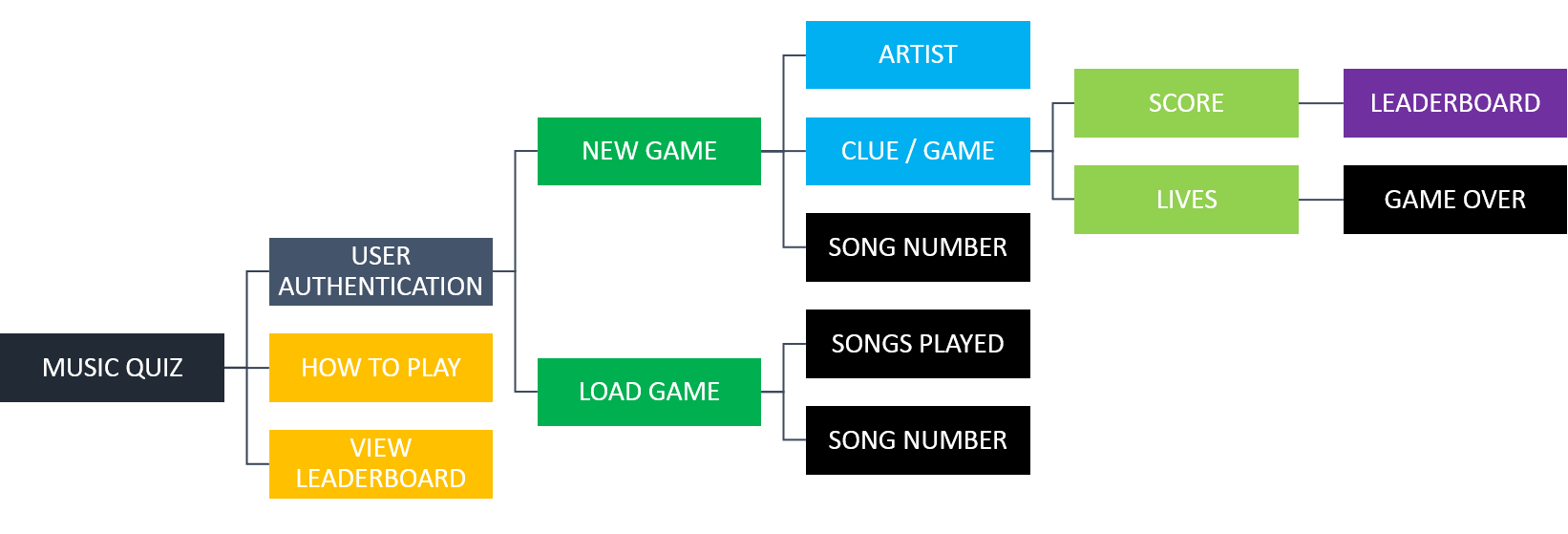
Task completed:

| Date started | Date completed |
| --- | --- |
|  |  |

Analysis

Problem Decomposition

****

Requirements

* Menu: when the program starts, a display will allow the user to:
  + Continue (“Play Game”)
  + Progress onto a ‘Leaderboard’ display
  + Advance onto a ‘How to Play’ instructions page
  + A GUI with buttons will be used to reduce the need for validation on inputs. It also seems sensible to use a GUI for a game that revolves around an art form.
* User Authentication: on clicking ‘Play Game’ a user will either be able to:
  + Login or Register using buttons.
  + An input field for a ‘Username’ and a ‘Password’ will be present. They will have a maximum length of 16 and 20 respectively and the username field will be traced to prevent uppercase entry. Disabling uppercase helps during validation with the SQL database.
  + The password field will be traced to display a real-time validation output at the bottom of the screen, telling the user what their password contains. A user cannot register until all requirements are met. This is helpful in creating a strong password (reducing chances of unwanted users playing their game) and allows a real-world user to understand what is required in a strong password.
    - When registered, details are submitted to an SQL database (after encrypting the password with MD5 – speed is chosen at the expense of security; however, strong password requirements should make cracking slightly harder). A hashed password prevents hackers from openly viewing sensitive details when accessing the database. The database could be hosted on a server making it more secure, however this would require the installation of modules thus making the program bulky. Instead, it will be saved locally (using the inbuilt SQLite module)
    - The SQL database is also used to fetch/pass existing user details during login/registration. Python’s string formatting causes unsafe data to be passed through the database, thus making the database vulnerable to errors and hacking attempts. To prevent SQL injection, placeholders will be used instead.
* Load Game: previous progress can be loaded if saved. A list of saves will be presented with:
  + Score, Lives, Date and Time of saving for each save (helps distinguish between different save games):
  + The ability to select or delete a save.
    - Warning popups will help prevent accidental deletion, helpful during real-world utility.
  + The ability to create a new save
  + A scrollbar in the case that there are more listboxes than possible on the frame. Estate needed will be calculated using a counter.
  + Allowing the saving and loading of progress is helpful as most real-world users will not be able to play the whole game at once.
    - Saves will be created and attributed to a specific username so that a user can only play their saved game.
    - Saves will also include a ‘played’ songs list so that songs are not repeated.
* Main Game: artist’s name will be displayed at the top. Boxed input fields will be used for each word in the song. A loop is used to fetch the first letter.
  + During the check, inputs will be sanitised to the same case as the correct word list. Similarly, whitespace will be removed and special characters ignored. Users are given leniency on correct input as errors can be made when typing. Often songs are known without certain special characters, thus allowing a fairer judgement.
    - On clicking submit (or Enter key), inputs will be checked to ensure they are not empty. An error message will be displayed. This prevents accidental entries. The same frame will be displayed without the lives counter being affected.
  + At the bottom, there will be a Lives and Score counter. Each session will begin with two lives (or guesses) which will be decremented by 1 on each incorrect answer until Lives is 0 at which point the Game Over screen is displayed.
  + There will be a Submit button (as an alternative to clicking the Enter key) as well as a Save and Exit button. A substitute to the latter is the close window button at which point the user is greeted with a popup to confirm exit without saving.
  + The game will follow the brief’s scoring system: 3 points on 1st guess (if lives = 2) and 1 point on 2nd guess (if lives=1).
  + Background processes:
  + A random song number is generated and added to a ‘previously played’ list.
  + This song is fetched from the SQL database and translated/unscrambled(song names are ciphered to prevent plain reading from database during gameplay)
  + A list is created from the original song matching the correct (expected) input (first letter, special characters and whitespace removed). This list will be checked against the user input list.
* Game Over: a simple display containing the eponymous text, a final score count and the option to play again or view leaderboard.
* Leaderboard: a list containing the headers: ‘Username’ and ‘Score’ – a 3D list also containing a number column. If there are more than 5 scores, a button to toggle all scores will be present which on-click will pack a scrollbar onto the frame. There will also a button to go back to the start.

Success Criteria:

* GUI
* Present user with button options to play game, view leaderboad or learn how to play.
* Allow login or registration for each user.
* Usernames should have a maximum character limit of 16 characters, passwords – 20
* Passwords should be encrypted before being passed into the database.
* A real-time output to inform the user on their password’s strength.
* Ability for users to save and load multiple games.
* A save game screen displaying details about each save.
* Boxed input fields for each word in a song.
* A scoring system as specified by the brief.
* Ability to generate a random song not previously played.
* A game over display.
* A leaderboard frame with ability to show top 5 scores.

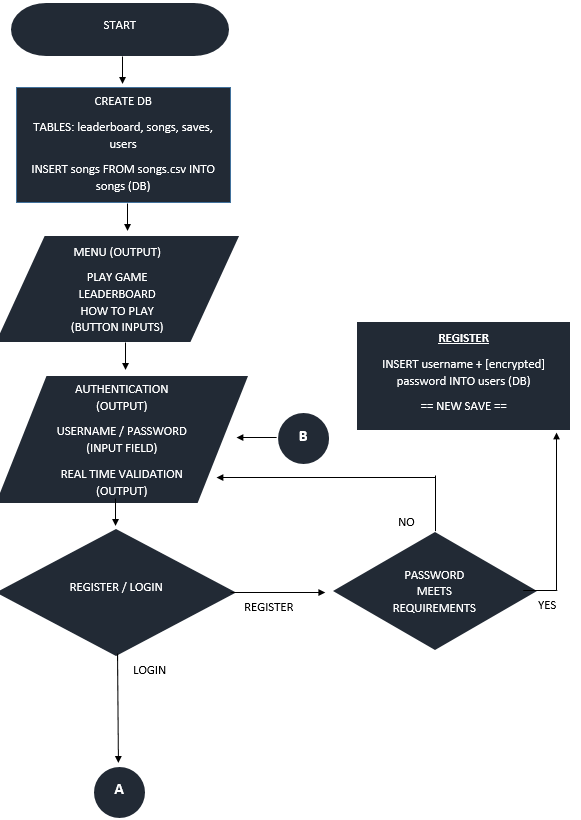
Testing:

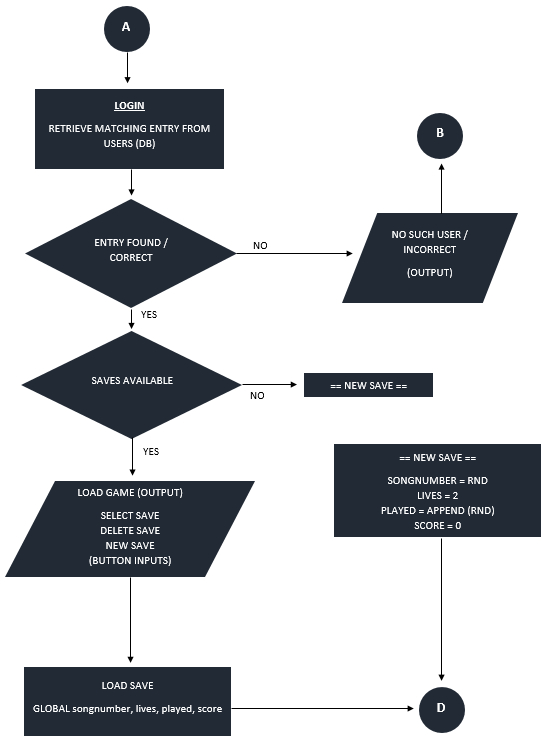
* Presence checks will be used during user authentication as well as game play to ensure there are no empty fields. The former prevents empty database entries and therefore errors and crashing during SQL execution; the latter, accidents.
* Presence checks will also be used to ensure there aren’t existing users in the database before entering new details as this would cause multiple entries causing the program to crash. It might allow users to change existing users’ passwords. In this case, a display will inform the user.
* Type checks will be used during registration to ensure the password meets the requirements for a strong password. This ensures that only the intended user can play their save games.
* Type checks will also be used to prevent whitespace entry into username/password fields. Whitespace may cause errors when passed through a database. Input from gameplay will also be sanitised to remove whitespace, this is helpful as mistakes can be made during input.
* Length checks will be used during username and password entry. The username field variable will be traced to slice its length to maximum of 16 (a suitable length for usernames). This is required to prevent extreme inputs. Passwords will have a maximum length of 20 characters (as a password length positively correlates with strength, thus a higher limit is needed).
* File not found errors will be checked for during initial start-up. Try and except statements will be used to ensure the program runs smoothly. Most errors will be accounted for by downloading the desired file from a saved location online. In the case that a connection cannot be made (for a crucial file), an error message will be displayed and the program ended.

