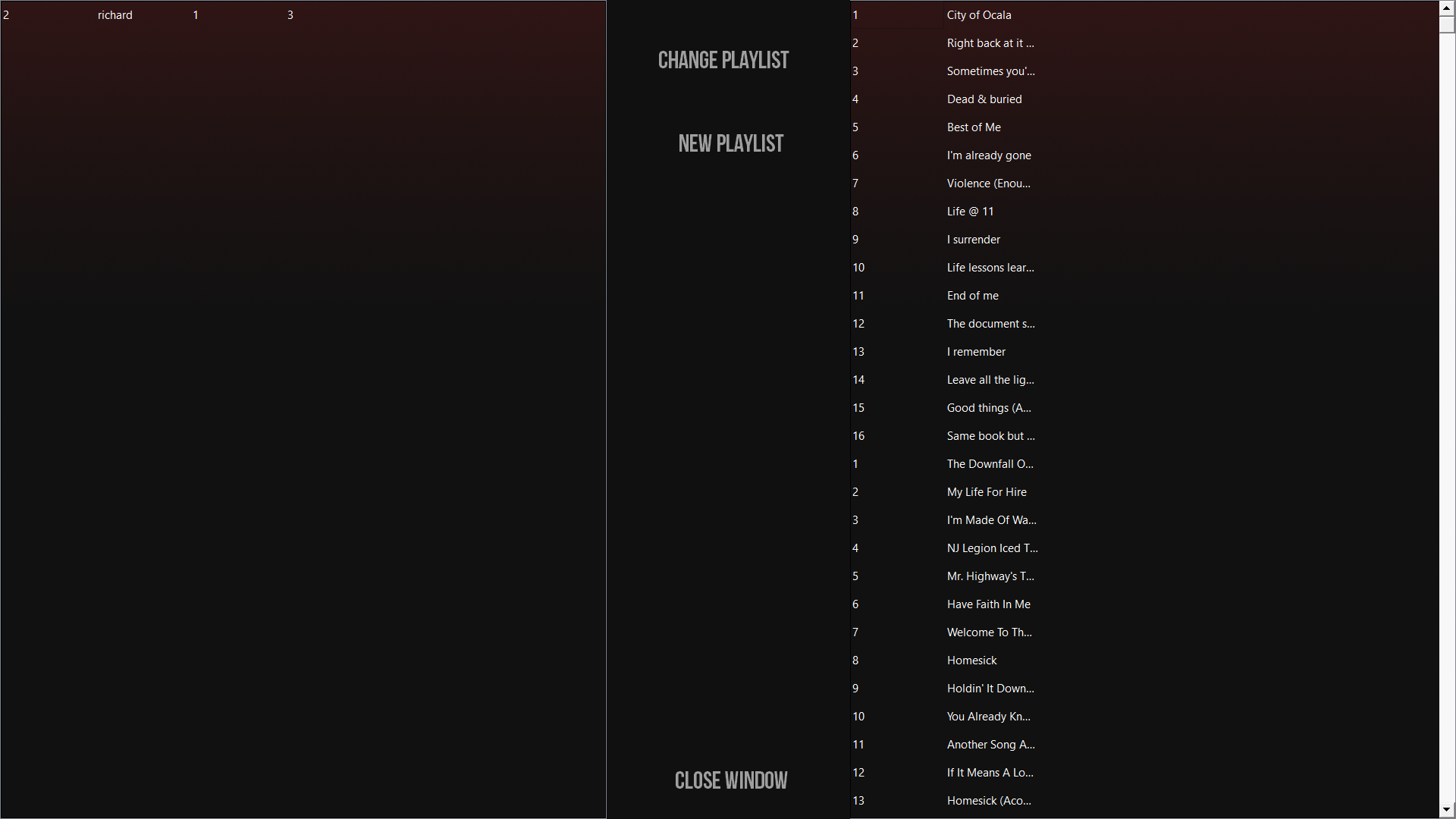
This method is activated when either a song is added or removed from a playlist. The code must refresh the playlist table to include the recent changes. This will be done be re-executing the previous SQL statement.

#### Playlist Reset

This method is activated when the ‘Change Playlist’ button is pressed. The method will execute a query that will retrieve all of the playlists, the ‘load playlist data’ method will then be activated with the retrieved data.

