Computer science coursework project

Main Program

Table of Contents

[Analysis 3](#_Toc45549284)

[Description 3](#_Toc45549285)

[Computational methods used 4](#_Toc45549286)

[Problem recognition 4](#_Toc45549287)

[Problem decomposition 4](#_Toc45549288)

[Abstraction 5](#_Toc45549289)

[Stakeholders 5](#_Toc45549290)

[Interview 6](#_Toc45549291)

[Questions 6](#_Toc45549292)

[David 7](#_Toc45549293)

[Erin 7](#_Toc45549294)

[Max 8](#_Toc45549295)

[Existing solutions 9](#_Toc45549296)

[Chordify 9](#_Toc45549297)

[MuseScore 9](#_Toc45549298)

[Guitar Hero World Tour 10](#_Toc45549299)

[Stakeholder Input 11](#_Toc45549300)

[**Question 1:** How would you prefer sheet music to be displayed to you during a performance? 11](#_Toc45549301)

[**Question 2:** Would you rather have a microphone input that may be less accurate in some scenarios, or a MIDI/similar input that would be accurate but restricts what instruments can be used with the program? 12](#_Toc45549302)

[Program Features 12](#_Toc45549303)

[Settings Menu 12](#_Toc45549304)

[User Account 12](#_Toc45549305)

[Song Selection 12](#_Toc45549306)

[Performance 12](#_Toc45549307)

[After performance Analysis 13](#_Toc45549308)

[Limitations 13](#_Toc45549309)

[Software and Hardware Requirements 14](#_Toc45549310)

[Hardware 14](#_Toc45549311)

[Software 14](#_Toc45549312)

[Success Criteria 14](#_Toc45549313)

[Design 15](#_Toc45549314)

[User interface design 15](#_Toc45549315)

[Main Menu 15](#_Toc45549316)

[Options Menu 16](#_Toc45549317)

[Login Menu 17](#_Toc45549318)

[Song Select Menu 17](#_Toc45549319)

[Song Screen 18](#_Toc45549320)

[Algorithms 19](#_Toc45549321)

[Main Menu pseudocode 19](#_Toc45549322)

[Song Select Screen 20](#_Toc45549323)

[Options Screen 20](#_Toc45549324)

[Login Screen 21](#_Toc45549325)

[Inputs, processes, outputs and storage 22](#_Toc45549326)

[Development 23](#_Toc45549327)

[Analysing a microphone input as to predict the musical note played 23](#_Toc45549328)

[Coding initialisation of SoundData object 23](#_Toc45549329)

[Testing initialisation of SoundData object 24](#_Toc45549330)

[Coding reading stream data 24](#_Toc45549331)

[Coding the framing function 25](#_Toc45549332)

[Coding the window function 27](#_Toc45549333)

[Testing the framing and window functions 29](#_Toc45549334)

[Coding finding the dominant frequency of a frame 29](#_Toc45549335)

[Coding finding the dominant frequencies of a group of frames 31](#_Toc45549336)

[Testing finding the dominant frequencies of a stream 32](#_Toc45549337)

[Coding conversion of frequency to note name 33](#_Toc45549338)

[Testing the conversion of frequency to note name 36](#_Toc45549339)

[Prototype of note detection 38](#_Toc45549340)

[Creating the application window and main menu 39](#_Toc45549341)

[Coding the Application class 39](#_Toc45549342)

[Testing initialisation of the Application class 40](#_Toc45549343)

[Coding the main menu screen function 41](#_Toc45549344)

[Testing the main menu window 42](#_Toc45549345)

[Coding buttons 43](#_Toc45549346)

[Testing the buttons 45](#_Toc45549347)

[Stakeholder feedback 48](#_Toc45549348)

[Creating the song select screen 49](#_Toc45549349)

[Coding the scroll bar 49](#_Toc45549350)

[Testing the scroll bar 51](#_Toc45549351)

[Coding the song tabs 54](#_Toc45549352)

[Coding the load files function 56](#_Toc45549353)

[Loading song files 58](#_Toc45549354)

[Coding the main loop 59](#_Toc45549355)

[Testing the song selection screen 62](#_Toc45549356)

[Creating the performance screen 62](#_Toc45549357)

[Coding the song parser 63](#_Toc45549358)

[Testing the song parser 66](#_Toc45549359)

[Coding variable declaration 68](#_Toc45549360)

[Coding background rendering 69](#_Toc45549361)

[Coding the entering and countdown of the performance screen 71](#_Toc45549362)

[Testing the starting of the performance screen 73](#_Toc45549363)

[Evaluation 75](#_Toc45549364)

[Usability Features 75](#_Toc45549365)

[Questionnaire 75](#_Toc45549366)

[Success Criteria 76](#_Toc45549367)

[Evidence 76](#_Toc45549368)

[A clean and intuitive, consistent design. 77](#_Toc45549369)

[Main window with start menu UI including buttons to navigate to other menus. 77](#_Toc45549370)

[Main window with song selects UI 78](#_Toc45549371)

[Notes of the song shown clearly and accurate to music theory, large and unambiguous. 78](#_Toc45549372)

[Limitations and Further Additions 79](#_Toc45549373)

[Maintenance 79](#_Toc45549374)

[Full Program 79](#_Toc45549375)

[Tree 80](#_Toc45549376)

[Dependencies 81](#_Toc45549377)

[main.pyw 81](#_Toc45549378)

[audio.py 96](#_Toc45549379)

[song.py 100](#_Toc45549380)

[ui\_elements.py 102](#_Toc45549381)

[user.py 107](#_Toc45549382)

# Analysis

## Description

The technology for real time music performance analysis is not easily and readily available to a wide audience. This type of technology is extremely useful for educational purposes, however as of now the best option is to find a tutor to teach the instrument. This has a large financial cost, as well as potentially having to travel to a dedicated school to learn and only having certain times for lessons, which is very restricting to a weekly schedule and in learning to play.

My program will aim to provide a solution to this: it will be designed around the idea of a tutoring program. The program will take an audio input from a microphone and tell the user what note(s) they played and judge the quality of a performance. It will be able to handle multiple notes at once (chords).

It will display the notes to a song on screen and give a user a score after they have played along based on how accurate they played the notes and whether they played the notes to the beat of the song.

It will use graphics on a computer capable of audio data processing for displaying a user interface, to show the user what to play and analytics of their performance, as well as processing tools for analysing the sound input from a microphone and will need hardware capable of real-time audio analysis. It will have progress tracking using an account database as well as an AI to compare your results with the rest of the database to give accurate ratings of your performance.

This will enable users to practice an instrument with guidance whenever suits them, for as long as they like, and with the full attention of the virtual tutor, potentially making learning an instrument much more efficient and faster paced.

## Computational methods used

### Problem recognition

The primary challenge will be to analyse the data given from the microphone input. A solution will need to be found that is not only accurate but takes minimal computational time as to reduce lag during the performance. The frequency of the microphone is needed to determine which note the user is playing, so the underlying problem will be to take the digital raw input of the microphone and convert it to a usable format as to be able to perform calculations on it in order to find the frequency.

The problem is suited for a computer as most musicians currently have no way of getting judgement on their performances from an unbiased and reproducible way. A computer would not “hear” the music differently in different scenarios, whereas a human music tutor may be influenced by outside factors such as the music they like, tiredness that day, a distaste for certain music techniques or recency bias in music they’ve heard recently. Furthermore, if you played a piece of music to two different human tutors you may get one positive response and one negative response. However, playing a piece of music the same way twice with a computer-based tutor would yield the same outcome every time.

### Problem decomposition

#### Program components

**Menus** - there will be five menus: the main menu, options menu, log in menu, song select menu and post-performance review menu.

**Game** - the game will consist of these basic steps:

1. The chosen song will be read from a file and displayed onto the screen in a scrolling fashion.
2. Using a microphone to provide an input to the program as musical notes are displayed.
3. The sound will be processed to find the dominant frequency and using that to conclude what note was played versus what note was expected to have been played.
4. The user will receive feedback on their performance both in real-time as they are playing as well as at the end of the performance as a review.

**Profiles** – used to store the user’s log in information and keep track of personal bests in a dedicated user file.

### Abstraction

My program will abstract several features to make it simpler and more accessible to a wider audience.

|  |  |
| --- | --- |
| Abstraction | Reason |
| The program will not consider whether an instrument is tuned or not. | It does not matter if an instrument is tuned differently from the standard tuning, as long as each different note is tuned by the same amount the program would not need to tell if it is not the standard tuning (e.g. the note A does not need to be 220Hz exactly as long as the note B is A + 26Hz). |
| The program will not provide any reduction of background noise from the microphone input. | As instruments are very loud, they usually dominate the microphone soundscape anyway, so background noise is unlikely to affect the result. |
| The program will not dynamically track user progress nor restrict songs based on difficulty; they will have to choose difficulty for themselves. | Users may prefer to play easier songs at sometimes and harder ones at others or may just prefer to play songs they like. |
| Will not show how far off a user was in the on the fly feedback. | Instead a simple rating out of 10 will suffice to let the user know tangibly how well they played an individual note |
| Multiple notes played at the same time (chords) will not be included in the program. | The work required to do this is a lot more difficult than a single frequency and thus due to time constraints would not be feasible to do in the given time frame. |
| Some musical notation will not be supported by the program. | Very short notes and niche notation will not be supported as they are either rarely used and thus not necessary for the majority of songs or so short in duration that it may make the program inaccurately track the user’s performance when playing them. |

## Stakeholders

The main stakeholders for my project will be music students under the age of thirty. Music students will be actively looking to improve their skills and so would benefit from the program. The program will be available to musicians of all musical abilities and skill ranges; beginner to advanced/professional.

That said, another large market for the program would be casual musicians – people looking to improve their musical abilities, but who do so in a more relaxed and less intensive way. This program would be ideal for them as they do not have to commit to weekly tutor sessions like they would with a human tutor. Not only that, but for both the casual market and music students, having a free alternative for music tutoring is very beneficial, as casual musicians would not be willing to invest large sums of money that would be required with a human tutor, and music students generally do not have a lot of money *to* be spending.

A younger audience will be more receptive and willing to use a computer program over a traditional tutor, due to them being more likely to use technology in general over older people. Not only this but people tend to pick up an instrument at an earlier age, often in childhood, so many beginners will be looking for a cheap alternative to a typically expensive traditional tutor.