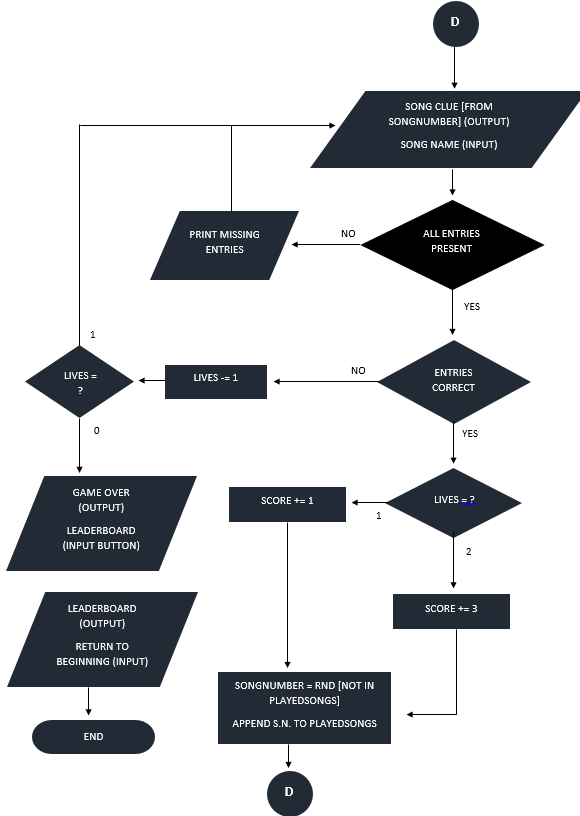
Design

Algorithms

**FLOWCHART**

****

****

****

**PSEUDOCODE**

**\*\*** This simplified version is loosely based on the expected program. The pseudocode is mainly for a CLI-based program \*\*

**LOGIN**

1. FUNCTION authenticate()
   1. username = INPUT("Enter your username")
   2. password = INPUT("Enter your password")
   3. IF username == “” OR password == ”” THEN
      1. PRINT(“Missing Entries”)
   4. IF user == db\_fetch THEN
      1. PRINT("Correct!")
   5. ELSE
      1. PRINT("Incorrect")

A sort of loop where function is repeated until user enters a valid option,

* + 1. authenticate()

1. ENDFUNCTION

**RANDOM SONG GENERATOR**

1. LIST playedsongs
2. songnumber = “”

WHILE loop repeats program until a valid number is generated. This is then appended to a playedsongs list to avoid repeats.

1. WHILE songnumber in playedsongs
   1. songnumber = generate(1,400)
2. ENDWHILE
3. playedsongs[x] = songnumber

**SCORING SYSTEM**

WHILE loop repeats question as long as both guesses are left

1. song = “Aaa Bbb”
2. lives = 2
3. WHILE lives != 0
   1. entry = inputfield(“A\_\_\_ B\_\_\_”)
   2. IF entry == song THEN
      1. IF lives == 2 THEN
         1. score = score + 3
      2. ELIF lives == 1 THEN
         1. score = score + 1

If guessed correctly, loop is exited and the outer function is restarted with a new song. If loop ends, Game Over frame is presented.

* + 1. newsession()
  1. ELSE
     1. lives = lives - 1

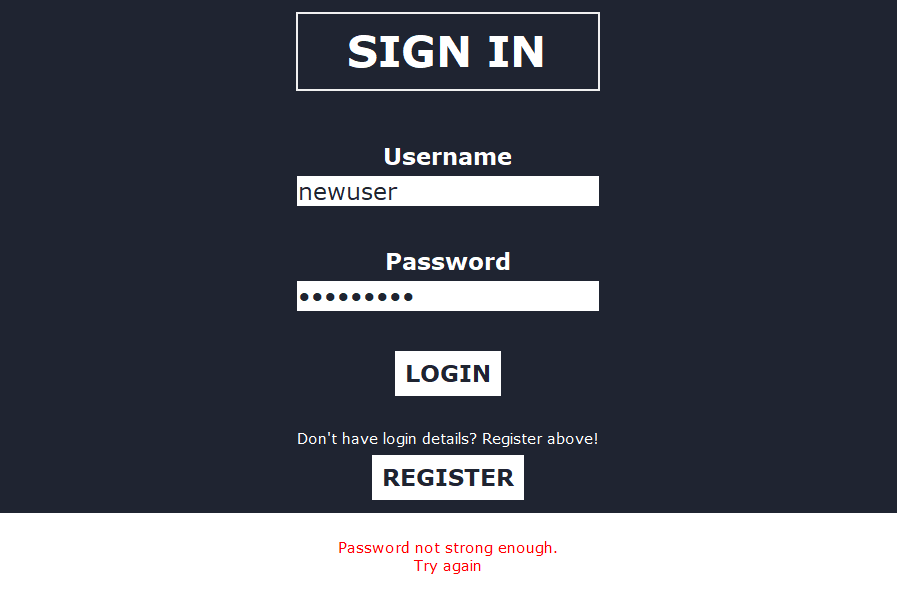
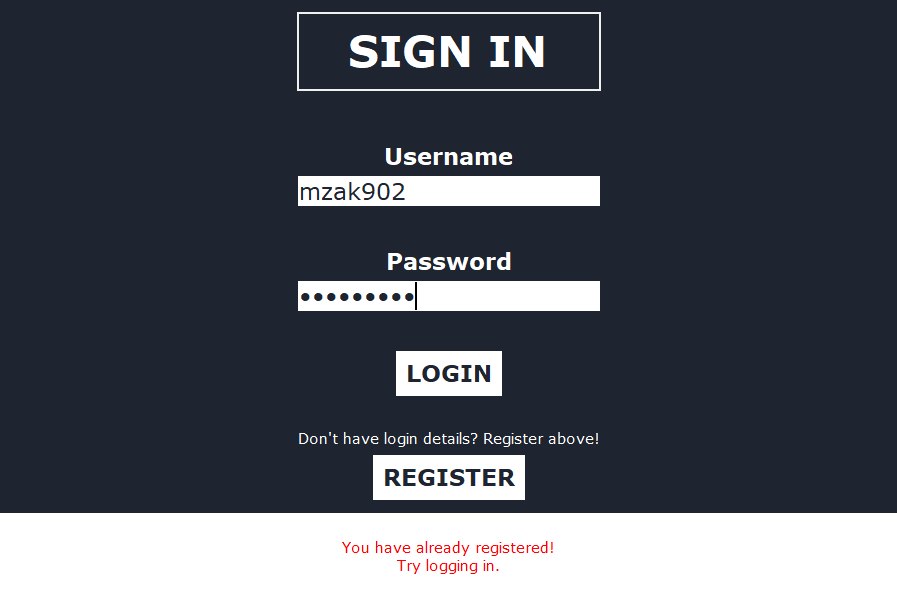
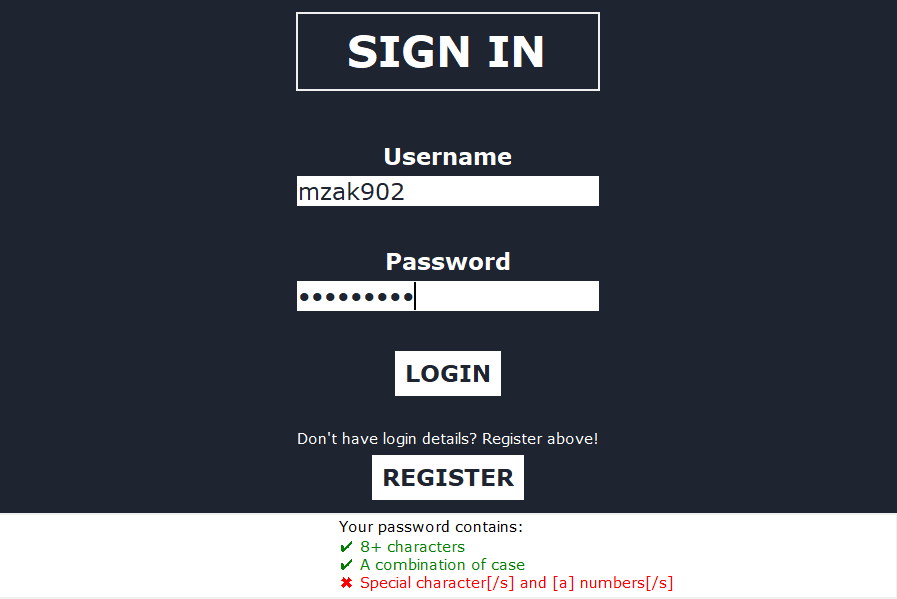
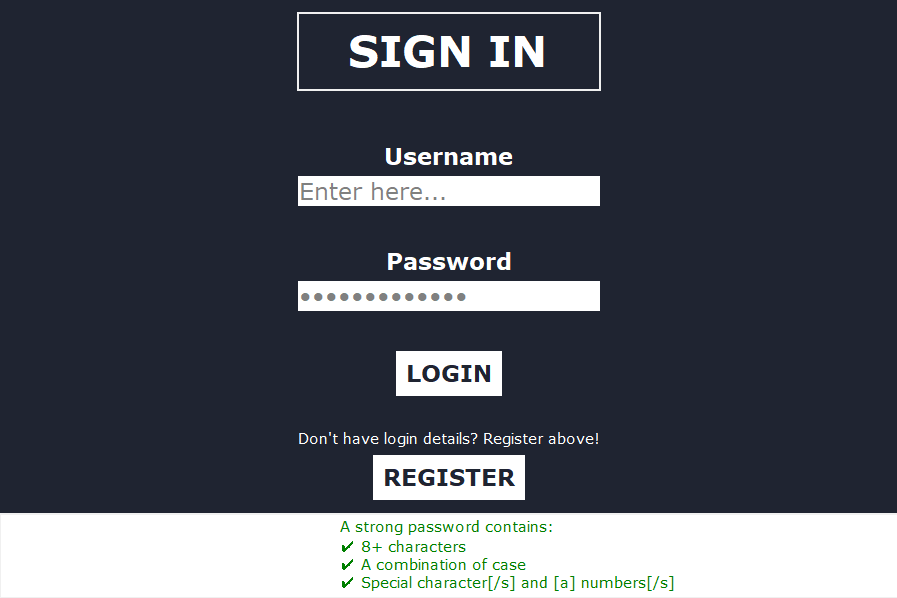
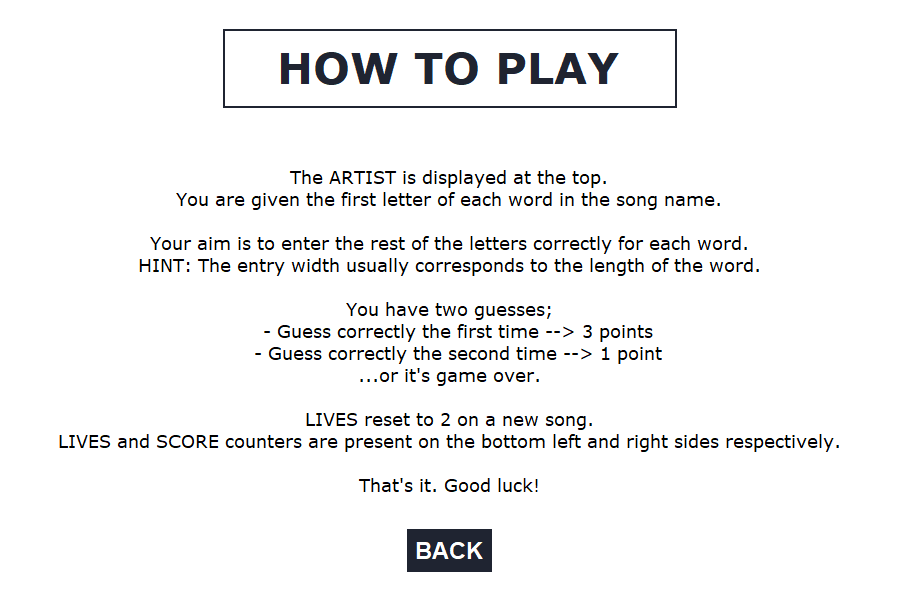
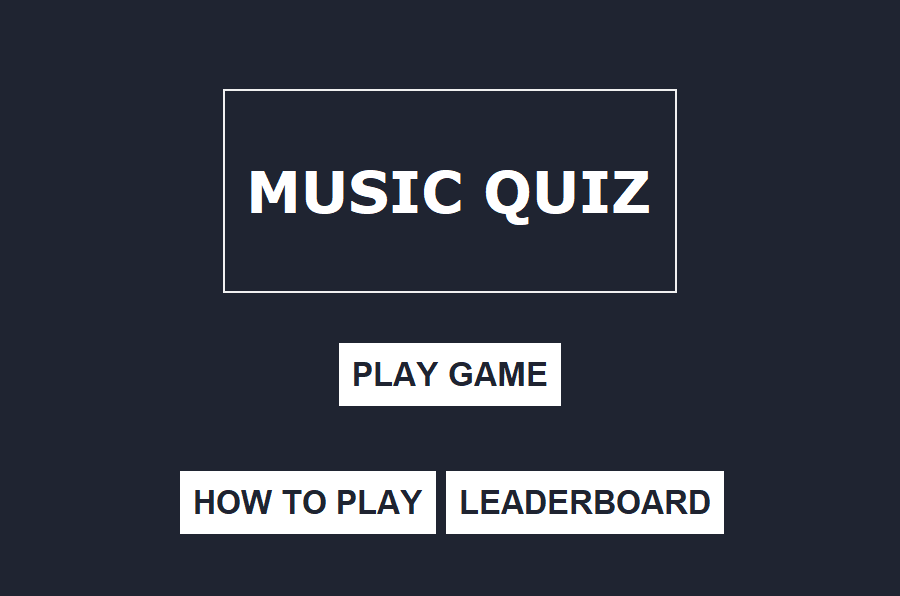
1. ENDWHILE
2. PRINT(“Game Over”)