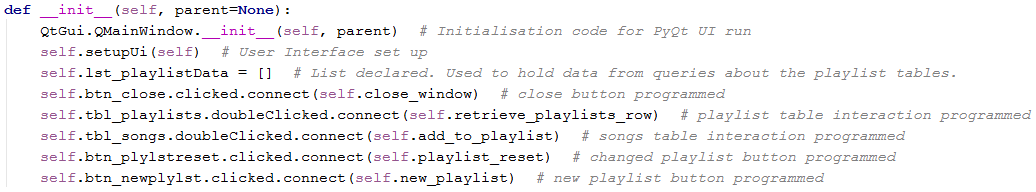
### Final Code

#### User Interface



Here is the final release of the user interface.

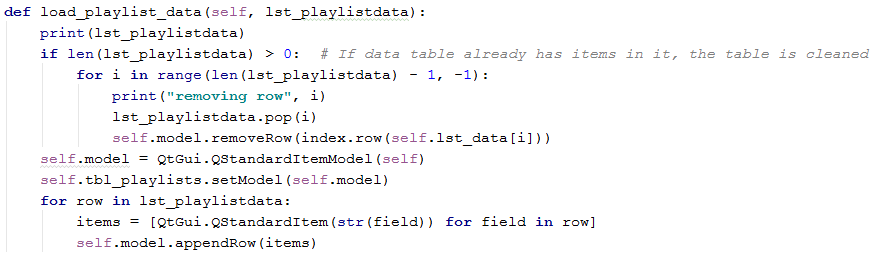
#### Initialisation Code



First, the User Interface was set up. Then the playlist data list (a self variable) was declared. The list is used between methods and holds all the

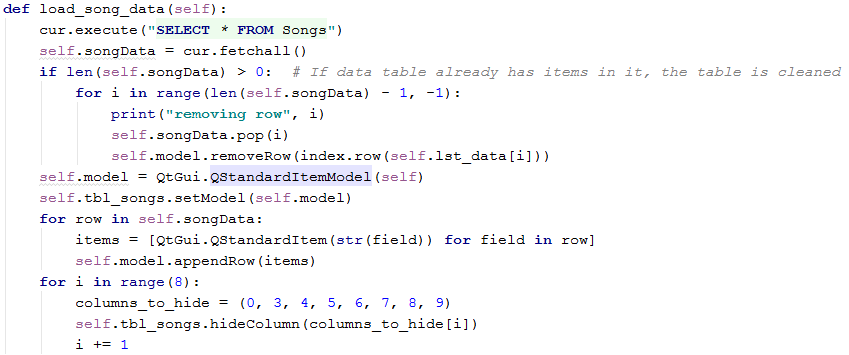
#### Loading data into the Playlist and Songs table

Here is the final code for the load\_playlist\_data method:



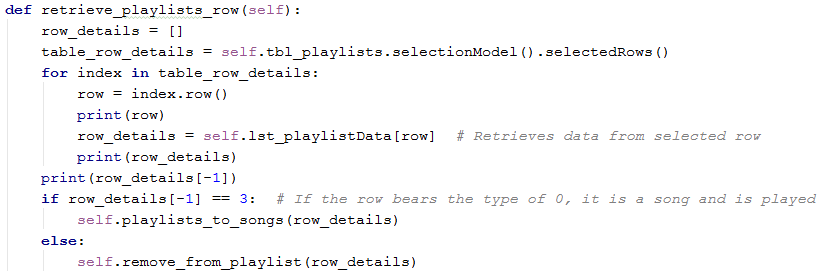
This code is a slight adaption from the load\_data method in the Main Window class. The changes made to this code were the names of the variables and the name of the table object.

Here is the final code for the load\_song\_data method:



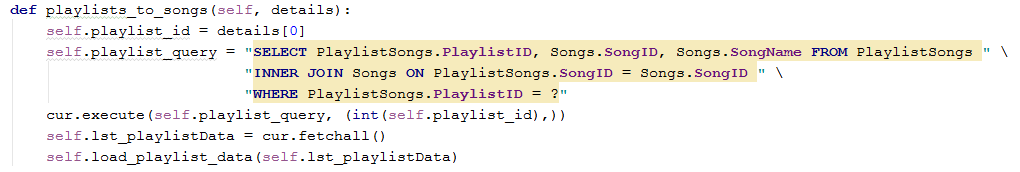
Once again, this code is an adaption of the load\_data method, showing how the solution uses **polymorphism** to ensure consistency and efficiency of the code. The code also includes another example of polymorphism at the end. The code uses an iteration loop that hides the selected columns (indicated by an array), this has been adapted from the hide\_?\_column methods in the Main Window class.