#### David

An advanced saxophonist, David plays his instrument to a high degree and has a diploma. He will be able to identify whether the program is able to challenge (and therefor help improve) a skilled and experienced user. He will require high difficulty songs and will use the results scoring to get an accurate impression of where he needs to improve. He will test the utility of the program – how helpful it is for advanced musicians, whilst not being too bothered for aesthetics or ease-of-use.

#### Erin

She has only played piano for a very short amount of time; she will be useful to test the lower bound of the difficulty of the program and making sure the program can help a user starting off on an instrument. She will need very easy tracks to play along to and will not make use of technical reviewing; preferring simple scoring of pitch and rhythm. Being young, she will need a UI that is simple to use and be intuitive to move around the program.

#### Max

Max is an intermediate guitar player, who plays in his free time for fun. He is very casual about his improvement and will be looking to enjoy using the program over just raw improvement. He will be focusing on real time accuracy analysis to make sure the music he is playing sounds as nice as possible. He will be testing how enjoyable the program is rather than just it’s utility, so ease-of-use, design and enjoyability will be measured with him.

## Interview

I will be trying to identify features of the software that the stakeholders would like and use, or dislike and never touch, as well as overall preferences for the aesthetic look of the program and accessibility and ease of use. During the interview follow up questions may be asked, and I anticipate further questions being formulated later in production to be asked to the stakeholders.

### Questions

#### **Question 1:** Do you currently use any computer software to aid your instrument practice? If so what do you use?

This will find the scope of which software is currently used to aid instrument practice for the individual.

#### **Question 2:** What is more important to you; correct rhythm or correct pitch?

This will help determine the weight of rhythm compared to pitch in the scoring system, if one is found to be more important than the other then it should count more in the final score.

#### **Question 3:** Could you trust a computer program to give an accurate and relevant breakdown of a performance?

This question will give insight as to what criteria is needed to make users trust the output of the program, as if they do not trust the software, it is not successful at what it is supposed to do.

#### **Question 4:** What pushes your musical skills most when practicing?

Again, this will help define what the scoring system will need to consider as easy skills should not be equivalent to difficult ones when rating the performance.

#### **Question 5:** What would you miss most if you were to switch from a human tutor to a computer tutor?

This will give insight as to what the program will need to achieve in order for the transition from human to computer to be as seamless as possible.

### David

#### **Question 1:** Do you currently use any computer software to aid your instrument practice? If so what do you use?

“I use MuseScore to find sheet music, but no software to help me improve. So far I have only ever had a real tutor for improvement.”

#### **Question 2:** What is more important to you; correct rhythm or correct pitch?

“Correct rhythm as the pitch can take some artistic liberty.”

#### **Question 3:** Could you trust a computer program to give an accurate and relevant breakdown of a performance?

“It depends on how it was done; I imagine a computer could pick up on all the errors fairly accurately.”

#### **Question 4:** What pushes your musical skills most when practicing?

“Complex rhythms and difficult key signatures. Anything that is too complex to be easily sight-read pushes my abilities and really allows for progression.”

#### **Question 5:** What would you miss most if you were to switch from a human tutor to a computer tutor?

“Probably the chance to be shown how to play something rather than just reading the music.”

These answers show that rhythm is more important than the actual notes, which could be reflected in the score a user may receive at the end of a song. Another idea could be to have the program able to play a rhythm itself, to show a user what a tricky rhythm may sound like. Having some more difficult songs that focus on complex rhythms and difficult key signatures would also be beneficial in creating a range of musical skills to be worked on during use of the program.

### Erin

#### **Question 1:** Do you use a computer to help with your practice?

“I only use books when practicing because they are bigger and easier to read.”

#### **Question 2:** What is more important to you; correct rhythm or correct pitch?

“I think the notes sound good when played correctly but the song as a whole only sounds nice when the rhythm is also correct.”

#### **Question 3:** Do you think a computer could tell you how good you played well enough?

“Yes, computers are very good at that sort of thing. Would the computer be as wise as a real-life tutor though?”

#### **Question 4:** What pushes your musical skills most when practicing?

“I find it hard to make large jumps between notes, especially in fast songs.”

#### **Question 5:** What would you miss most if you were to switch from a human tutor to a computer tutor?

“I think I’d still quite like to be shown how to do a tricky part of the song rather than just be told a number.”

Erin’s responses show that it will be necessary to make the stave and notes large rather than small during the performance, so it is easy to read from a monitor. She also raises an interesting question about how “wise” the program would be, compared to a human tutor with perhaps decades of experience in music theory. This limitation of the computer (being “unwise”) would be overcome by making its role less of how to improve in the future, and more so how well the user is playing *currently*. This is not a drawback of the program, but rather a misrepresentation of its role as a music tutor.

### Max

#### **Question 1:** Do you currently use any computer software to aid your instrument practice? If so what do you use?

“I use a computer to play backing tracks whilst I am practicing but no software that directly impacts my learning.”

#### **Question 2:** What is more important to you; correct rhythm or correct pitch?

“I think although you can’t have one without the other in a good performance, rhythm is definitely the trickier one to manage and can really make or break a piece of music, especially when performing with others.”

#### **Question 3:** Could you trust a computer program to give an accurate and relevant breakdown of a performance?

“I’d imagine a computer could definitely analyse a performance much more effectively as it lacks any bias or judgement, however its lack of real-life experience and knowledge could potentially be an issue.”

#### **Question 4:** What pushes your musical skills most when practicing?

“Learning fast paced songs or part of songs that are very quick.”

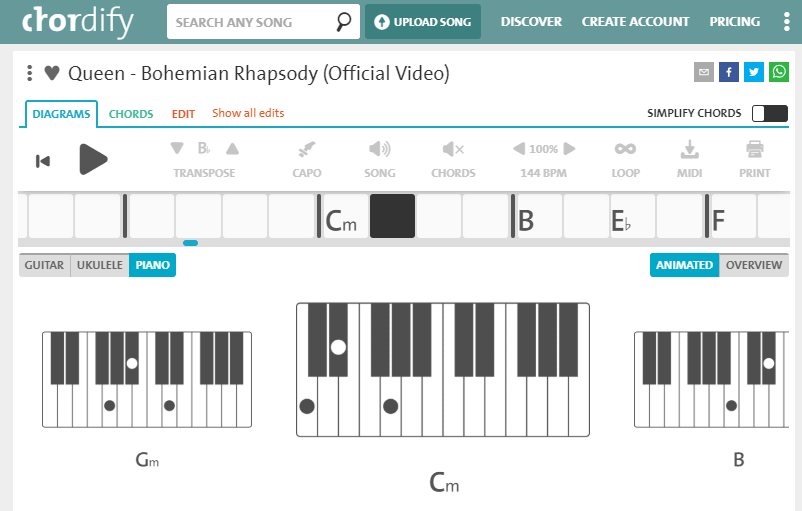
#### **Question 5:** What would you miss most if you were to switch from a human tutor to a computer tutor?

“The experience said tutor would have from their own music career would be hard to replace.”

Max’s input highlights the difficulty in learning fast paced songs as well as supporting the unanimous opinion that rhythm is more important than pitch. The program will reflect this by taking careful consideration as to when a note is available to be played, with longer duration notes having a larger window of when they should be played as to reflect their own duration.

## Existing solutions

### Chordify



Chordify is a free online tool that displays chords for songs to be played in real time. It works by taking audio from popular songs, turning the waveform into a spectrogram, then finding chords and beats from this visual representation using deep neural networks. This is a great way to do audio processing, however Chordify does this beforehand and has no interaction with the user in real time. Its simple aesthetic is very user friendly whilst not sacrificing functionality (such as tempo control and transposition). This is something I will aim to achieve with the graphics of my own program.

Its intuitive design makes it easy to follow along with the song; the chords scroll across the screen at the correct speed (tempo), so the user can follow along. To support less experienced musicians, the tempo can be reduced, and the song can be transposed to make it have easier notes to play.

The website may be slightly too limited, however, and is far too simplified for advanced users. It is useful for playing along with a song, but melodies aren’t provided, just chords.

#### Features of interest

The idea of the song scrolling across the screen as it supposed to be played is useful as it allows for the current part of the song to be displayed clearly and take up most of the screen space, whilst still not requiring any user input to move the song along during the performance – it is all done automatically. This will be something that I make use of in my own program.

### MuseScore



MuseScore is a free song writing tool that excels in providing a simple, clean and intuitive working environment for composers and performers, unlike other similar programs that may be difficult to learn for new users. Users can search for music and find sheet music for almost any instrument or make their own score of a song in the software, which they can then upload for other users to find.

Its simple look is aesthetically pleasing and a design my program shall take inspiration from, as it is very accessible. The songs can be played from within the program at the correct tempo, meaning musicians can play along with it.

However, it does not offer any feedback on performance, rather just being a tool to show sheet music.

#### Features of interest

The key features of MuseScore are the simplicity and design of the program. This is something I will aim to achieve in my own program as to make it as accessible and easy to use as possible. I will also be taking reference as to how the sheet music is displayed in MuseScore and implement it into my own program as to make it as accurate as possible.

### Guitar Hero World Tour



Guitar Hero World tour is a popular video game for the Nintendo Wii. You purchase their specialised equipment, such as the Guitar Hero guitar and the Guitar Hero drum set, and can play popular rock tunes, being scored for accuracy and number of consecutive correct notes played, then ranked against other players. One of the game-modes is a singing one, where you sing along with the lyrics of the song, and can visually see your pitch on screen, along with where the correct pitch lays.

This feature is a very interesting solution to showing how far away a player is from the correct pitch and showing the difference in terms of the y-axis of the screen is a good way to let the player know whether they were sharp or flat (or higher/lower) than the correct note. The program also takes into consideration note length, so if a player sings for too long or doesn’t sing at the right time, they will be penalised. The length of notes is easy to see, making it ideal for beginners.

However, the program does not provide any real music theory – it is designed to be as globally inclusive as possible; having sheet music is not something most people would be able to interpret. This severely limits the learning potential of the program, which is fine for its use as a game but would not work well for a tutoring program.