**IMPORT SONGS INTO DB**

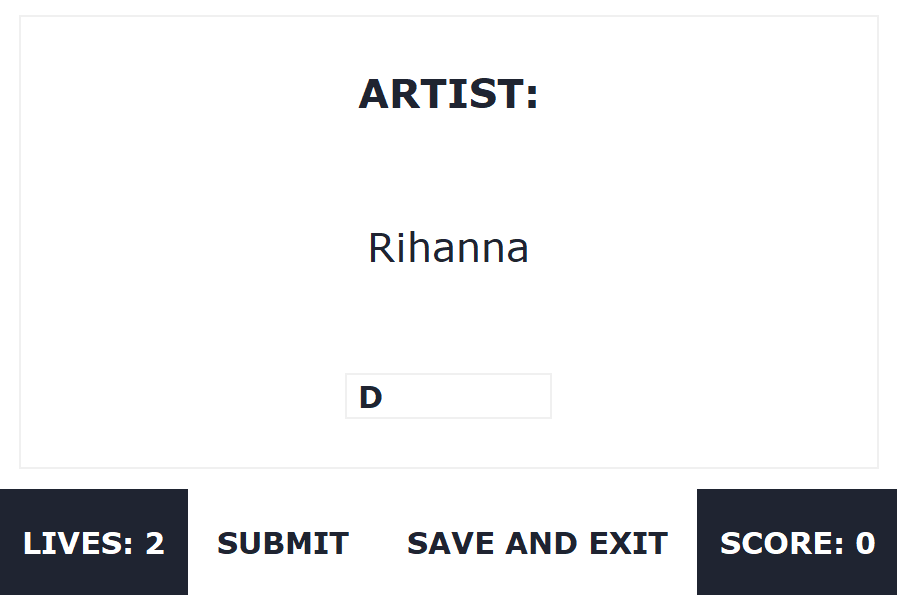
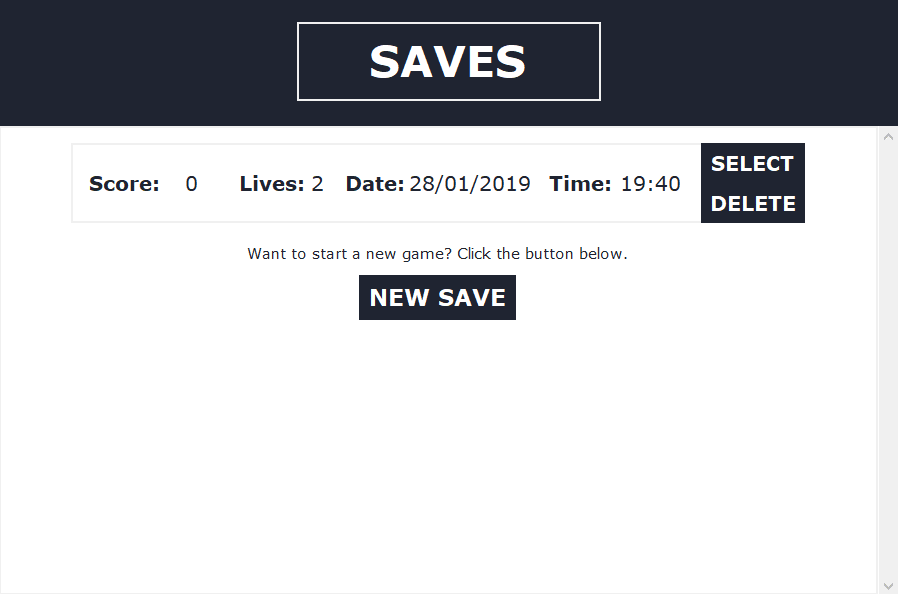
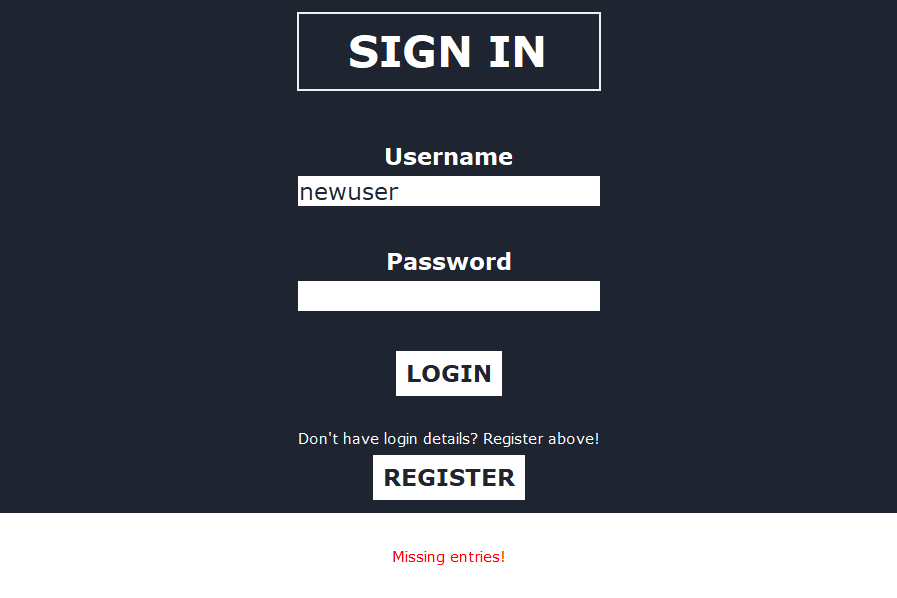
1. songlist = OPENREAD(“songs.csv”)

File is read and imported into database. This is easier to fetch from during gameplay,