#### Retrieve Playlist Row



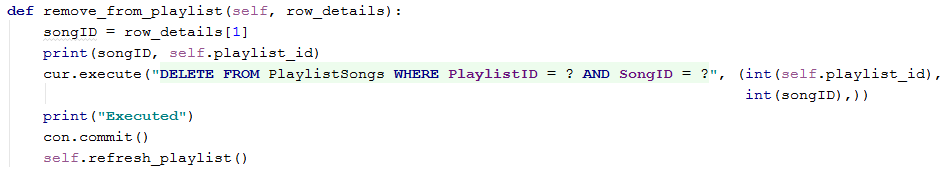
This section of code is a slightly adapted version of the retrieve row method in the Main Window Class. However, once the data has been retrieved, the details are validated using the type columns in the tables. If the type in the column is a 3, the row is a playlist and the playlists\_to\_song method is used. Else, the remove from playlist method is called.

Here is the code for the playlist to song method:



This method uses an **Inner Join** SQL query to select all of the songs from the Songs table that correspond to the songID in the PlaylistSongs table that correspond to the playlistID of the playlist selected by the user. The query is then executed and fetched, the load\_playlist\_data method is run using the fetched data.

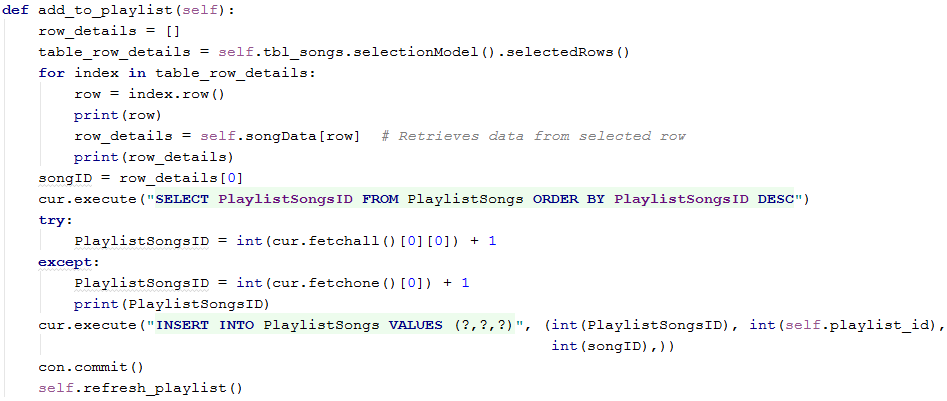
Here is the code for the remove\_from\_playlist method:



This code first assigns a variable for the songID, retrieving the value from the row that was previously retrieved. Then, a SQL query is executed that will remove the corresponding field from the PlaylistSongs table (it will delete the record where the playlistID and songID match the selected playlist and song). The changes are committed and the refresh playlist method is run (this method is shown later in the session).

#### Add Song to Playlist

Here is the code for the add\_to\_playlist method:



The first section of the code is an adapted version of the retrieve\_row method, showing how the code uses **polymorphism**. After the row has been retrieved, the songID (or first column) is retrieved from it. Then, a query selecting all of the PlaylistSongsIDs in the PlaylistSongs table (ordered by their PlaylistSongsID from highest lowest) is executed. A try and except clause is then used to retrieve the highest PlaylistSongsID and increment it by 1, creating the PlaylistSongsID for the new entry. An SQL query is then executed that inserts the a new record into the PlaylistSongsTable, using the selected SongID and PlaylistID and the newly created PlaylistSongID as inputs.

#### New Playlist

Here is the code for the New Playlist method:



This method simply shows the Playlist Name Dialog Box. This box allows the user to enter the name of the new playlist and will be explained later in the development.

#### Close Window

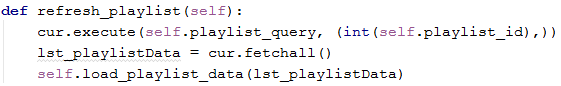
Here is the code for the Close Window method:



This code simply closes the Playlist Manager Window, returning the user to the Main Window screen.