#### Features of interest

From this program I will be adapting the idea of using the microphone as the input rather than alternative methods, such as a MIDI input. With a MIDI input, an electric instrument would be plugged into the computer and a translator algorithm would pass the note on for the program to make use of. While this would give the program a 100% accuracy in terms of knowing which note is played, it severely limits the range of instruments that would be able to be used, as only electric instruments would be able to be plugged in to the computer.

## Stakeholder Input

After conducting some research into existing solutions, I have decided to ask the stakeholders some further questions to gain a deeper understanding into what they would like to see in a music tutor program.

### **Question 1:** How would you prefer sheet music to be displayed to you during a performance?

#### David

“I would like to have as little interaction with the program as possible during the performance as I would have both my hands on my saxophone. In fact, if the program could automatically display the music as it was needed that would be beneficial over traditional sheet music as even that requires page turns.”

#### Erin

“I think the music should be big so it can be read easily as the computer screen is already smaller than a book and so would not fit the whole song onto it very well.”

#### Max

“In my limited experience in reading sheet music off a screen, I have always found myself zooming in and forever scrolling about the page as if I want many notes on the screen then they are too small to read but if I zoom in and make the notes readable there are very few of them on the screen and so I end up continuously scrolling which is not very helpful. A solution that overcomes both these obstacles would be very nice and definitely enable me to use the program over alternate methods.”

### **Question 2:** Would you rather have a microphone input that may be less accurate in some scenarios, or a MIDI/similar input that would be accurate but restricts what instruments can be used with the program?

#### David

“As a saxophonist it would make sense that I’d rather have a microphone input as otherwise I couldn’t use the program. There are electronic versions of the saxophone, such as a EWI [Electronic Wind Instrument], however it is incredibly different to play than a sax and I would consider it a different instrument altogether, so that would be no use anyway.”

#### Erin

“I’m not sure I’d know how to setup an instrument so it works with the computer, so definitely a microphone input as it would be easier to use. I’d need to buy a microphone though.”

#### Max

“If the microphone was accurate enough in relatively quiet environments then that would suit a lot more people I think. It wouldn’t affect me as much to have a MIDI input as I have an electric guitar with all the wires and adapters needed to plug it in to a computer, but for most people I appreciate that is a far more difficult solution.”

From the research and stakeholder questions it is very apparent to me that the most important features of the program would be making the music very clear and easy to read, which involves making it take up as much room as possible on-screen whilst showing as much information at once as possible without direct user input. I am confident in my choice to use the microphone as an input rather than MIDI or otherwise as using a microphone drastically increases the general accessibility of the program.

## Program Features

### Settings Menu

The user will be able to go to the sign in menu in order to sign into their account, as well as create an account, and once signed in they will have the option to sign out or delete their existing account.

### User Account

Users will have the option to sign into an account, which will keep track of the scores they received on each song, so users can try and beat their score later. User account details will include a username and password as well as personal best scores for each song. This will be stored in a separate file as a JSON format.

### Song Selection

This will provide a list of available songs, for users to select which they would like to play. Each song will have a tab with the name of the song, difficulty, and user personal best if they are signed into their account.

### Performance

The performance of a selected song will be the main portion of the program. Once a song has been selected, the user will be taken to a new screen where the song will be displayed in a scrolling fashion, start to finish. Each piece of notation making up the song will traverse the screen from right to left one by one, and upon passing a designated section on the screen, the user is expected to play that sound, whether it is a note or a rest. If they have played correctly, their song score will increase.

To determine whether a note has been played correctly, a microphone input will be taken for the program to make use of. Analysis will be performed on the audio input data to determine the dominant frequency of the input over an arbitrary amount of time. Once a “likeliest” frequency has been obtained, it can then be converted to a note value using a list of notes and their associated frequency values.

### After performance Analysis

This will combine the data collected during the song performance, including the accuracy of rhythm, pitch and dynamics and create a final score as well as a breakdown of the performance and parts of the song that were performed especially poorly or strongly. It will show a graph of whether each note was played correctly so the user can see exactly where they went wrong.

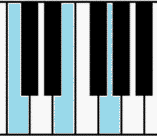
### Limitations

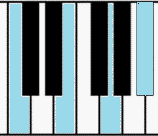
I anticipate I will be able to program the microphone input to usable data algorithm, a user database, a working song performance concept and analysis of said performance against other users in the database, as well as multiple songs to choose from. If I had more time, I would add extra songs and make a full set of custom graphics (as currently I will use basic graphics to save time).

Something that may restrict the program is background noise – whilst instruments tend to dominate the sound space of a microphone, background noise could interfere with what note is interpreted if the microphone is far away from the instrument or the background noise is particularly loud (such as a nearby passing train).

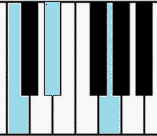
Furthermore, a microphone is required to use the program – it will not work with other input methods such as MIDI. If I had more time, I would add more ways to input instruments into the program for users who would want that.

Chords will also not feature in this program. Whilst there are lots of instruments that do not play chords, namely brass and woodwind instruments, piano, guitar and other similar instruments do. I will not be including chords due to the dramatically reduced accuracy when looking for the fundamental frequencies of multiple notes at once. Looking for chords would be fairly inaccurate and discerning similar but different chords, such as a C to a C7 chord:

**C (major)**

**C7**

Or even a C major to a C minor chord:



The difference would be a matter of 37Hz, which is very small considering the accuracy reduction, however the difference between a major and minor chord is very large. By including inaccurate chord detection in which a user could play a major chord when they should have played a minor chord and still get it “correct”, the utility of the program’s tutoring capabilities would be lowered. For that reason, they will not be included. If I had more time to work on the program, I would look into methods of more effectively and reliably finding the frequencies of multiple notes played at the same time.

## Software and Hardware Requirements

### Hardware

**A computer capable of running the software** – the analysis will require a large amount of processing power so if a computer is not able to process the data quick enough the program may not work as intended. The computer will also need a large enough hard drive to store the program and user account data (although this won’t be a large amount of data). Most computers and even laptops will be fast enough to run the program.

This program also requires **a microphone**, as the entire program pivots around the user playing an instrument, which is an audio input to the program. The computer running the program will therefor need an I/O interface to plug the microphone in to.

### Software

**Windows** is the operating system of choice as it is the most common and supports Python. Linux and MacOS may also be supported, however no testing of the program on those operating systems will be done.

I will be using **Python version 3.6** to run the program on. Therefor a Python 3.6 interpreter will be required.

The **PyGame** library for Python primarily to display the program to the user.

I will also be using third party dependencies including: **PyAudio** to take an audio input through Python, **SciPy** to assist in the audio input formatting and **NumPy** to perform complex maths and analyse the audio input.

## Success Criteria

|  |  |
| --- | --- |
| Criteria | Screenshot for evidence of: |
| A clean and intuitive, consistent design. | Main window uncluttered with large and clear interactive UI elements. |
| Main window with start menu UI including buttons to navigate to other menus. | Main window that shows start menu. |
| Main window with options menu UI with buttons to navigate to log in screen and back to start menu. | Main window showing options menu. |
| Main window with song selects UI including button to go back to start menu, song tabs with buttons to start the song and a slider to navigate through the songs. | Main window that shows song select. |
| Create and sign into accounts. Text shown on screen of which account is currently signed in. | Sign in window and difference between logged in and logged out. |
| Song selection screen with a range of songs to choose from at varying difficulties. | Entire song library including difficulties. |
| Notes of the song shown clearly and accurate to music theory, large and unambiguous. | Game screen with notes and stave. |
| Playing a note into the microphone gets registered by the program. | The note being registered and processed. |
| Get on the fly feedback of each note played in the song. Shown by an accuracy mark out of 10 in text at the top of the screen. | Video of a song being played, showing notes being hit accurately and notes being missed. |
| After-performance analysis of how well the song was played, with a rating based from how other users performed. Should be concise and easily interpretable. | Post-performance screen showing stats of the performance. |
| The song score to the account so it can be improved later. The user will be able to see the score of each song in the song select. | Song selection showing high scores on each song. |
| The program is easy to use and can be intuitively navigated. | Stakeholder input rating usability of the program out of 10 after one minute, five minutes and 30 minutes. |

# Design

## User interface design

### Main Menu

This will be the first screen the user will come to upon opening the program. It provides a central “hub” in which the user can access all the features of the program using large clearly visible and legible buttons to bring up further menus: “Song select”, “Options” and “Login”. You can also exit the program through use of the “Quit” button.

The design will be simple and uncluttered, with the buttons centred in the middle of the screen to make it as clear and concise as possible without restricting functionality. Graphics may be used in the background instead of a plain white screen – relating to music in some way such as musical notation or instruments.

Notably, the login button will feature on the main menu, as well as the options menu. This is so the user is aware of this feature immediately upon opening the program, so they don’t start using it without their progress being saved, as most users are unlikely to open the options menu before using going onto the song select.

I have opted to use as few menus for users to find each feature of the program as possible to make the program as easy to use as possible. No menus or options will be hidden behind a ribbon or drop-down menu, preferring the features to be immediately available on each menu. This is to try and minimize the amount of time a user will be looking through menus rather than using the program.

In order to have consistent button functionality and make the code simpler, I will be creating a button class that will handle all buttons across the program.

### Options Menu

The options menu will not be populated with many features however it will be necessary for volume control and profile management.

Changing the output volume of the program will allow users to decrease or turn off any sound the program will make if it is too loud for their preference or increase it if it is too quiet/they have poor quality speakers etc. This will be handled with a slider that ranges from 0% to 100% of the system volume value. The motive for the use of a slider is the visual aid it provides for how loud the program will be.

Profile management in the options menu will allow the user to sign in/out of their account (depending on if they are signed in or not already) as well as reset their recorded stats (such as performance on each song) if they wish to start over.

**The user will also be able to up the accuracy of the pitch detection if they have better hardware, which can be adjusted here.**

The back button will return the program to the main menu.

### Login Menu

The login menu will appear when a user uses the login button either on the homepage or in the options menu when signed out.