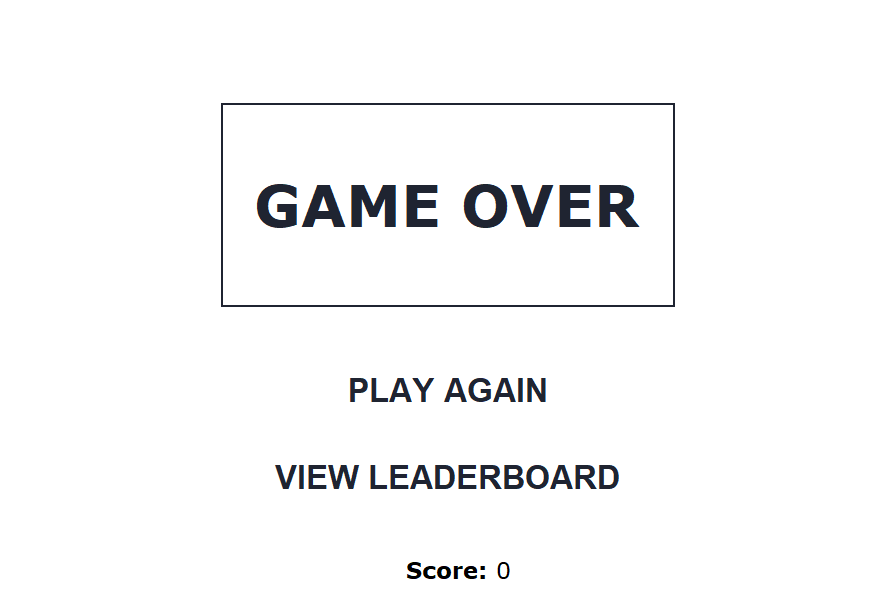
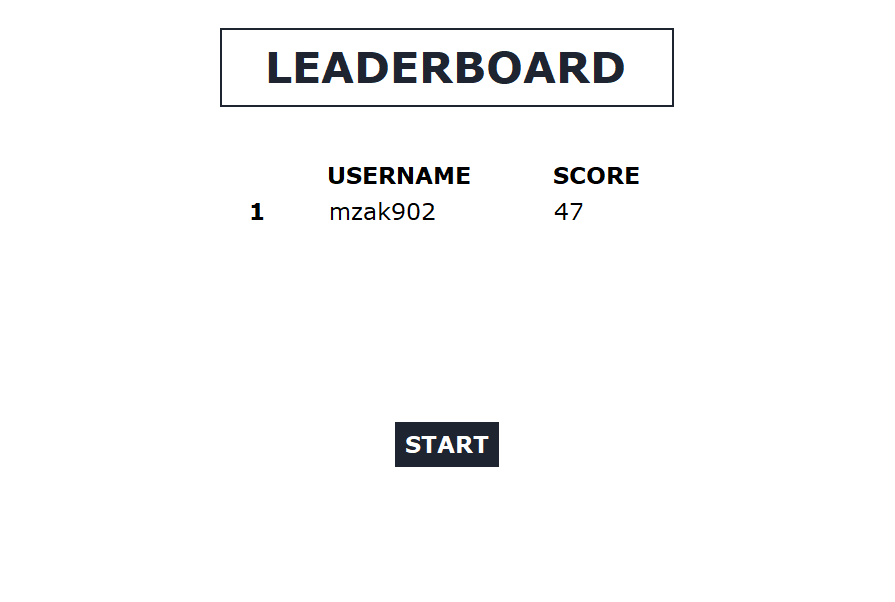
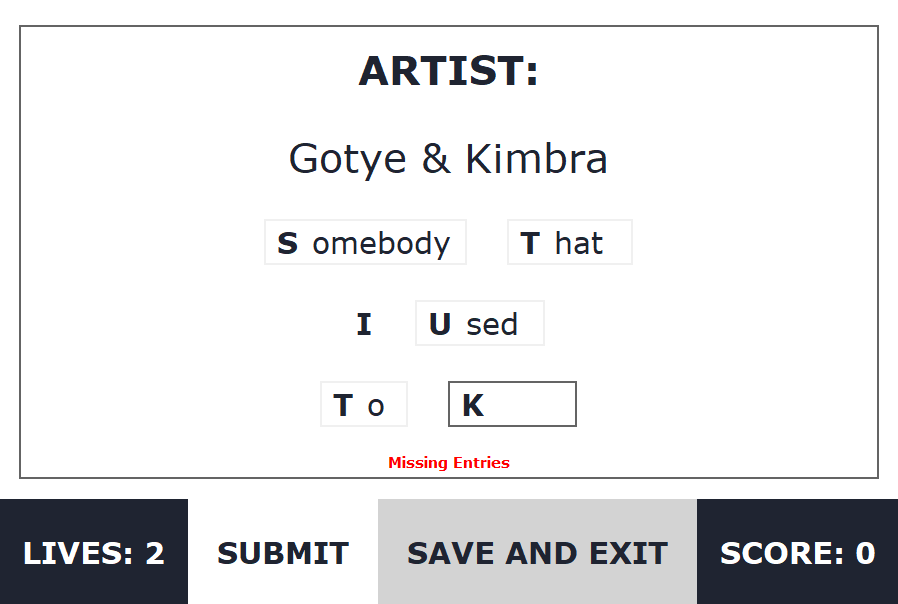
1. for row in songlist
   1. INSERT INTO db

USER INTERFACE

Login and registration frames were combined to decrease program size. Details are entered and button is clicked dependent on if user is logging in or registering as a new user. A ‘How to Play’ frame aids in playing the game in case the gameplay frame is not understood well.



The output display, as well as being used for real-time validation, will be used to show helpful messages notifying the user about specific errors. Messages will disappear after a set amount of time. Buttons will change colour on hover to help distinguish between them (bottom-left).

****

Testing

### SIGN IN:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | What am I testing? | What data will I use? | Normal/Boundary/Erroneous? | Expected Result |
| 1 | Check a valid input is accepted | mzak902  Password123\* | Normal | Accepted |
| 2 | Ensure empty inputs are not possible | No data | Erroneous | Inform user |
| 3 | Check 1 character username | z | Boundary | Accepted |
| 4 | Check less than 8 characters in password | BCabc\*1 | Erroneous | Decline, inform user |
| 5 | Check special characters in username | mzak\*\* | Robustness | Accepted |
| 6 | SQL injection possibility | DELETE FROM leaderboard | Robustness | Unperturbed |

### GAME:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | What am I testing? | What data will I use? | Normal/Boundary/Erroneous? | Expected Result |
| 1 | Entry of numbers | 123456 | Normal | Accepted |
| 2 | Ensure empty inputs are not possible | No data | Erroneous | Decline, inform user |
| 3 | Check whitespace | Song name | Robustness | Disregard (sanitised) |
| 4 | Check special characters | ABCabc\*\* | Robustness | Disregard (sanitised) |

### LOAD SAVE FRAME:

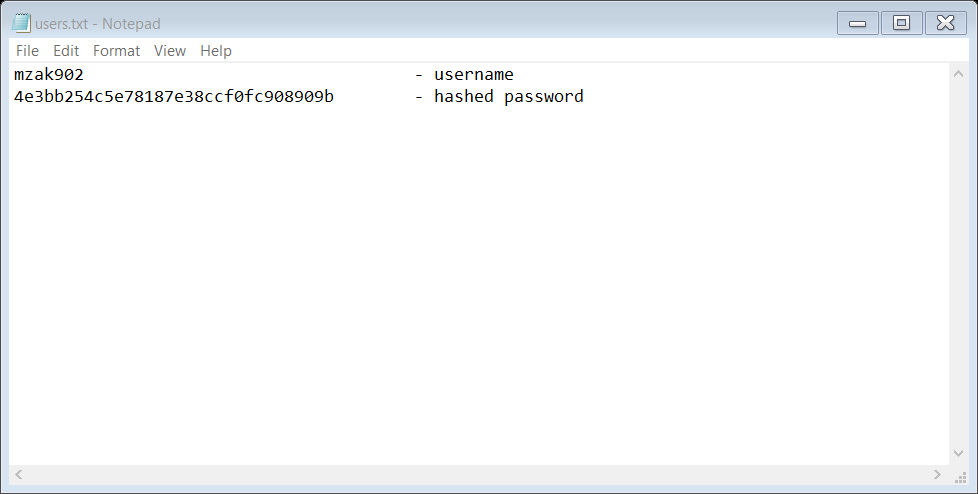
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test | What am I testing? | What data will I use? | Normal/Boundary/Erroneous? | Expected Result |
| 1 | Allow deleting of saves | DELETE | Normal | Item deleted, no repeats on frame (left over) |
| 2 | Allow selection of saves | SELECT | Normal | All data imported, no errors |
| 3 | Allow new save creation | NEW SAVE | Normal | Create new save without clashes due to old ones |
| 4 | Allowing deleting of multiple saves | DELETE x3 | Robustness | Accepted |

Variables, structures and files

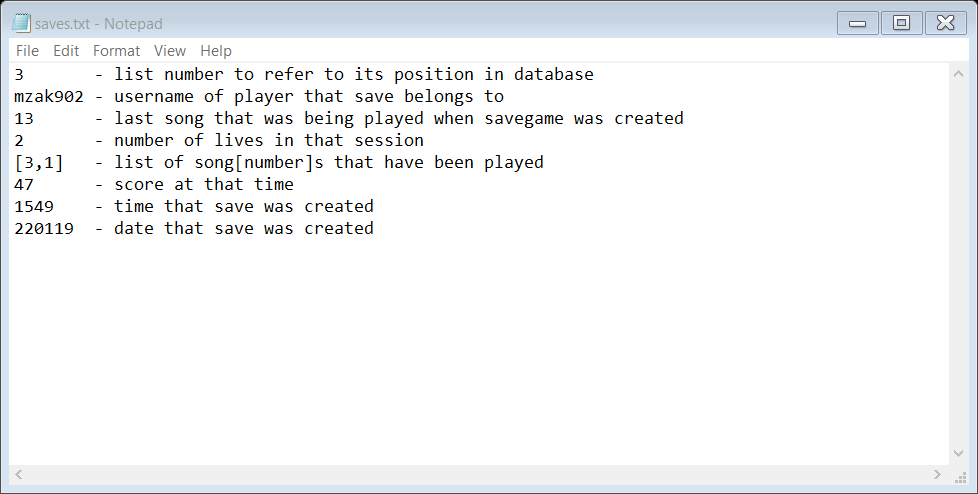
### VARIABLES

|  |  |  |  |
| --- | --- | --- | --- |
| Variable/List | Type | Use | Validation |
| username\_entry | String | Input field for username | > 1 character  < 16 characters  No whitespace |
| password\_entry | String | Input field for password | >= 8 characters  < 20 characters  No whitespace |
| songnumber | Int | Stores the list number in the database of the song to be played | Must not be in played[] |
| lives | Int | Stores number of guesses player has left. | N/A |
| played[] | List | Stores the songnumber value of songs already played to prevent reuse. | N/A |
| listno | Int | Stores the list number in the database of the save that has been imported or needs to be deleted. | N/A |
| artistname | String | Stores the name of the artist to be displayed for a specific song | N/A |
| songname | String | Stores the name of the song imported from database | N/A |
| wordsentered[] | List | Stores user input in each field (for each word) | Must not be empty  No whitespace or special characters  All lowercase |
| correctwords[] | List | Stores the input that is considered correct and shall give points to user | First letter removed  No special characters or whitespace  All lowercase |
| score | Int | Stores the number of points user has accumulated | N/A |