#### Refresh Playlist

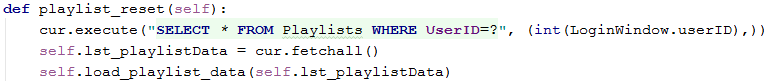
Here is the code for the Refresh Playlist method:



This code starts by executing the SQL query denoted by the playlist\_query variable, which is the most recent query executed by the playlist table. The data from the query is then fetched and the data is then loaded into the playlists table object (using the load\_playlist\_data method previously developed).

#### Playlist Reset

Here is the code for the Playlist Reset method:



The code starts by executing an SQL query to select all the playlists that the user has access to (meaning the playlists that the user has created). The data from the query is then fetched and loaded into the playlist data using the Load Playlist Data method (which has been previously developed).

## Playlist Name Dialog Box Class

### Project Decomposition

#### User Interface

The user interface must include all of the objects described in the objectives and User Interface Design, these include:

1. An input box to allow the user to input their desired Playlist Name.
2. A Button that allows the user to confirm their input.
3. A button that allows the user to cancel the request and close the window.
4. A label to instruct the user and notify them of any problems with validation

#### Initialisation Code

Like all of the other initialisation methods, this code will include the initialisation code that the user interface requires. The code will also set up listeners for the two buttons to check when they are activated.

#### Add Playlist

This method is called when the OK button object is clicked by the user. When called, the method will need to retrieve the user’s input from the input box and validate it to make sure the input is long enough (playlist names will need to be longer than 5 characters). If the validation fails, an error message will be shown from the label on the program. Else, a new playlistID will need to be created (by taking the highest current PlaylistID and incrementing it by 1). An SQL query will then need to be executed to insert a new record into the Playlist table (using the previously calculated playlistID, the playlistName entered by the user, userID of the current user and the value of 3 for the type (as the record is a playlist)). The changes to the database are then committed and the Playlist Reset method from the Playlist Manager screen is called. The Playlist Name dialog box is then closed.

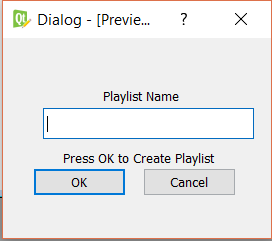
#### Close Window

This method is called when the cancel button object is clicked by the user. When called, the method will hide the Playlist Name dialog box, leaving only the Main Window screen.

### Final Code

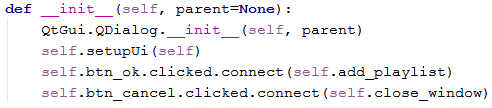
#### User Interface

Here is the final design of the user interface for the Playlist Name dialog box:



#### Initialisation Code

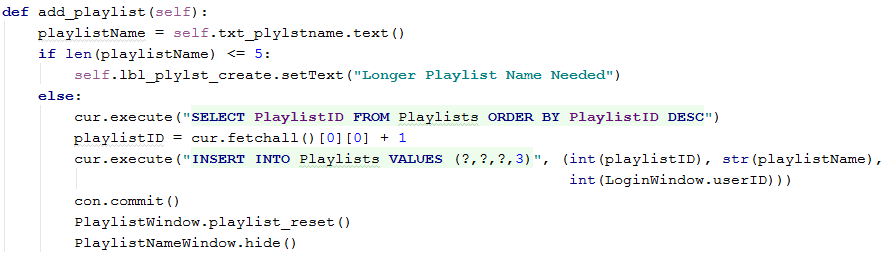
Here is the final code for the Initialisation method for the Playlist Name dialog box:



The code starts by running the initialisation code provided by PyQt to start the user interface. Then the listeners for the button objects in the user interface are set up to call their respective methods when they are clicked.

#### Add Playlist

Here is the final code for the Add Playlist method:



The method starts by retrieving the input from the txt\_plylstname object. The input is then validated to make sure its length is above 5 characters. If the validation fails, the text in the lbl\_plylst\_create object is altered to display an error to the user. Else, an SQL query is executed to retrieve all of the PlaylistIDs in the Playlists table, ordered from highest to lowest. The first record from this result is fetched and incremented by 1, creating the PlaylistID for the new Playlist record. Another SQL query is then executed to insert the values gathered (the PlaylistID, the Playlist name), the userID from the Login Window object and the value of 3 (for the type column, as the record is a playlist) into the Playlists table. After this query has been committed, the Playlist Reset method from the Playlist Window object is then called, in order to include the new changes that have been made. Finally, the Playlist Name dialog box is closed, returning the user to the Main Window screen.