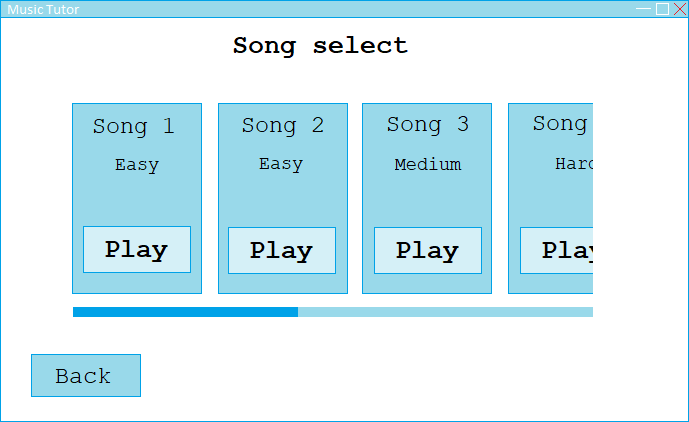
It features a username text box and a password text box in which the user can enter text.

The login button will take the inputted text from the username and password box and query it through the database of accounts. If the account exists and the password is correct, the user will be signed into their account. Otherwise, an error message will be displayed on screen.

The create account button will create the user a new account if the username is not already in the database (an error message will be displayed on screen if it is) and the user is signed in to their new account.

The back button will return the program to the main menu or options menu depending on where the login menu was called.

### Song Select Menu



All the songs will be listed based on difficulty rating (easy, medium, hard) and be displayed using a horizontal scroll bar. I have chosen using a scroll bar over alternate methods of displaying multiple items on screen due to its intuitive nature as well as clean look and functionality – sacrificing as little information as possible without taking too much screen space up. I intend on having 5-10 songs in the program and displaying them all on screen at once would clutter the screen and require each song tab to be very small, whilst making use of a drop-down menu, search bar or further sub menus and buttons would not make the songs visible to the user immediately, which may unnecessarily cause extra time spent in menus looking for a song.

Each song tab (song name, difficulty and play button) will enlarge upon the mouse hovering over it, to make it clear which song is currently being selected by the user. Each play button will start its respective song.

The back button will return to the main menu.

### Song Screen

The song screen will be the main section of the program. This will aim to provide all the necessary information to the user, so they can perform the chosen song. The clef (the symbol on the left of the 4/4-time signature) will change depending on what range the notes of the song are:

And the numbers to the right of the clef will signify the time signature and will display that accordingly.

Notes will move from the right-hand side of the screen along the staff (five black horizontal lines) and the user will be expected to play them as they pass through the blue “catchment” area. The score will increase for accurate performance and will show the user this in the top left of the screen.

The program will support a range of musical notation, including all notes from whole notes to sixteenth notes, bar lines, rests up to sixteenth rests, sharp, natural and flat accidentals, all key signatures, all common note relationship markings (ties, slurs, tuplets and chords) and dynamics from fortissimo (ff) to pianissimo (pp) as well as crescendos and diminuendos (however the program will not take into account the “loudness” of a performance due to limitations of the microphone input even though it will display these markings).

The program will not include trills, acciaccatura or tremolo notes as these are played too fast and will be extremely difficult to accurately measure performance. Repeat signs will not be included either as the song will be played continuously without jumping back and forth. Pedal notation will be abstracted too as most instruments won’t make use of it and it may cause issues with measuring timing (as using pedal artificially extends note length).

## Algorithms

The bulk of the program will consist of analysing the raw microphone input data and extracting the pitch from said input, using a Fast Fourier Transform.

For the GUI, I will use a class to handle menu control. Each menu will be in its own sub-routine within the main Application class; making it very easy to swap between menus and making readability of the code easier.

### Main Menu pseudocode

define menuScreen():

fill display white

song\_select\_screen\_button = button(text='Song Select')

options\_screen\_button = button(text='Options')

login\_screen\_button = button(text='Log In')

quit\_button = button(text='Quit')

title\_text = 'Music Maestro'

render title\_text

while True:

mouse\_position = get mouse position

if mouse button down:

if song\_select\_screen\_button.collide\_with\_point(mouse\_position):

songSelectScreen()

elif options\_screen\_button.collide\_with\_point(mouse\_position):

optionsScreen()

elif login\_screen\_button.collide\_with\_point(mouse\_position):

loginScreen()

elif quit\_button.collide\_with\_point(mouse\_position):

quit()

song\_select\_screen\_button.render()

options\_screen\_button.render()

login\_screen\_button.render()

quit\_button.render()

update display

return

### Song Select Screen

define songSelectScreen():

fill display white

back\_button = button(text='Back')

title\_text = 'Song Select'

render title\_text

song\_tabs = []

songs = getSongFiles()

for i=0, i=songs.length, i++:

if user.isSignedIn():

tab = createTab(song=songs[i],highscore=user.songs[i].highscore)

else:

tab = createTab(song=songs[i])

song\_tabs.append(tab)

scroll\_bar = scrollBar(scroll\_distance=(tab.width\*song\_tabs.length))

while True:

mouse\_position = get mouse position

mouse\_clicked = get if mouse is pressed

fill background behind tabs white

scroll\_bar.update(mouse\_position,mouse\_clicked)

back\_button.render()

for i=0, i=song\_tabs.length, i++:

tab = song\_tabs[i]

tab.set\_x(scroll\_bar.get\_notch\_position())

if screen.left < tab.get\_x() < screen.right:

tab.render()

tab.button.render(mouse\_position)

if mouse\_clicked == True:

if tab.button.collide\_with\_point(mouse\_position):

return performanceScreen(tab.song)

if mouse button down:

if back\_button.collide\_with\_point(mouse\_position):

menuScreen()

update display

return

### Options Screen

define optionsScreen():

fill display white

title\_text = 'Options'

render title\_text

user\_account\_text = 'User Account'

render user\_account\_text

back\_button = button(text='Back')

if user.isSignedIn():

logout\_button = Button(text='Log out')

delete\_account\_button = Button(text='Delete account')

else:

login\_button = Button(text='Log in')

while True:

mouse\_position = get mouse position

if mouse\_clicked == True:

if back\_button.collide\_with\_point(mouse\_position):

menuScreen()

if user.isSignedIn():

if logout\_button.collide\_with\_point(mouse\_position):

user = create new blank user instance

optionsScreen()

if delete\_account\_button.collide\_with\_point(mouse\_position):

remove user account

user = create new blank user instance

optionsScreen()

else:

if login\_button.collide\_with\_point(mouse\_position):

loginScreen()

back\_button.render()

if user.isSignedIn():

logout\_button.render()

delete\_account\_button.render()

else:

login\_button.render()

update display

return

### Login Screen

define loginScreen():

fill display white

title\_text = 'Log In'

render title\_text

username\_text = 'Username'

render username\_text

username\_input = TextInput()

password\_text = 'Password'

render password\_text

password\_input = TextInput(input\_hidden=True)

login\_button = Button(text='Log in')

create\_account\_button = Button(text='Create account')

back\_button = Button(text='Back')

while True:

mouse\_position = get mouse position

mouse\_clicked = get if mouse is pressed

keydown = get if a keyboard key has been pressed

if mouse\_clicked == True:

if back\_button.collide\_with\_point(mouse\_position):

menuScreen()

elif login\_button.collide\_with\_point(mouse\_position):

login(username\_input.get\_value(), password\_input.get\_value())

elif create\_account\_button.collide\_with\_point(mouse\_position):

create\_account(username\_input.get\_value(), password\_input.get\_value())

if keydown == True:

if key is acceptable character:

username\_input.key\_press(key)

password\_input.key\_press(key)

back\_button.render()

create\_account\_button.render()

login\_button.render()

update display

return

## Inputs, processes, outputs and storage

|  |  |  |
| --- | --- | --- |
| Input | Process/Storage | Output |
| Quit button | Break out of the main sub-routine, halt any remaining processes and close the PyGame window. | Close the window. |
| Menu buttons | Switch to the new menu state, disable current buttons and enable new menu buttons. | Display new menu and buttons. |
| Login text boxes | Store content of text box as a string-type variable. | Display the letters inputted in the text box. |
| Login button | Open account database and query username and password text box contents against all database entries. If a match is found set current account to username. | Display “signed in” message on screen or corresponding error message if the username and password were not matched. |
| Create account button | Open account database and query username text box contents against all database entries. If there is no match, write username and password to the database and save. Set current account to username. | Display “new account created” message or corresponding error message if the account could not be created. |
| Scroll bar | Change x position of each song tab in the song select menu by –1 \* (change in x of scroll bar). | Update position of the bar and each song tab on screen. |
| Play song button | Close all menus and initiate the song screen and the countdown. Then, read the respective song file and start the song. | Display the song screen and display countdown before the song starts. |
| Microphone | Perform a Fourier transform on the soundwave and extract pitch. Provide as input to the scoring algorithm. | Show the pitch as the y value of a bar on screen to give visual feedback of the pitch accuracy. |
| Return to song select button | Save song performance to account if signed in and enable song select menu. | Display song select menu on screen. |

# Development

## Analysing a microphone input as to predict the musical note played

### Coding initialisation of SoundData object

For a microphone input to be obtained in Python, a third-party module must be acquired as Python does not have any inbuilt tools for working with microphone inputs. The two most popular modules for performing such a task are sounddevice (<https://pypi.org/project/sounddevice/>) or PyAudio (<https://pypi.org/project/PyAudio/>). I chose to use PyAudio due to its lower level control over sounddevice, something that would be useful in extracting meaningful data from.

To feed Python a microphone input with PyAudio, a stream must be created that simply contains bytes making up the inputted sound.

audio\_stream = PyAudio().**open**(format=paInt16,

channels=1,

rate=44100,

**input**=True,

frames\_per\_buffer=1024)

This will create a PyAudio object using the given arguments:

* Format: paInt16 is the integer 8.
* Channels: only a mono signal is needed so to reduce data size only one channel will be used.
* Rate: the most common sampling frequency is 44.1kHz, so it is used here. Sampling frequency is the number of samples taken each second.
* Input: Set to true as we want to take an input not give an output.
* Frames per buffer: number of frames (or samples) in a buffer. Also known as chunk size. A chunk is simply a group of samples stored together.

A sample is an amplitude (loudness value) at a specific point in time.