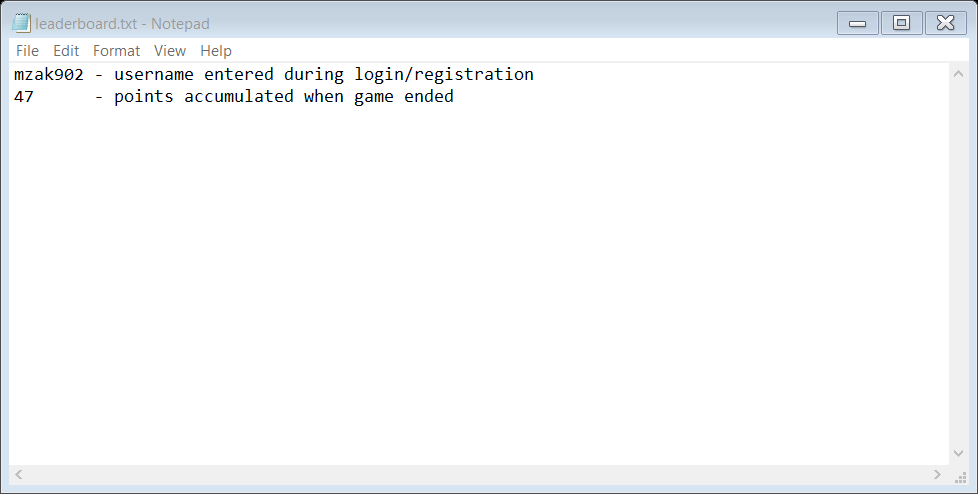
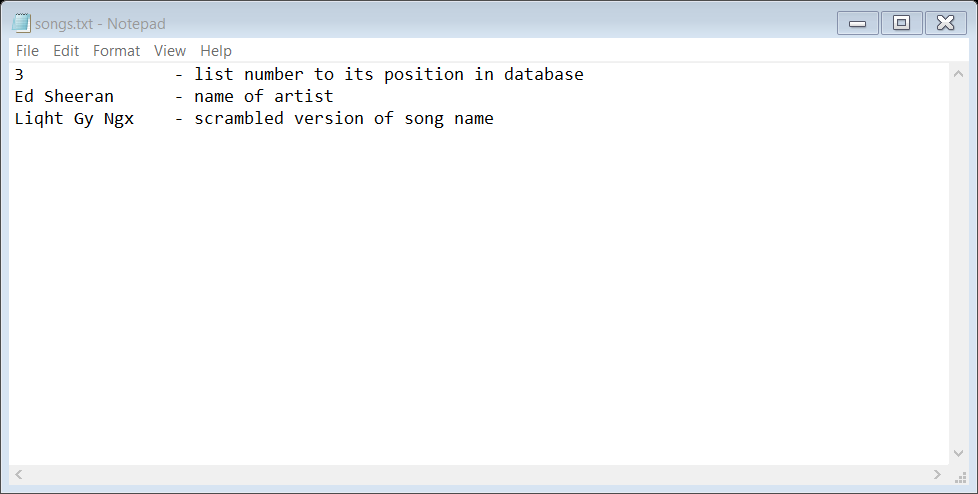
### FILE STRUCTURE

Inside the structured SQL database (details.db), there will be the following tables:

users – stores login details



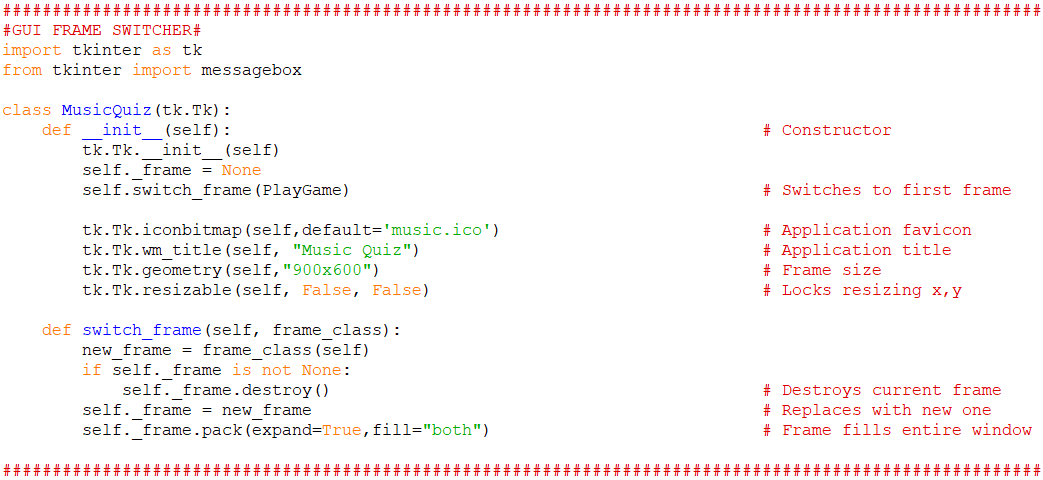
saves – stores progress

 leaderboard – stores all scores

songs – stores list of songs with artists name

Development / Testing

### GUI:



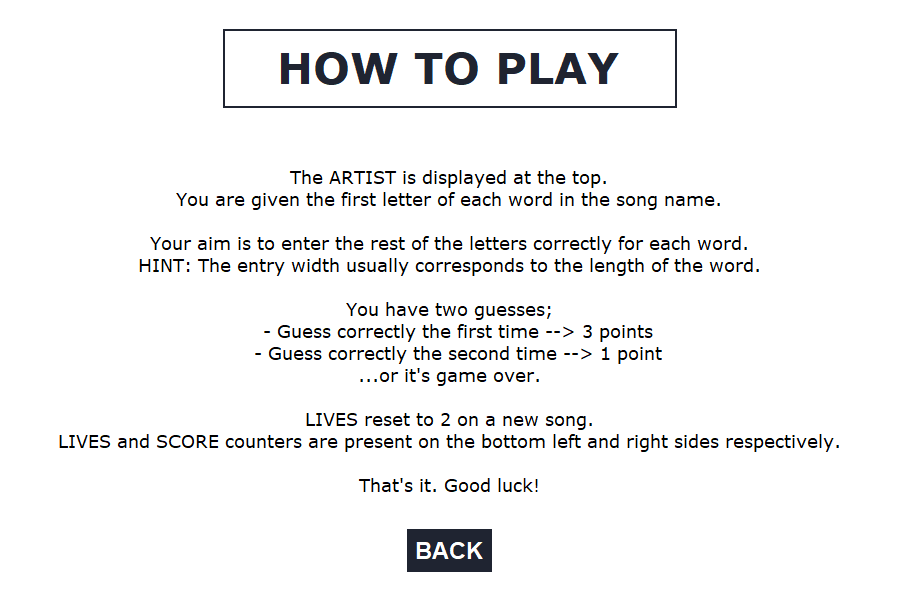
Classes were used to allow frame switching functionality. The boilerplate for this was found here:

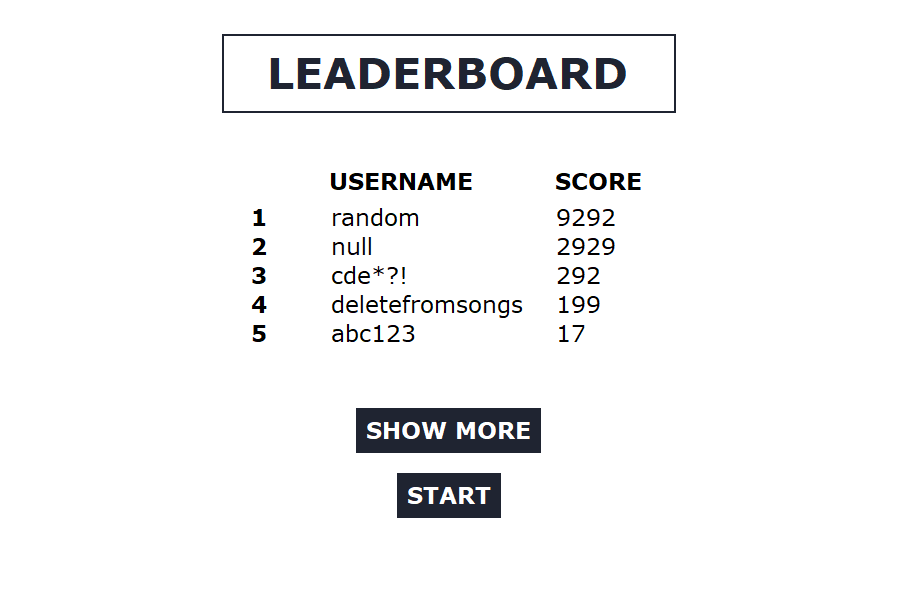
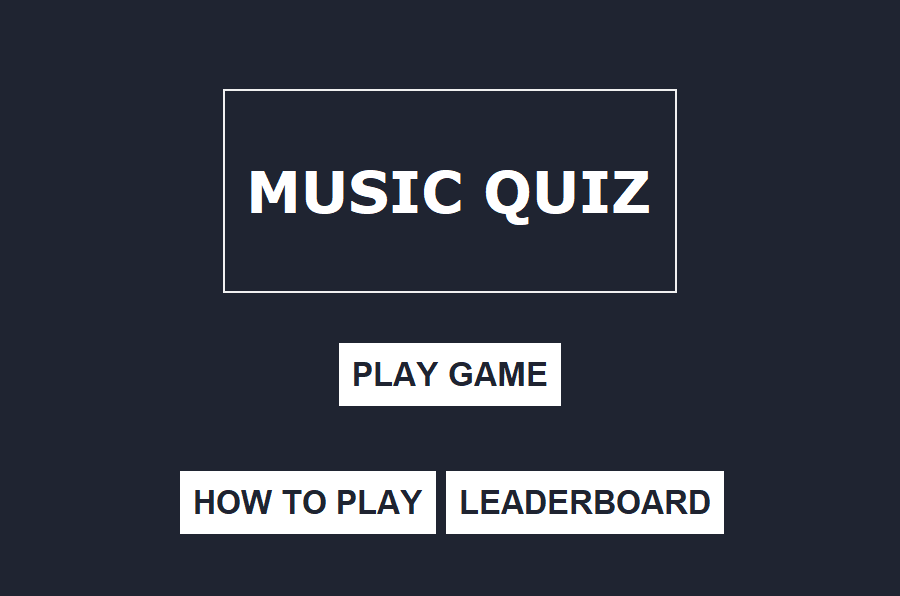
[**https://stackoverflow.com/a/49325719**](https://stackoverflow.com/a/49325719)

The use of this simplified frame switching as code did not have to be repeated every time a frame had to be created and then destroyed to build a new frame. This also sped up the program as frames were not on layered on top of each other.

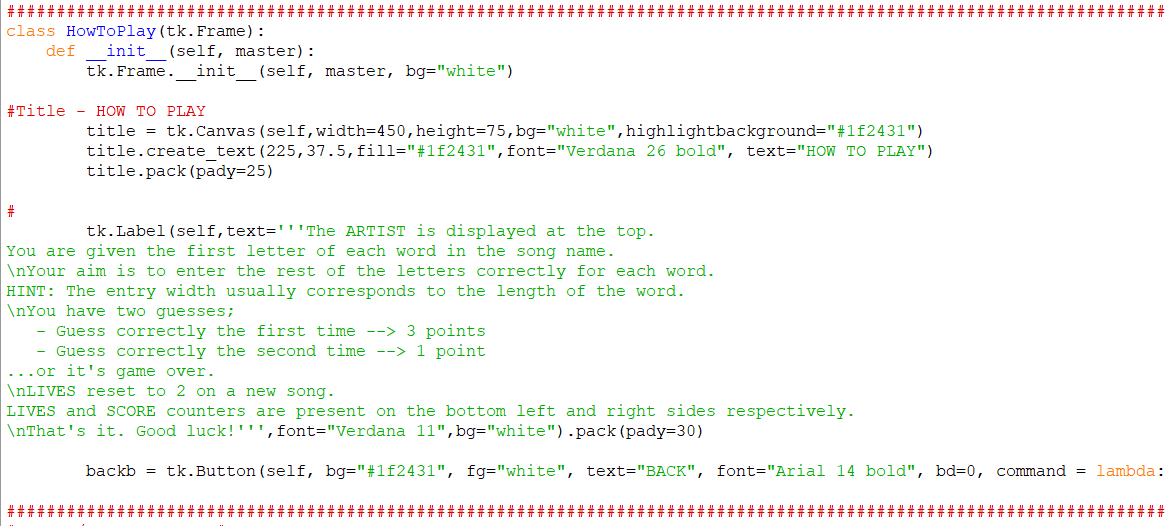
### MENU:

Once frame switching was initialised, it was simple to build a menu with buttons that would allow switching between frames. A function was used to reduce the need to repeat the same characteristics for different buttons.