# Evaluation

## Final Testing

In this section, I will test each section of the program to ensure that all objectives have been met.

### Database

#### Objectives to be tested

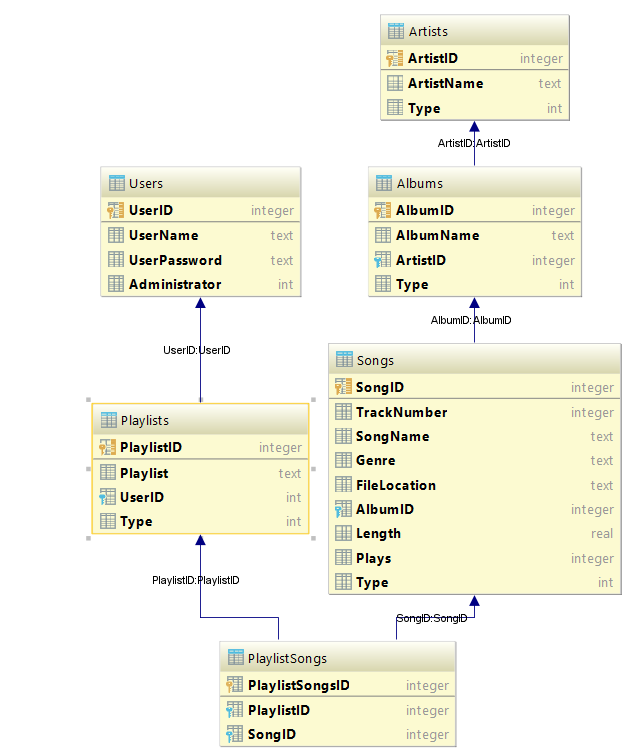
|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Success Criteria Tested** | **Input** | **Expected Output** |
| 1 | 1 | Open database file | All tables in the database will be related to one another and primary keys will exist for all tables. |
|  | 2 | Open database file | Songs table will exist |
|  | 3 | Open database file | Albums table will exist |
|  | 4 | Open database file | Artists table will exist |
|  | 5 | Open database file | Playlist table will exist |
|  | 6 | Open database file | Users table will exist |

#### Testing

Here are some screenshots showing proof of the database’s existence:



This diagram shows the actual existence of all of the tables in the database. Showing proof that objectives 2,3,4,5 and 6 have been met from the success criteria



This diagram shows the tables and the relationships between them (as well as the foreign keys that link them). The first entity of each table shown in the diagram shows the primary key. This overall proves that the solution has met objective 1 of the search criteria.

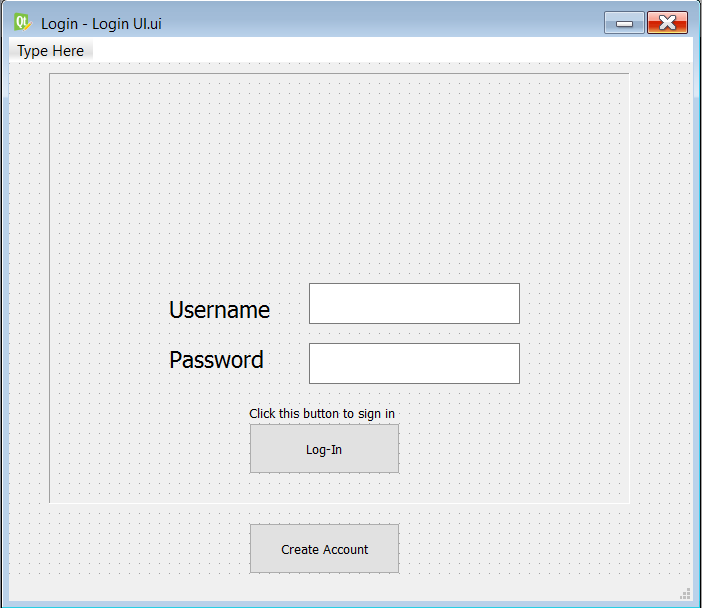
#### Conclusion

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Success Criteria Tested** | **Input** | **Expected Output** | **Output** | **Conclusion** |
| 1 | 1 | Open database file | All tables in the database will be related to one another and primary keys will exist for all tables. | All tables in the database are related to one another and primary keys exist for all tables. | Success |
| 1 | 2 | Open database file | Songs table will exist | Songs table exists | Success |
| 1 | 3 | Open database file | Albums table will exist | Albums table exists | Success |
| 1 | 4 | Open database file | Artists table will exist | Artists table exists | Success |
| 1 | 5 | Open database file | Playlist table will exist | Playlist table exists | Success |
| 1 | 6 | Open database file | Users table will exist | Users table exists | Success |