Once a stream is initialised, byte data can be read from it using PyAudio’s read function.

audio\_stream.read(1024)

This will produce an output that looks like this:

b'\x00\x00z\xff\x83\xff\x9a\xff\x8a\xff\x8d\xff\x9d\xff\x9e\xff\x95\xff\x9b\xff\x95\xffz\xff\x88\xff\x95\xff\x94\xff\x90\xff\x8c\xff\x88\xff\x9f\xff\x97\xff\xa2\xff\xa4\xff\xa2\xff\xb2\xff\x9d\xff\x9d\xff\xa7\xff\xc3\xff\xb1\xff\xa0\xff\xac\xff\x9c\xff\xa7\xff\x9a\xff\x96\xff\x91\xff\x9c\xff\xa5\xff\x8f\xff\xaa\xff\xa6\xff\x93\xff\xa5\xff\x83\xff\...x00\xf8\x00\xec\x00\xd9\x00\xdd\x00'

Which is a chunk’s worth of sound data in hexadecimal.

To create a modular piece of code to collect and analyse sound data, a class object must be created for the main file to import in. I have opted to name this “SoundData” under the filename “audio.py”. Upon initialisation of the class, chunk, rate, and the PyAudio stream will be setup so they can be accessed throughout the class.

**try**:

**from** pyaudio **import** PyAudio, paInt16

**except** ImportError **as** e: # most likely a ModuleNotFoundError

**raise** Exception(f'Could not import a module: {e}.')

**class** SoundData:

**def** \_\_init\_\_(self, chunk=1024, rate=44100):

self.chunk = chunk

self.rate = rate

self.audio\_stream = PyAudio().**open**(format=paInt16,

channels=1,

rate=rate,

**input**=True,

frames\_per\_buffer=chunk)

### Testing initialisation of SoundData object

**if** \_\_name\_\_ == '\_\_main\_\_':

sound = SoundData() # Create a SoundData object

**print**(sound.chunk) # Print class variables

**print**(sound.rate)

**print**(sound.audio\_stream)

Will output:

1024

44100

<pyaudio.Stream **object** at 0x0731CA70>

This is the expected result.

### Coding reading stream data

Creating a high-level function for reading the stream for a specified timeframe will be key for getting data in the performance of the program and will allow for adjustments to maximize the efficiency of the program, i.e. if the program runs slowly a smaller timeframe can be specified.

To allow for custom timeframes, a custom buffer must be made to store the data. Using a provided timeframe in seconds, a calculation can be done to find how many chunks of the stream are needed to fill the timeframe in the buffer.

Reading the stream a chunk at a time, a loop can be implemented to add stream data to a list.

Will be how many times to iterate through the loop. As the rate is measured in samples per second, the total samples needed can be identified by multiplying the rate by the given timeframe.

For example, for a timeframe of 0.1 seconds at a rate of 44100 samples per second, a total of 4410 samples are needed. If we read a chunk of samples (1024 samples) each loop iteration, that is 4 loop iterations to get roughly 4410 samples.

**def** stream(self, time=.1):

self.buffer = [self.audio\_stream.read(self.chunk) **for** i **in** **range**(**int**(self.rate/self.chunk\*time))]

### Coding the framing function

Now we have a buffer full of sound data, the dominant frequency of the sound can be identified. The sound data is technically a spectrum of frequencies, which is why the dominant frequency must be found rather than just “the” frequency of the sound. To do this, first a framing function must be applied to the sound data in order to overlap individual samples. This will reduce data degradation when a window function is applied to the sound data as window functions are quite lossy at each edge of the window.

First, frame/sample and signal length must be identified, as well as the total number of frames and frame step.

**def** \_framing(self, data):

frame\_length = **int**(.025 \* self.rate)

frame\_step = **int**(.01 \* self.rate)

signal\_length = **len**(data)

number\_of\_frames = **int**(np.ceil(**abs**(signal\_length-frame\_length)/frame\_step))

0.025 is the window length, chosen arbitrarily.

0.01 is also chosen arbitrarily.

np, used in number\_of\_frames, is shorthand for the third-party numpy module. It will be highly significant throughout the audio signal analysis.

Assuming a signal length of 4410 and rate of 44100 Hz, the number of frames would be:

Taking the ceiling of this result returns the number of frames as 8 frames, and stops the result ever being less than one (as there must be at least one frame to apply the framing function on).

Then, a two-dimensional array is created by generating a list of values between 0 and the frame length and then repeated for the number of frames desired. In the example case:

np.arange(0, 1102.5) = [ 0 1 2 ... 1099 1100 1101]

And thus:

np.tile(np.arange(0, 1102), (8, 1)) = array([[ 0, 1, 2, ..., 1099, 1100, 1101],

[ 0, 1, 2, ..., 1099, 1100, 1101],

[ 0, 1, 2, ..., 1099, 1100, 1101],

...,

[ 0, 1, 2, ..., 1099, 1100, 1101],

[ 0, 1, 2, ..., 1099, 1100, 1101],

[ 0, 1, 2, ..., 1099, 1100, 1101]])

A second two-dimensional array is then created;

index\_b = np.tile(np.arange(0, number\_of\_frames\*frame\_step, frame\_step), (frame\_length, 1))

Which produces:

np.tile(np.arange(0, 8\*441, 441), (1102, 1)) = array([[ 0, 441, 882, ..., 2205, 2646, 3087],

[ 0, 441, 882, ..., 2205, 2646, 3087],

[ 0, 441, 882, ..., 2205, 2646, 3087],

...,

[ 0, 441, 882, ..., 2205, 2646, 3087],

[ 0, 441, 882, ..., 2205, 2646, 3087],

[ 0, 441, 882, ..., 2205, 2646, 3087]])

Then, its axes are swapped so rows are columns and columns are rows:

[ 0, 0, 0, ..., 0, 0, 0],

[ 441, 441, 441, ..., 441, 441, 441],

[ 882, 882, 882, ..., 882, 882, 882],

...,

[2205, 2205, 2205, ..., 2205, 2205, 2205],

[2646, 2646, 2646, ..., 2646, 2646, 2646],

[3087, 3087, 3087, ..., 3087, 3087, 3087]

Finally, both arrays are added together which produces the final indices array:

[ 0, 1, 2, ..., 1099, 1100, 1101],

[ 441, 442, 443, ..., 1540, 1541, 1542],

[ 882, 883, 884, ..., 1981, 1982, 1983],

...,

[2205, 2206, 2207, ..., 3304, 3305, 3306],

[2646, 2647, 2648, ..., 3745, 3746, 3747],

[3087, 3088, 3089, ..., 4186, 4187, 4188]

Next, the buffer needs to be padded with zeros as to be at least the same length as the indices array.

padding\_amount = number\_of\_frames \* frame\_step + frame\_length

padding = np.zeros((padding\_amount-signal\_length)) # Creates a numpy array filled entirely of zeros

padded\_buffer = np.append(data, padding) # Merges two arrays into one

Finally, get the desired frames from the padded buffer:

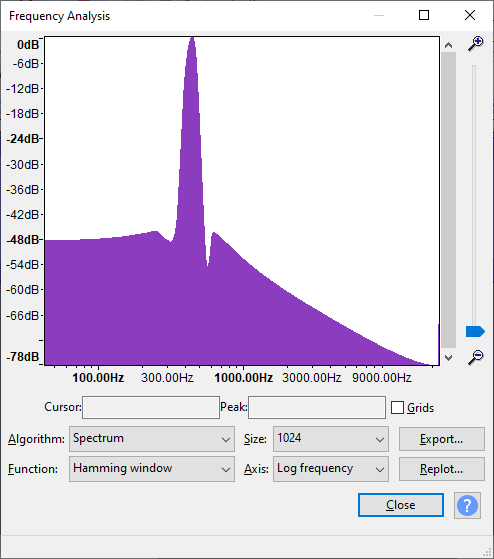
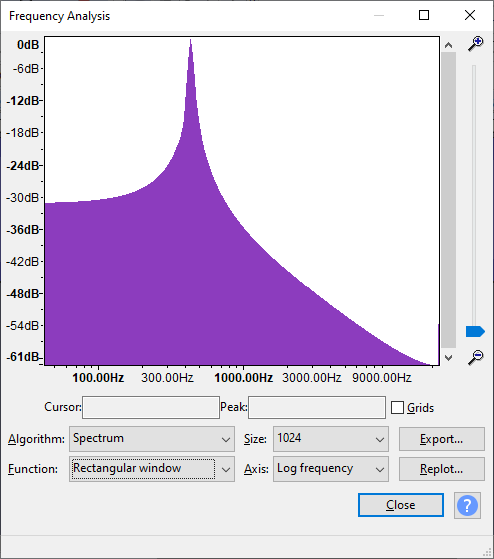
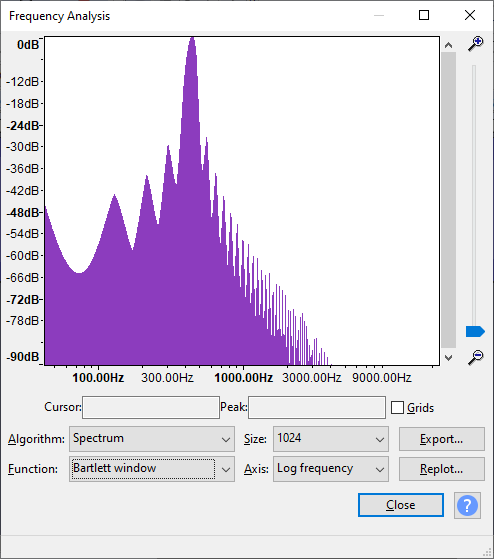
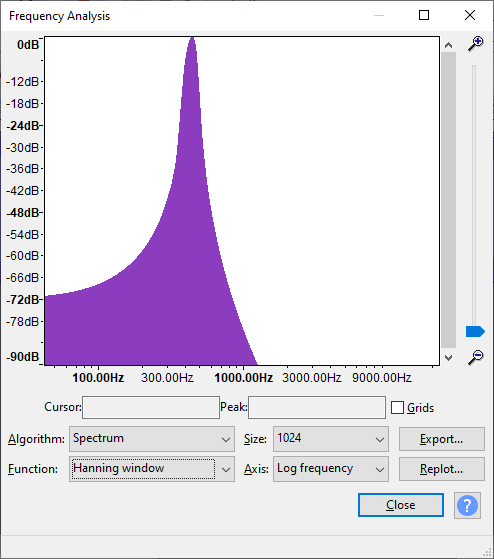
frames = padded\_buffer[indices]

**return** frames, frame\_length

### Coding the window function

After performing the framing function, a window function is applied on the frames. A window function is nonzero for some period of time, and then zero before and after that period. When multiplied by the frames, it produces an output of 0 except during the nonzero portion of the window, when it exposes the frames.