****This was the result:

****

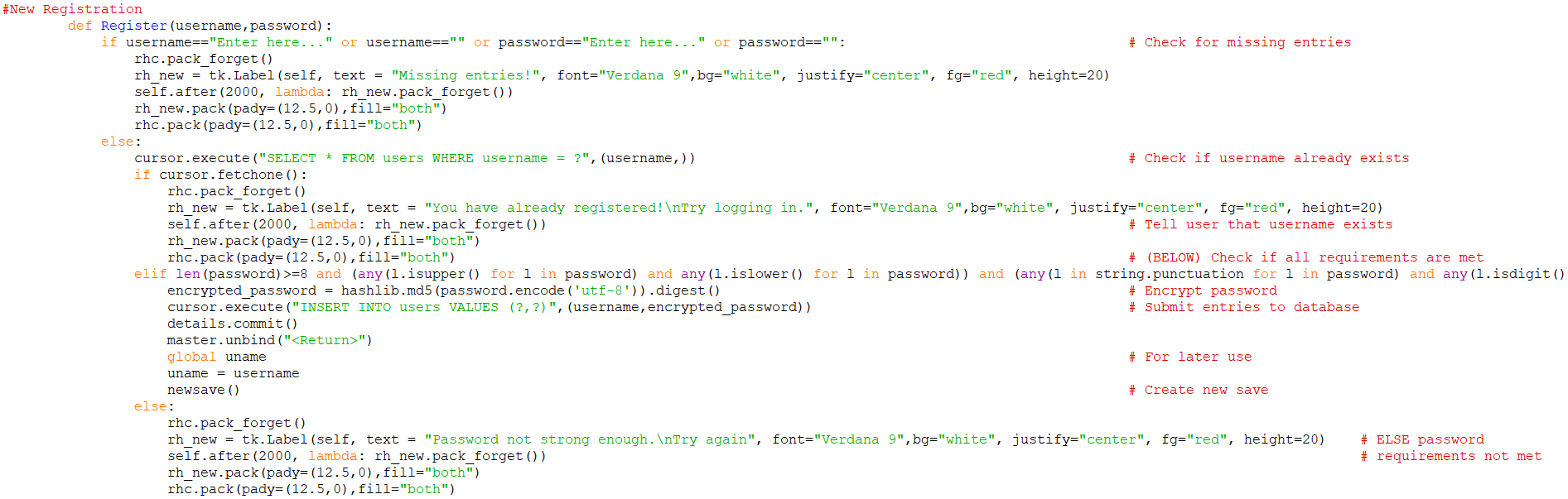
All buttons worked as expected. Frames were switched back and forth without a hitch.

The How to Play screen also worked as expected.

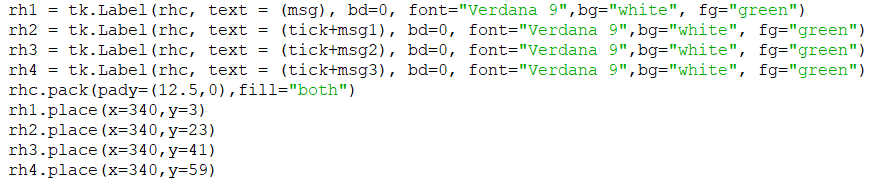
### LEADERBOARD:

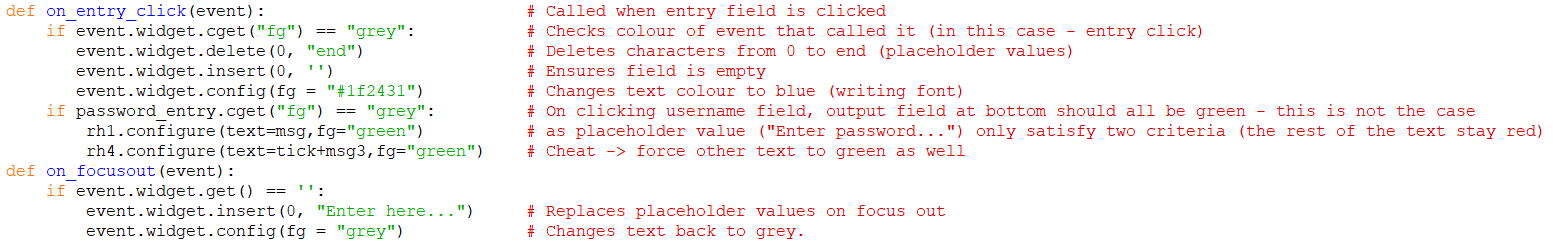
This frame required a lot of trial and error. Combining multiple list boxes at first did not work. It turned out that a separate canvas had to be created to hold both the list boxes and scrollbar. Each listbox was then linked to the same scrollbar.

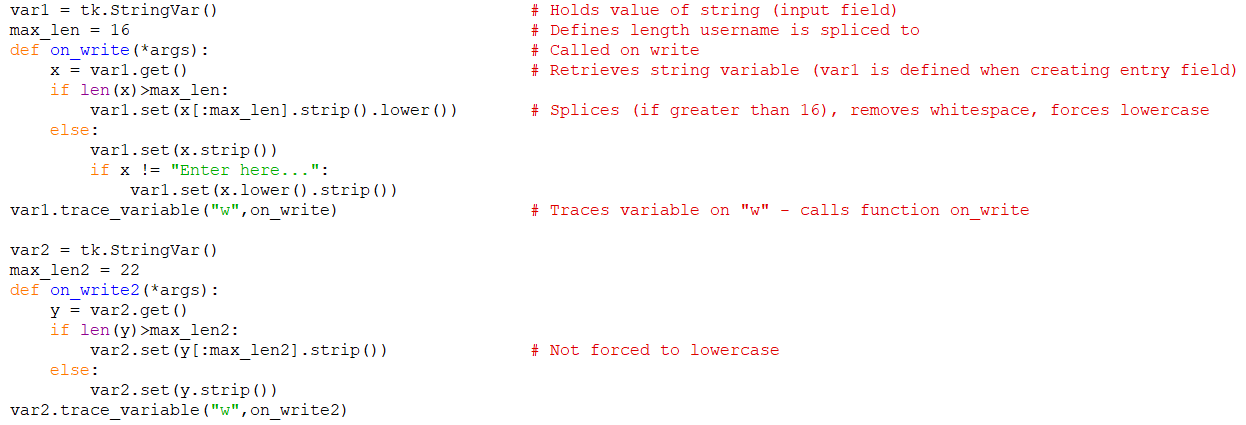
### USER AUTHENTICATION:

Login and registration functions were easily coded. Password was successfully encrypted and stored.

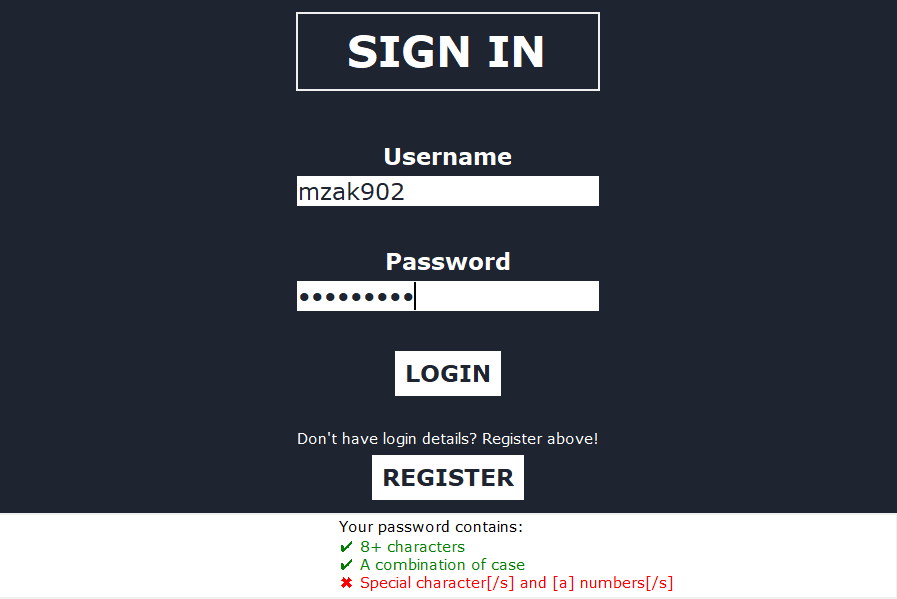


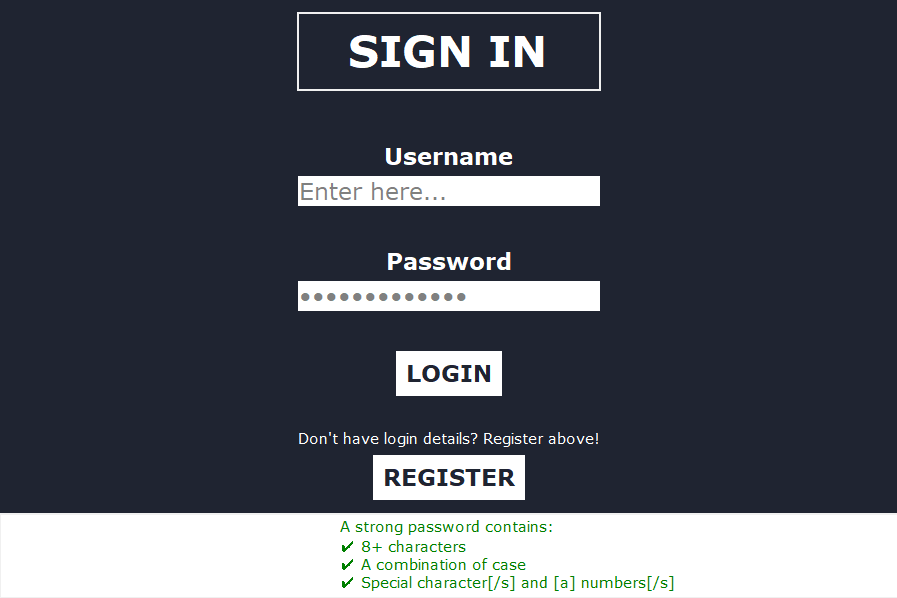
The real-time validation feature was slightly more arduous. Text would not change colour on write, instead showing errors (indicating that variable labels did not exist). At first, I thought it was due to the placement of specific lines (where labels were packed/placed). It was later that I realised that labels that needed to be ‘configured’ later had to be packed/placed separately and not in the same line as such:

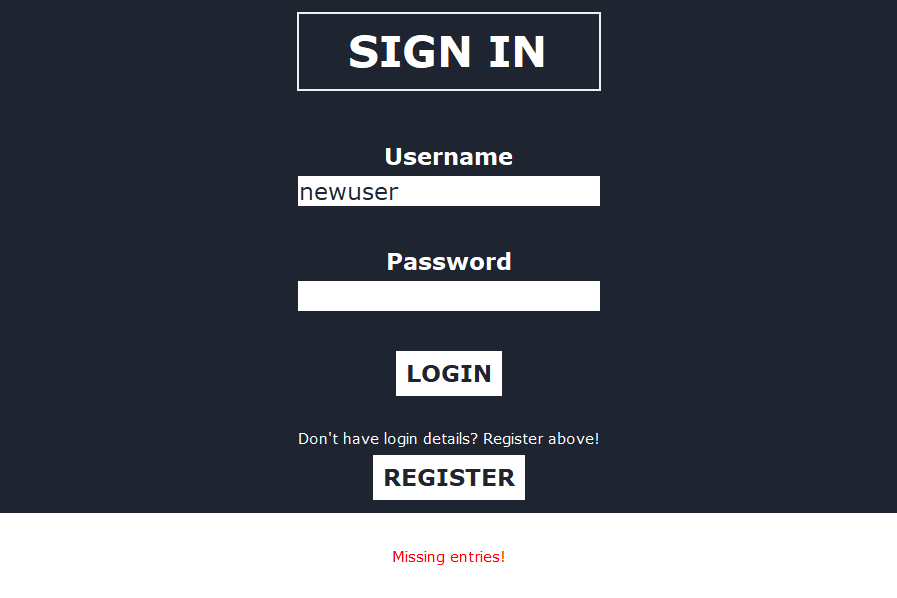
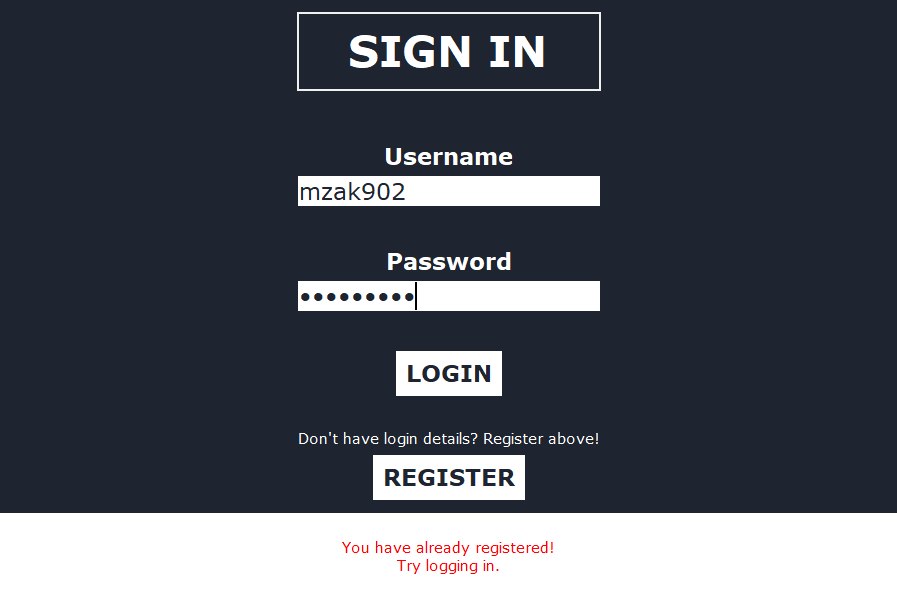
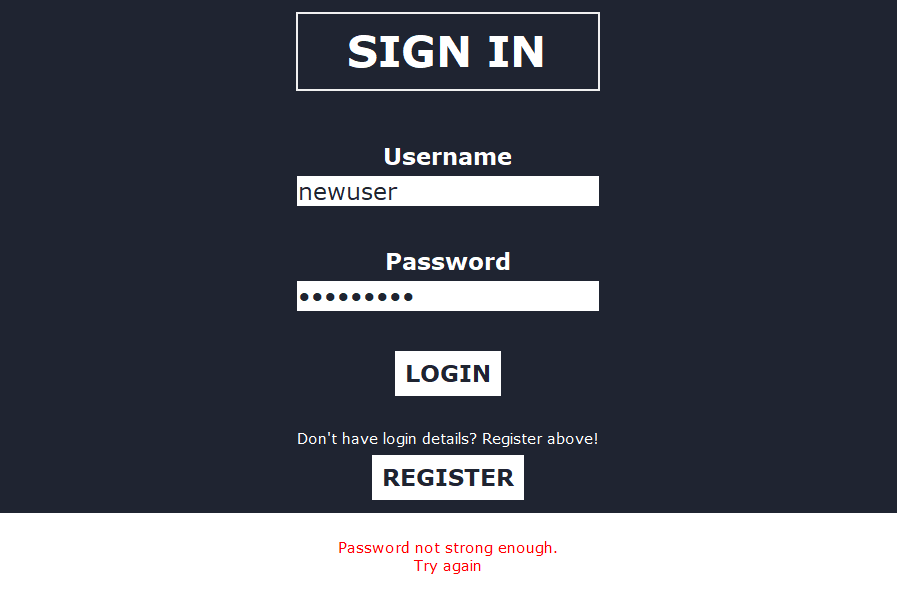
Later, it was decided to add focus effects on entry fields. This way a user could differentiate from their entry and placeholder values. A grey foreground colour was used on focus out, and dark blue on focus.

Input fields were sanitised to requirements on write. This meant less errors as incorrect input, especially on the username field, was not possible.

The code here slices username to a maximum length of 16 characters. It is not possible to go beyond this. It also removes whitespace and converts it to lowercase. There was a problem with pasted text allowing these whitespace/uppercase and so ‘Ctrl-V’ usage was disabled on both fields. The problem was solved. The password is limited to 20 characters.

This was the outcome:





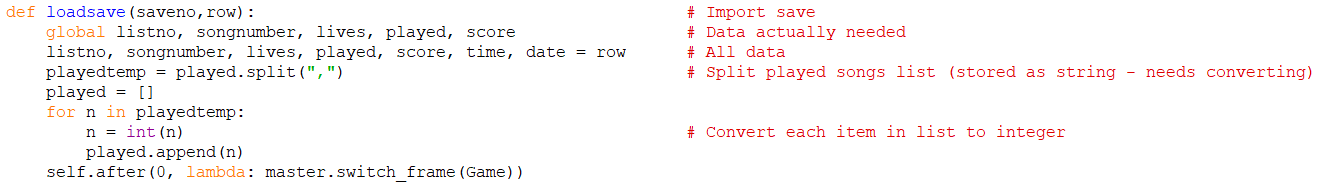
Inputs were tested rigorously for various borderline and (for the password field) erroneous inputs with successful results. Buttons were also repeatedly clicked on correct and incorrect inputs to check for errors. Everything went fine.

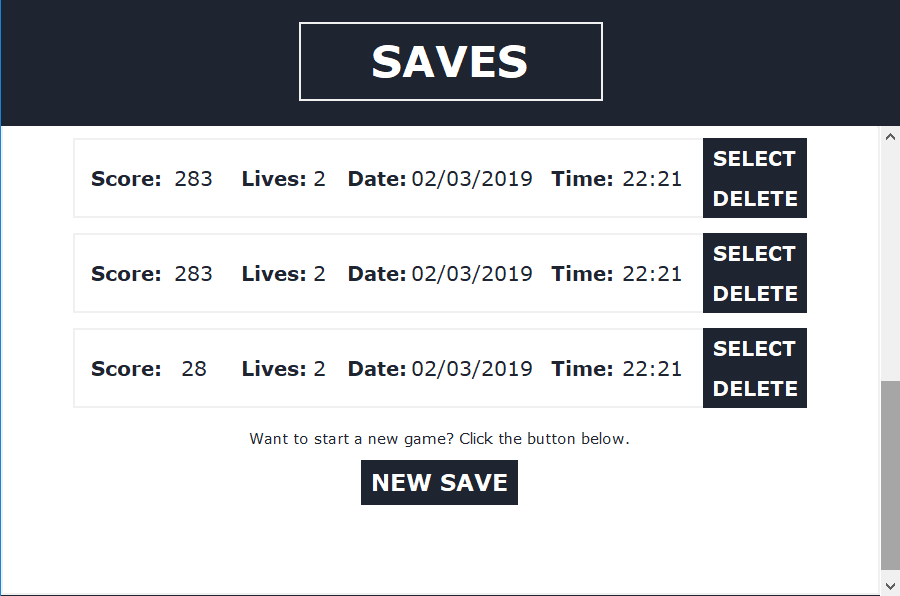
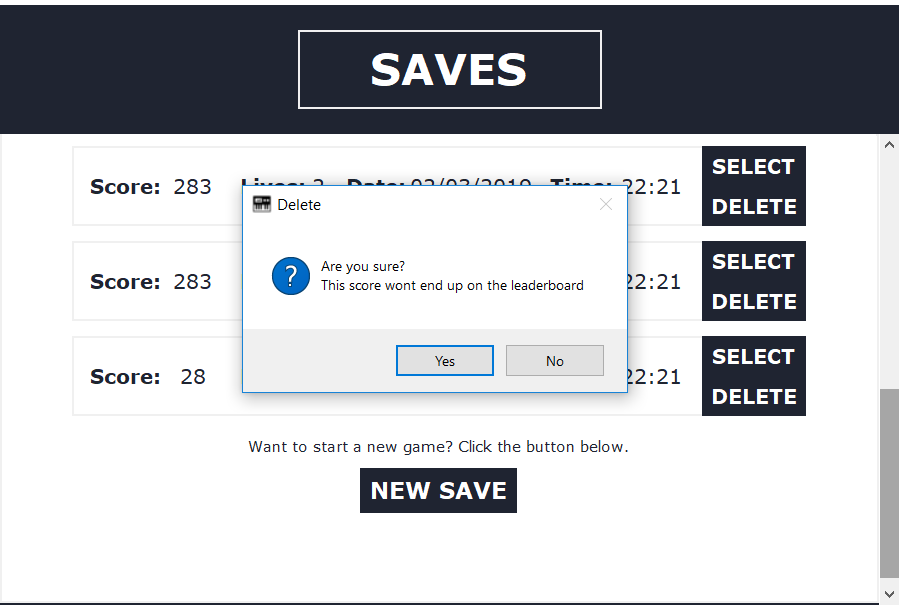
### LOAD GAME:

A list of saves had to be looped over to create a list of containers with buttons for each save game available. This presented a problem as data could not be fetched from a container (by clicking a button linked to it) created during a FOR loop – as variables are not saved and are overwritten on each iteration.

At first I tried to append each box to a list as an object. However data could not be retrieved from the resultant object as there was no ‘StringVar’ function.

After a few other attempts, the working solution required using a lambda function to store the required data in a variable (n) and passing it through as a parameter in the loadsave() function or delsave() function when the respective button was clicked.