### User Interface

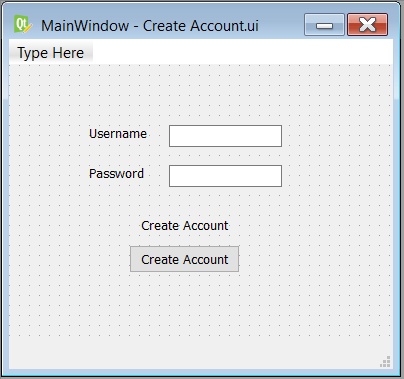
#### Objectives to be tested

#### Provide a login window



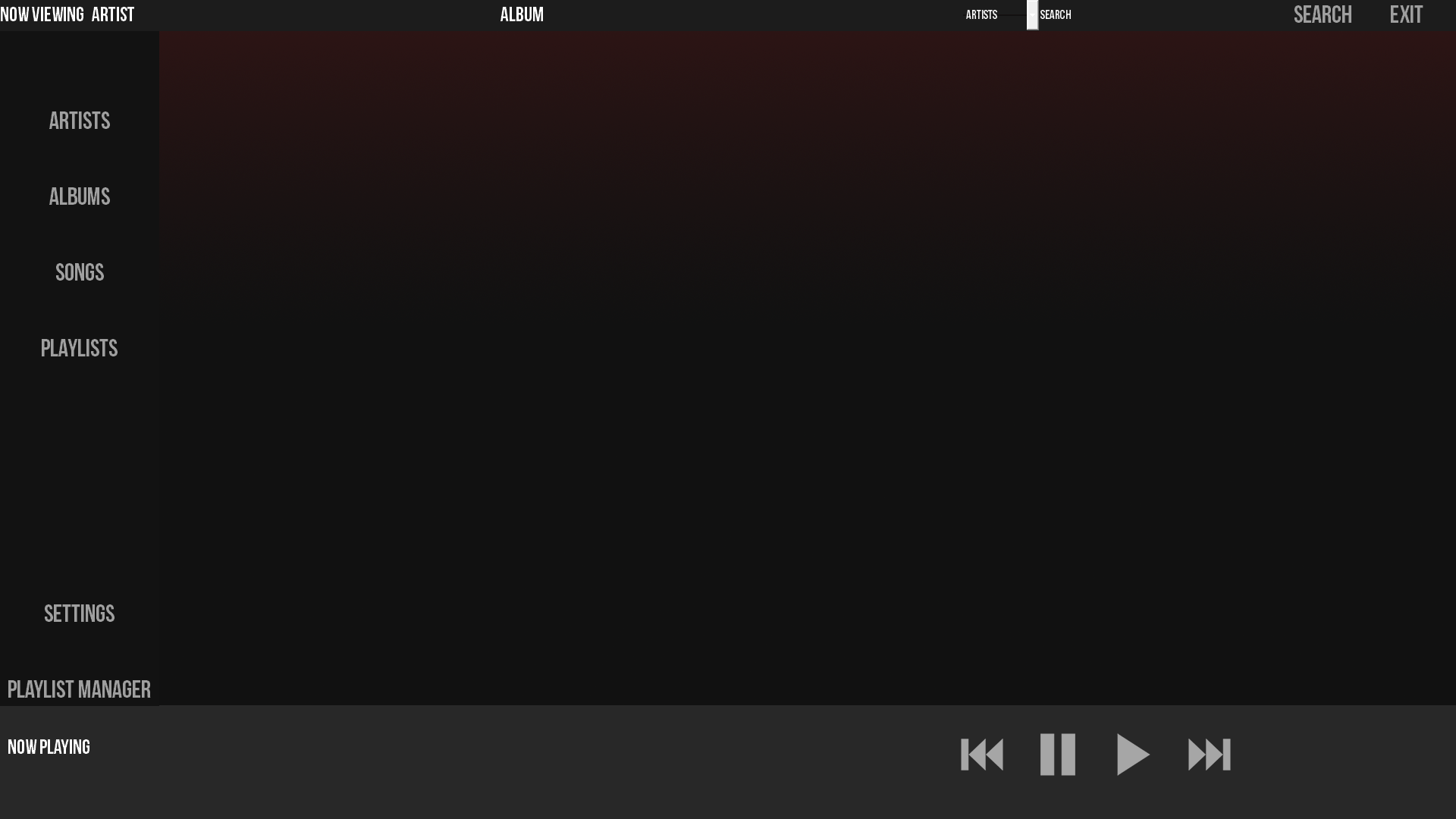
This screenshot shows the login window for the final program. Proving this objective.