For the program I have decided to use a Hamming window, as it usually gives the most accurate results in this use case compared to other windows. On a 440Hz signal, different window functions will yield different results:

The difference between functions is not very noticeable on a small scale, so really any window function is fine to use.

The Hamming function is as follows:

However, numpy already has a Hamming function, making the task simpler:

# Perform framing on the signal

frames, frame\_length = self.\_framing(self.buffer)

# Perform Hamming window function on the frames

windows = frames \* np.hamming(frame\_length)

### Testing the framing and window functions

Upon running driver code to test the window function, an error occurred:

TypeError: ufunc 'multiply' did not contain a loop with signature matching types dtype('S2048') dtype('S2048') dtype('S2048')

After some debugging, I found this was caused due to the buffer being a bytes object. Instead, it needed to be a numpy array with integer values.

I found that the third-party scipy module contained a wavfile function that returned the results of a Wave file as a numpy array formatted correctly for the error to be resolved. What this meant was that by writing the buffer contents to a Wave file and then reading said file back into the buffer, the issue would be solved.

I created a new function to write the buffer contents to a file using the in-built wave module:

**def** \_write\_stream\_to\_file(self, filename, data):

wave\_file = wave.**open**(f'../assets/{filename}.wav', 'wb') # Open the Wave file in binary write mode

wave\_file.setnchannels(1) # Set details of the data being written

wave\_file.setsampwidth(PyAudio().get\_sample\_size(paInt16))

wave\_file.setframerate(self.rate)

wave\_file.writeframes(b''.join(data)) # Convert the list into a binary string and (over)write to the Wave file

wave\_file.close()

This would create a Wave file in ./assets/ with the filename “buffer”, containing the bytes data of the buffer.

Then, the stream function could be rewritten to account for the change:

**def** stream(self, time=.1):

buffer\_hex = [self.audio\_stream.read(self.chunk) **for** i **in** **range**(**int**(self.rate/self.chunk\*time))]

self.\_write\_stream\_to\_file('buffer', buffer\_hex)

self.rate, self.buffer = wavfile.read('../assets/buffer.wav')

### Coding finding the dominant frequency of a frame

First, a function to find the dominant frequency of any given frame must be created. In that function, it would begin by performing a fast Fourier transform on the frame. The fast Fourier transform is a mathematical method for transforming a function of time into a function of frequency.

**def** \_get\_dominant\_frequency(self, frame):

nfft = 2\*\*14 # Fast Fourier transform points to be calculated

fourier\_transform = np.fft.rfft(frame, nfft) # Perform a fast Fourier transform on a real input

The larger the nfft variable, the more accurate the result will be, at the cost of processing time.

The typical Fourier transform output will include a real part and imaginary part:

[-4292.03923912 +0.j -4143.73492452 +965.22073516j

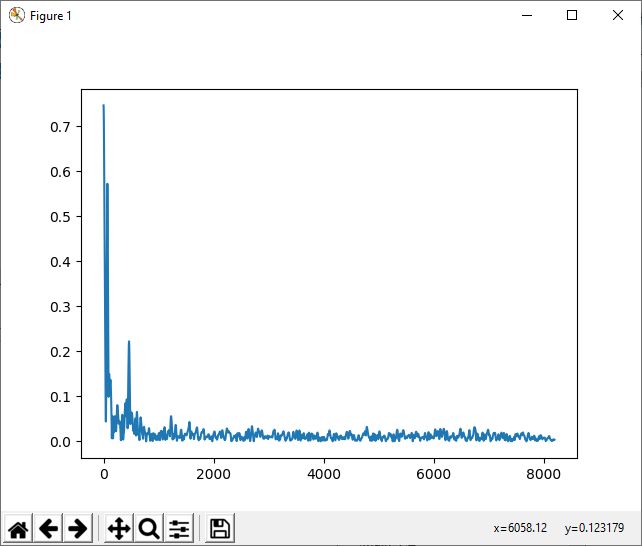
-3715.56519723+1833.79068782j ... -51.41734305 -25.12434982j

-54.92432583 -12.74663432j -56.10118959 +0.j ]

These can be filtered to just the real part by taking the absolute value of the output.   
Next, a magnitude and thus power spectrum can be generated using the Fourier transform:

magnitude\_spectrum = (1/nfft) \* **abs**(fourier\_transform)

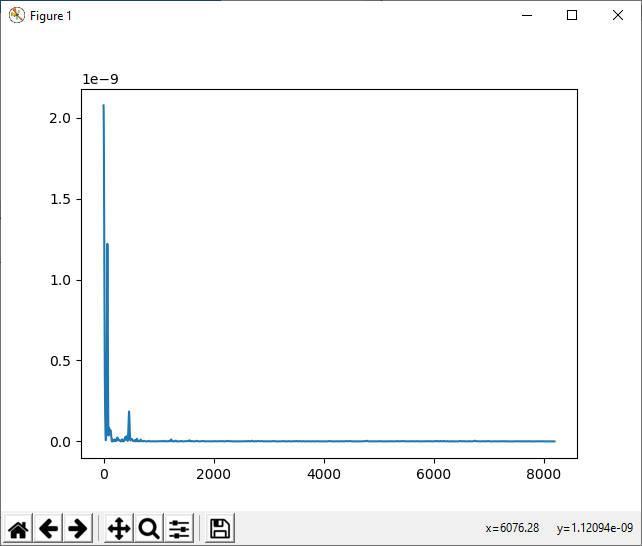
Which produces:



This graph indicates that the microphone had little input (the large spike at 0 on the x-axis) with a small amount of background noise.

power\_spectrum = (1/nfft)\*\*2 \* magnitude\_spectrum\*\*2

Which produces:



Making it very clear where the point of interest is. In this case, as no sound except whatever imperfections the physical components of the microphone has and any interference in the wire was present, there is a large spike at 0, indicating a lack of sound.

Using the power spectrum, we can then find the frequencies associated with the coefficients of the power spectrum array (each index). We only need the positive results, and because of this a floor division of 2 must be performed, before adding 1 to each value.

power\_spectrum = power\_spectrum[:**len**(frequencies)]

Then, simply find the index of the highest value and return its associated frequency:

maxiumum\_index = np.argmax(power\_spectrum) # .argmax() returns the maximum values along an axis

**return** frequencies[maxiumum\_index]

### Coding finding the dominant frequencies of a group of frames

After performing the framing and window functions on the buffer, find the dominant frequency of each frame in the window function output, and collect the data into a master list. Then, round and remove duplicate data and the dominant frequencies will have been identified:

**def** get\_dominant\_frequencies(self):

# Perform framing on the signal

frames, frame\_length = self.\_framing(self.buffer)

# Perform Hamming window function on the frames

windows = frames \* np.hamming(frame\_length)

dominant\_frequencies = np.array([self.\_get\_dominant\_frequency(window) **for** window **in** windows])

dominant\_frequencies = np.**round**(dominant\_frequencies, 3) # Round to three decimal places

dominant\_frequencies = np.unique(dominant\_frequencies) # Remove all duplicate values

**return** dominant\_frequencies

### Testing finding the dominant frequencies of a stream

Running the driver code:

**if** \_\_name\_\_ == '\_\_main\_\_':

sound = SoundData()

**while** True:

sound.stream()

**print**(sound.get\_dominant\_frequencies())

Will produce a stream of outputs in the console showing dominant frequencies:

[ 1. 157. 175. 189. 197.]

[ 1. 127. 146. 165. 227.]

[ 1. 259.]

[ 1. 165. 175. 184.]

[ 1. 22. 165.]

[ 1. 35. 41.]

[ 1. 30. 35. 162. 178. 189. 264.]

[ 1. 184. 186.]

[ 1. 178. 186.]

[ 1. 138.]

[ 1. 127. 140. 149.]

[ 1. 35. 181.]

[ 1. 114. 127. 130. 135.]

[ 1. 27. 184.]

[ 1. 162. 167. 175.]

[ 1. 30. 173.]

[ 1. 38. 248.]

[ 1. 122. 151. 167.]

[ 1. 3. 124. 143.]

[ 1. 189.]

[ 1. 186.]

[ 1. 6. 130. 165. 167.]

[ 1. 25. 35. 167.]

[ 1. 157. 167. 170. 184. 253. 267.]

[ 1. 138. 149. 175. 189. 286.]

[ 1. 41. 184. 243. 259.]

[ 1. 33. 170.]

[ 1. 14. 27. 33. 218.]

[ 1. 130. 149. 170. 173. 178. 259.]

[1.]

To test whether it is accurate, however, a single tone of 440Hz can be played through the use of a tone generator directly into the microphone to get near perfect results, as shown:

[ 1. 35. 178. 436. 439.]

[ 1. 436. 439. 445.]

[ 1. 439. 442.]

[ 1. 439.]

[ 1. 436. 439. 442.]

[436. 439. 442.]

[ 1. 436. 439. 442. 447.]

[ 1. 439. 442.]

[ 1. 439. 442. 447.]

[ 1. 436. 439. 442.]

[ 1. 439. 445.]

[ 1. 439. 442.]

[439. 442. 445.]

[ 1. 439. 442.]

[ 1. 439. 442. 445.]

[436. 439. 442. 445.]

[ 1. 442.]

[439. 442.]

[439. 442.]

[439. 442.]

[439. 442.]

[439. 442.]

[439. 442.]

[442.]

[442.]

[442.]

[442.]

Clearly indicating that the audio processing works as intended.

### Coding conversion of frequency to note name

Finally, to finish the SoundData class, a conversion from frequency to note value is to be created. Taking an input of frequencies and a dictionary of note names and associated frequencies, the function should return the correct note corresponding with the frequencies provided.