Lists could not be stored in a database, this meant the played (playedsongs) list could not be stored as is. Instead it had to be converted to a string with commas. This was then converted back to a list of integers when imported.

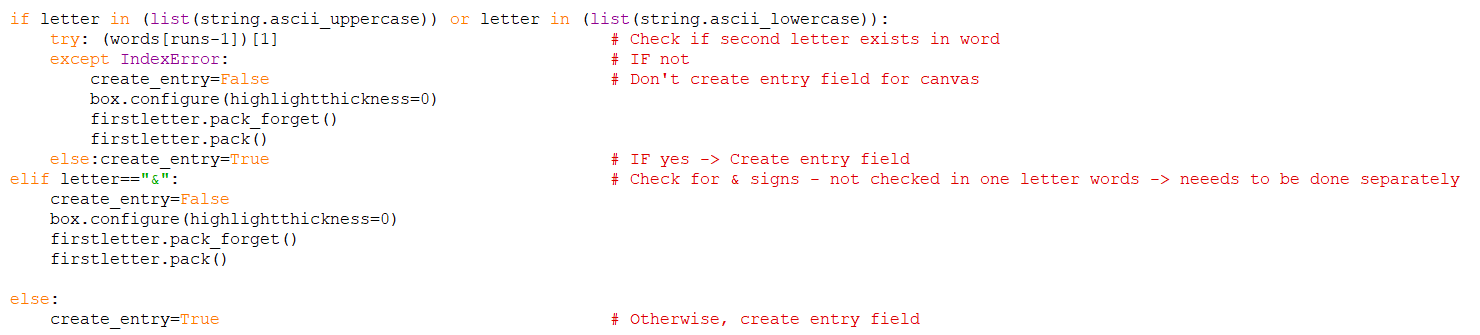


The final result:

Message boxes were used in the case of accidental deletion. Users are warned that progress will not affect leaderboard. Combinations of deleting and selecting saves was tested as well as spamming of buttons. No errors were found. Message box was also tested to ensure that answering ‘No’ would not affect anything. All saves were deleted to test if that works, after which the new save button was used.

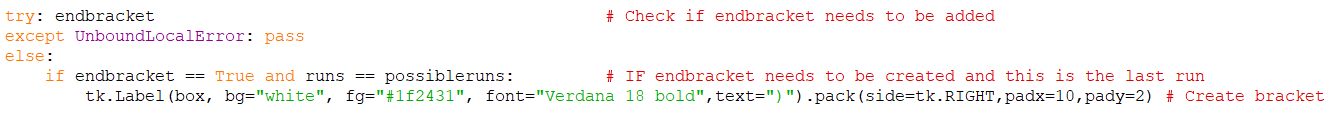
### GAME:

This frame required a lot of changes during testing.

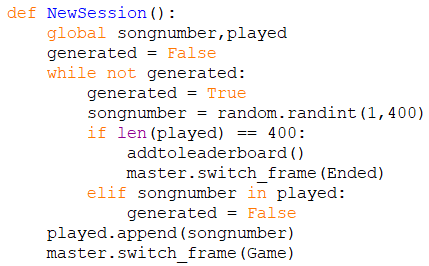
Certain songs had one letter words which required a check to ensure they do not have an input field attached to them. This in turn meant an empty item in the correctwords list which had to be removed.

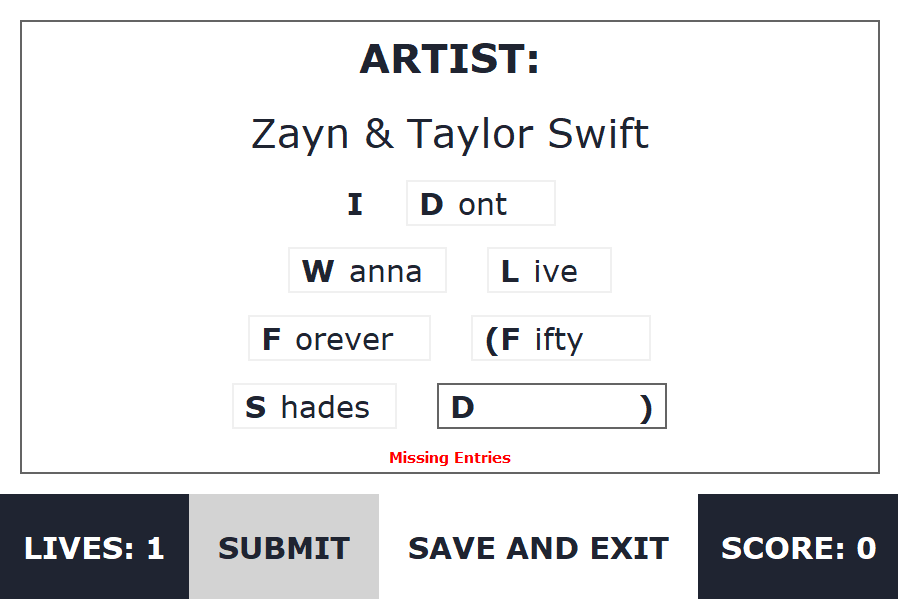
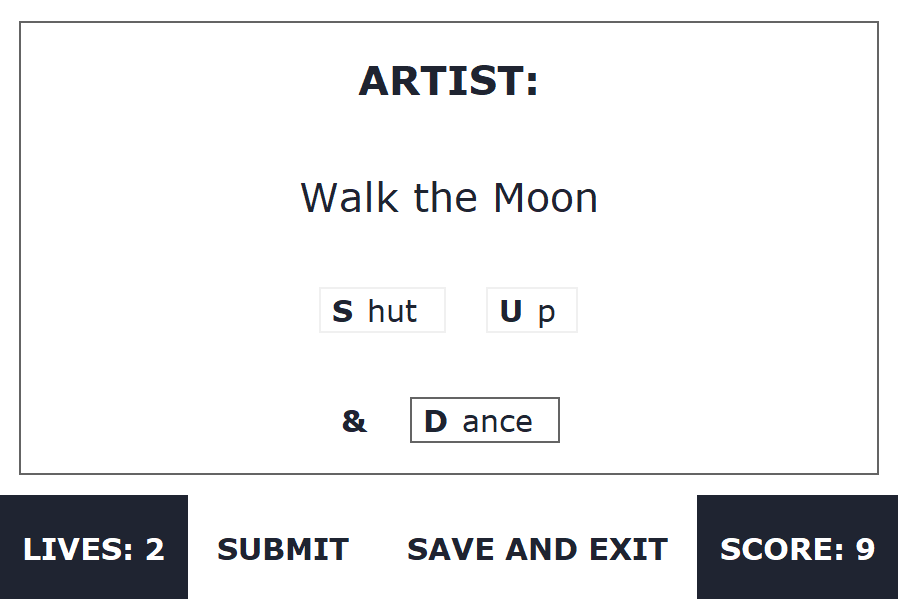
After this, it was also found that the ‘&’ sign affected answer checks. This was also corrected.

Later, it was decided to add brackets into the output so that user could differentiate between different known versions of song names.



The program can successfully generate a random integer that has previously not been played. There haven’t been any errors since the inception of this code.

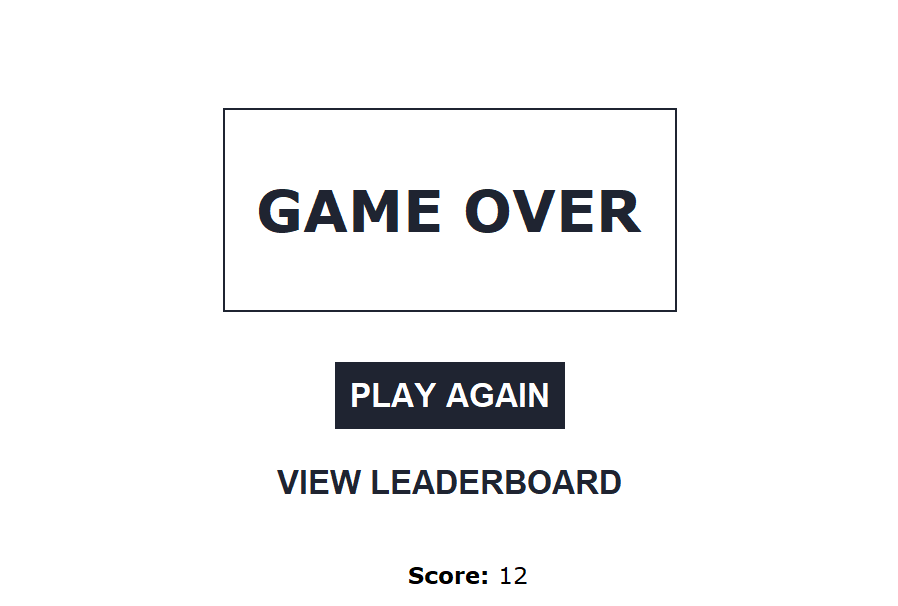


Results:

Game was played for a while with various inputs. Any errors that did happen were fixed. However, there is possibility of bugs as all 400 songs have not been tested with all possible combinations. The game follows the scoring system specified by the brief.

### GAME OVER:

The game successfully uses a Game Over frame to output the user’s score as well as options to return to beginning or view the leaderboard. Buttons have a hover effect to enrich its appearance.



Buttons work as expected. Score has been outputted correctly throughout testing phase.

Testing screenshots in the development section were taken after each subsection had been fully developed. This means that present results are the same. Testing results above encompass everything including final/terminal tests.

### FULL CODE LISTING:

A copy of the full code can be found here:

<https://raw.githubusercontent.com/MZakariyya9/MusicQuiz/master/MusicQuiz.py>

### REFERENCES

<http://effbot.org/tkinterbook/>

Tutorials on this site were often used to learn about various possibilities with tkinter.

N:\...\How to write up the Analysis section.docx

N:\...\How to write up the Design section.docx

N:\...\How to write up the Development section.docx

N:\...\How to write up the Testing, evaluation and conclusions section.docx

CraignDave’s NEA report advice was used to help write this report.

<https://chart2000.com/about.htm>

- chart2000-song-2010-decade-0-3-0046.csv

- chart2000-song-2000-decade-0-3-0046.csv

The above files from the aforementioned site were formatted to create the list of songs used during gameplay.

Evaluation

I think my program was successful in achieving the requirements and success criteria that had been set out by myself as well as the brief. Use of a graphical user interface was well implemented. Many extra features that were not thought of at first were added during development. Overall, the game seems interesting.

I was restricted to the use of SQLite to allow compatibility with the Python program available. This means that the database is easily accessible to hacking attempts and for cheating. It may also mean the database can easily be corrupted, causing unexpected errors. If access was given to allow module installation, a MySQL database (stored on a server) could have been used thus making database access harder. Connections to online databases would also allow multiple players on different devices to compete on leaderboards.

Through the development of this program, I have been able to improve my knowledge on Python, especially the tkinter module. I have also been able to consolidate my understanding of basic programming concepts. It has also given me the confidence and ability to try out slightly more complicated projects.