#### Provide a create account window



This screenshot shows the create account window that is used in the final program, catering for this objective.

#### Provide a Media Player window



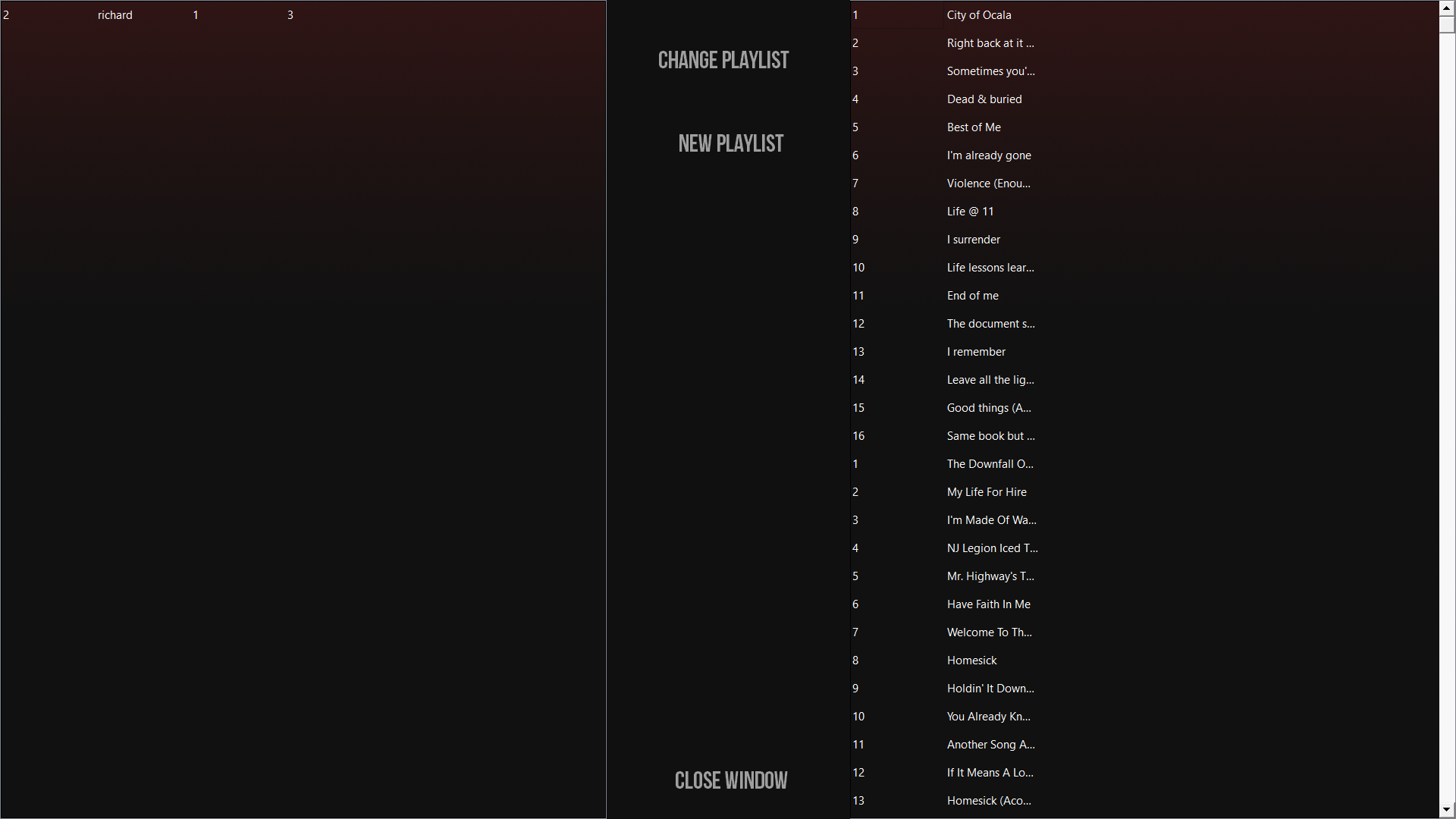
As this screenshot, the window for the final program exists. Proving this objective to be completed.