The notes dictionary should look something like:

{'A' :[440],

'A#/Bb':[466],

'B' :[493],

'C' :[523],

'C#/Db':[554],

'D' :[587],

'D#/Eb':[622],

'E' :[659],

'F' :[698],

'F#/Gb':[739],

'G' :[783],

'G#/Ab':[830]}

The reason for each frequency being in a list (as well as the frequencies argument being a list) is to allow for multiple frequencies for one note, which may be needed in some cases.

First, each note in the notes dictionary should be iterated through, with a nested loop going through each of the provided frequencies.

**def** get\_note\_from\_frequency(self, notes\_dict, frequencies):

**for** note **in** notes\_dict.keys():

target = notes\_dict[note]

weight = 0

**for** freq **in** frequencies:

min\_distance\_from\_target = **min**([**abs**(np.sin((np.pi\*freq)/value)) **for** value **in** target])

**if** **not** min\_distance\_from\_target:

min\_distance\_from\_target = -100

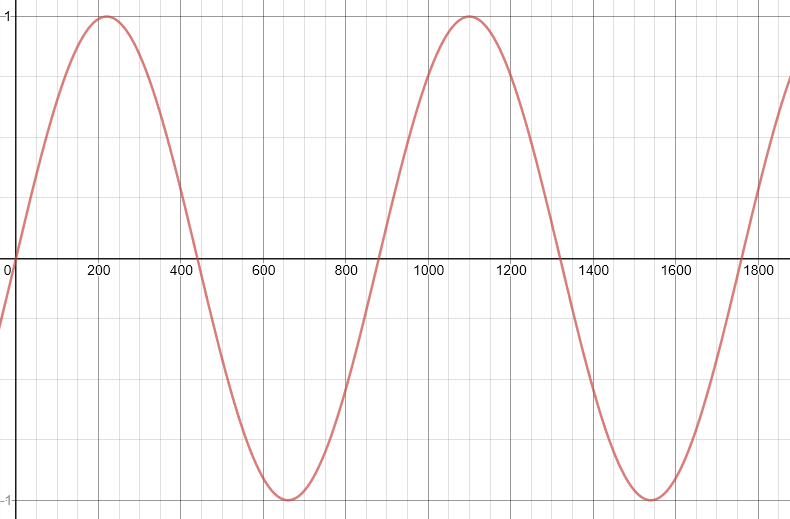
weight += min\_distance\_from\_target

The target variable is the frequencies of the note on that loop iteration.

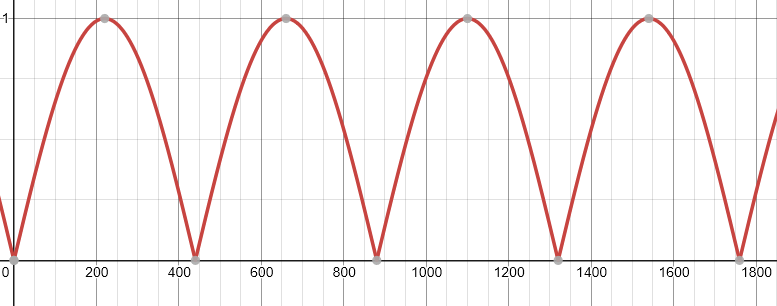
weight is set at 0 and increased by the distance each frequency in frequencies is from the target value. However, if distance from the target is 0, then the frequency is almost guaranteed to be the target note, so weight is set very low to ensure success of that frequency.

Finding the distance from the target is done first by calculating a sine wave of:

Which produces a sine wave with roots that are multiples of w. So, if w = 440 Hz;



We only want positive values though (but still a continuous function), so the absolute value is taken;



The further away x (in this case frequency) is from a multiple of w (value), the larger the output will be.

Calculate the frequency’s distance from each frequency in the target, and find the minimum distance for the best case scenario.

**for** freq **in** frequencies:

min\_distance\_from\_target = **min**([**abs**(np.sin((np.pi\*freq)/value)) **for** value **in** target])

**if** **not** min\_distance\_from\_target:

min\_distance\_from\_target = -100

weight += min\_distance\_from\_target

**try**:

**if** weight < closest\_match[1]:

closest\_match = [note, weight]

**except** NameError: # On the first iteration closest\_match has not yet been declared

closest\_match = [note, weight]

Finally, replace closest\_match with the new note if the weight is lower than the old closest\_match note’s weight.

When all notes have been tested against all frequencies, return the closest match to give the likeliest note from the frequency.

### Testing the conversion of frequency to note name

Using:

sound = SoundData()

**print**(sound.get\_note\_from\_frequency({'A' :[440],

'A#/Bb':[466],

'B' :[493],

'C' :[523],

'C#/Db':[554],

'D' :[587],

'D#/Eb':[622],

'E' :[659],

'F' :[698],

'F#/Gb':[739],

'G' :[783],

'G#/Ab':[830]}, np.array([444,451,430,449])))

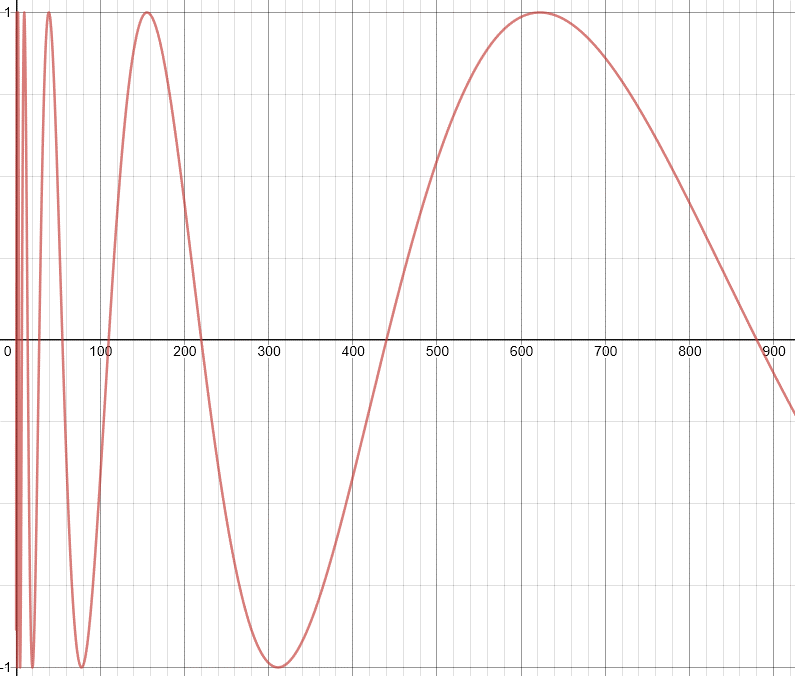
A plan of tests can be made to determine accuracy:

|  |  |  |
| --- | --- | --- |
| Frequencies | Expected Result | Actual Result |
| **444, 451, 430, 449** | A | A |
| **38, 1100, 659, 630** | E | D#/Eb |
| **38, 659, 630** | E | E |
| **880** | A | A |
| **440** | A | A |
| **220** | A | G#/Ab |

Unfortunately, the min\_distance\_from\_target formula failed the test plan. This indicates that this is not the correct equation to be using. After some consideration, I realised having the roots of the sine wave as multiples was not precise enough, as each rising octave of a note (for example from a low A to the next A up in pitch) was always double the frequency, and the falling octave was half. This means each root should be double the last.

The formula I found to do the trick is as follows:

Which produces the graph:



With w=440, you can see each root is at an octave of A, so will make the nearest note identification much more accurate.

Using the new equation:

min\_distance\_from\_target = **min**([**abs**(100\***round**(np.sin((np.pi/np.log(2))\*np.log(freq/value)),4)) **for** value **in** target])

The new tests all come out positive:

|  |  |  |
| --- | --- | --- |
| Frequencies | Expected Result | Actual Result |
| **444, 451, 430, 449** | A | A |
| **38, 1100, 659, 630** | E | E |
| **38, 659, 630** | E | E |
| **880** | A | A |
| **440** | A | A |
| **220** | A | A |

### Prototype of note detection

Using some driver code for the audio.py module, the following prototype is created:

**if** \_\_name\_\_ == '\_\_main\_\_':

notes = {'A' :[440],

'A#/Bb':[466],

'B' :[493],

'C' :[523],

'C#/Db':[554],

'D' :[587],

'D#/Eb':[622],

'E' :[659],

'F' :[698],

'F#/Gb':[739],

'G' :[783],

'G#/Ab':[830]}

sound = SoundData()

**while** True:

sound.stream()

**print**(sound.get\_note\_from\_frequency(notes, sound.get\_dominant\_frequencies()))

In which upon running the note currently detected is displayed in the console. This will allow the stakeholders to try out the note detection, using a variety of instruments, locations, and microphone qualities to see how versatile the algorithm is.

#### *Stakeholder email*

*Hello,*

*Here is the first prototype of the core mechanic of the program. By running the program, you will see a stream of notes in the console of which indicate what note is currently being detected in the microphone. If you try playing your instrument, you should find it recognises notes correctly.*

#### Responses

David:

*Working good for me – playing the saxophone at it produces the correct note. I think it should have its own window rather than be in the console though.*

Erin:

*I tried singing at the program as well as playing some notes on the piano. It works very well.*

Max:

*I have the issue where under certain circumstances the note detected will always be two or three semitones below the correct value. That is something to look into.*

The actual program will be in its own window, so the first criticism is not too much of a concern. However, the issue raised with incorrect note detection seems to be something that would be caused by the surroundings or microphone, something the user could not control. For that reason, I will later add a way to change the frequencies associated with the notes in the dictionary passed to the get\_note\_from\_frequency function as this should hopefully allow the program to be tailored to the unique conditions of the user.

## Creating the application window and main menu

### Coding the Application class

The whole application will be its own class as this will allow for certain variables to be shared seamlessly to different functions/parts of the program.