#### Provide a settings window



This screenshot shows the final screen for the settings window, completing this objective.

#### Provide a playlist manager screen



This screenshot shows the final screen for the playlist manager window, completing this objective.

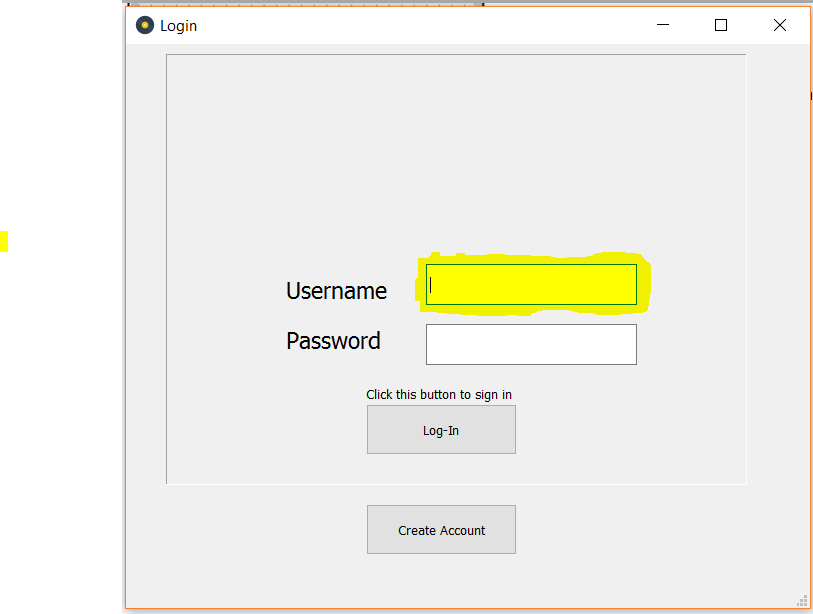
#### Provide a dialog box to change a playlist’s name



This screenshot shows the final screen for the Playlist ‘name change’ dialog box, completing this objective.

### Login Screen

### Provide an input box where users can enter their username



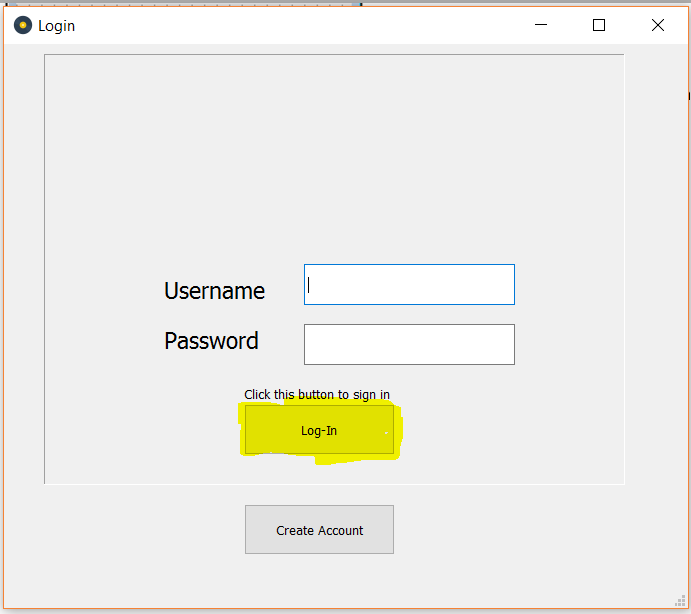
The highlighted section in this screenshot shows the input box discussed, completing this objective. The input box also has a label that notifies the user that this is where the username must be inputted.

#### Provide an input box where users can enter their password

#### 

The highlighted section of this screenshot shows the input box discussed in this objective. A label has also been used to denote what should be inputted in this box.

#### Provide a button that users can use to log in to the program



The highlighted section of this program denotes the button mentioned in this objective. Showing the objective complete.

#### Validate the username and password against the program’s database.

For this test, inputs will need to be made in the two text boxes. A test will be done with two correct details