To begin, import the required modules and create the Application class:

#!/usr/bin/env python3

**try**:

**import** pygame

**from** pygame.**locals** **import** \*

**from** scripts.audio **import** SoundData

**except** ImportError **as** e: # most likely a ModuleNotFoundError

**raise** Exception(f'Could not import a module: {e}.')

**class** Application:

**def** \_\_init\_\_(self):

pygame.init() # initialize all imported pygame modules

pygame.mouse.set\_cursor(\*pygame.cursors.arrow)

self.screen = pygame.display.set\_mode((1280, 720)) # initialize a window

pygame.display.set\_caption('Music Maestro') # set the text in the window caption (top left)

self.clock = pygame.time.Clock()

self.audio = SoundData()

self.notes = {'A' :[440],

'A#/Bb':[466],

'B' :[493],

'C' :[523],

'C#/Db':[554],

'D' :[587],

'D#/Eb':[622],

'E' :[659],

'F' :[698],

'F#/Gb':[739],

'G' :[783],

'G#/Ab':[830]}

The shebang on the first line indicates to the interpreter the correct version to run the program if run from the shell, in this case the latest version of Python 3.

I have included the audio.py SoundData class ready for future use, as well as the default note-frequency dictionary.

Finally, the pygame.time.Clock instance will allow for control of the FPS of the program, which will be useful in controlling moving parts of the program, and general timing.

Next, I created the high-level run and quit functions, to allow easy start and close of the program:

**def** run(self):

'''

    Higher level initilization of the program.

    '''

# Call main menu function here

**return**

**def** quit(self):

'''

    End all PyGame processes and close the PyGame window.

    '''

pygame.font.quit()

pygame.quit()

**return**

Then, some driver code to start the program when the file is run:

**if** \_\_name\_\_ == '\_\_main\_\_': # Only run if the program file is run

MusicMaestro = Application()

MusicMaestro.run()

**raise** SystemExit

### Testing initialisation of the Application class

Placing a print statement after calling the run function will show the details of the Application instance:

**print**(MusicMaestro.\_\_dict\_\_)

Yields:

{'screen': <Surface(1280x720x32 SW)>, 'clock': <Clock(fps=0.00)>, 'notes': {'A': [440], 'A#/Bb': [466], 'B': [493], 'C': [523], 'C#/Db': [554], 'D': [587], 'D#/Eb': [622], 'E': [659], 'F': [698], 'F#/Gb': [739], 'G': [783], 'G#/Ab': [830]}}

And upon running the program, an empty window should be created due to the line:

self.screen = pygame.display.set\_mode((1280, 720))



I chose these dimensions 1280x720 pixels as it creates a fairly large window that is commonly used and familiar, ready for use by the program. I intend for the program to load immediately into the main menu so the next step will be to create the main menu function.

### Coding the main menu screen function

The run function will now contain the call to the main menu screen function, named \_menuScreen:

**def** run(self):

self.\_menuScreen()

**return**

The \_menuScreen function starts off by setting the background of the window to white and displaying the title text.

**def** \_menuScreen(self):

# background color

self.screen.fill((255,255,255))

# title text

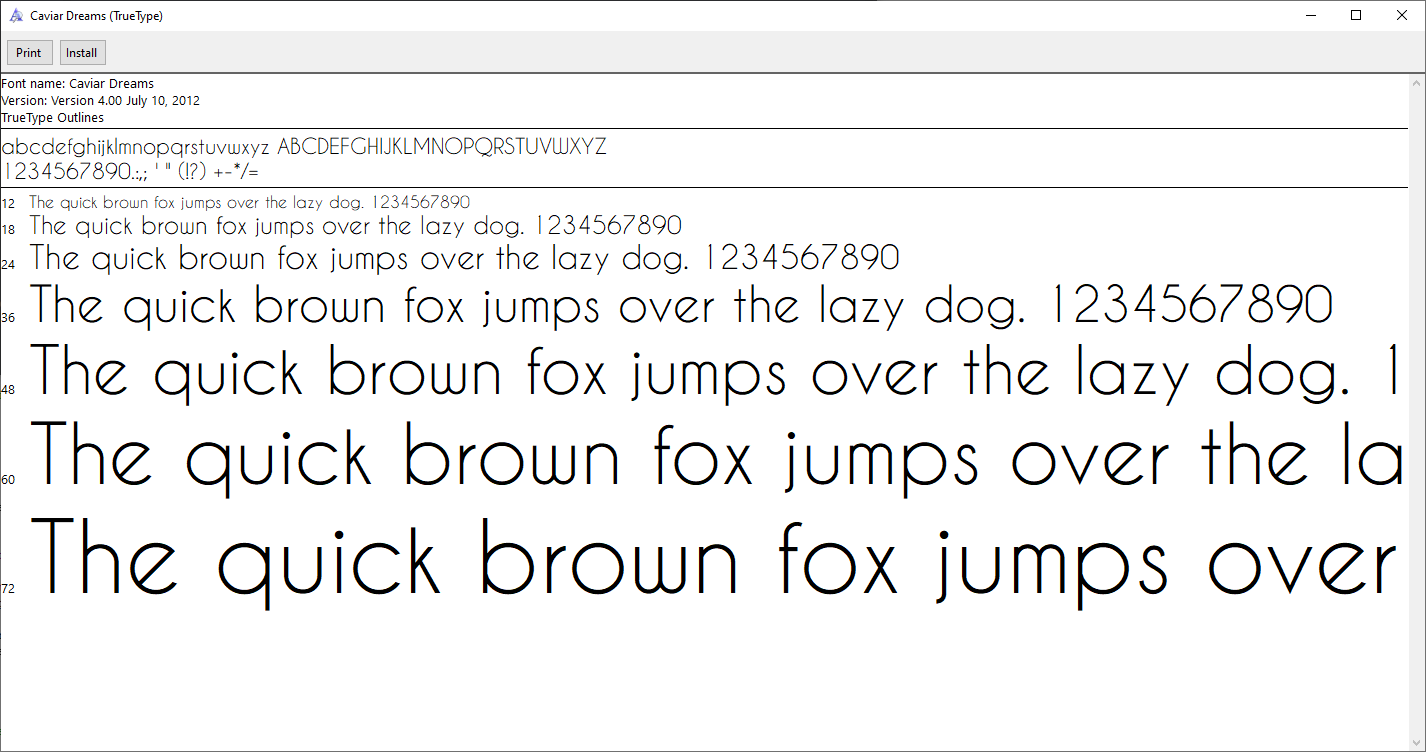
title = self.font['70'].render('Music Maestro', True, (0,0,0))

self.screen.blit(title, (190, 135))

In anticipation of there being many various sizes of text throughout the program, I created a dictionary of various font sizes in the initialisation of the class:

self.font = {**str**(i):pygame.font.Font('.\\assets\\font.ttf', i) **for** i **in** **range**(10,510,10)} # Load various font sizes

Which uses a custom font file and loads it in with every multiple of 10 as a font size up to 500 inclusive.



Next, a loop is created to keep the window open and active:

# event loop

**while** True:

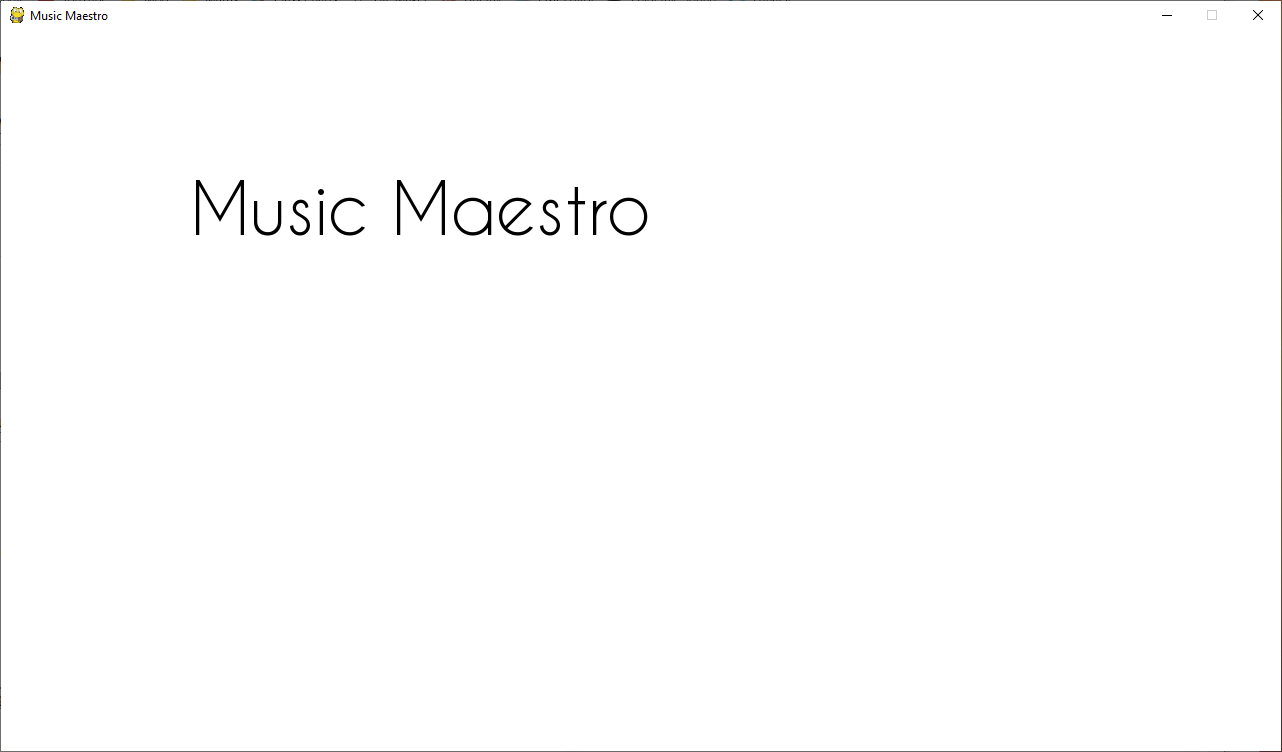
pygame.display.flip()

**return**

pygame.display.flip updates the entire screen, applying any new graphical changes that may have occurred.

### Testing the main menu window

Upon running the program, a white window with the title text is successfully initialised:



However, when attempting to close the window using the X in the top right of the window, the program freezes, becomes unresponsive and needs task manager to close it fully.

To solve this, I implemented the following piece of code into the while loop:

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

In PyGame, events are ways of interacting with the program. Event objects will have a type and other attributes to make interaction possible. Here, the QUIT event type signifies if the program is closed using the red X in the top right of the window, or if Alt+F4 is pressed on the keyboard.

If the QUIT event occurs, the program returns out of the \_menuScreen function immediately and calls the quit function to close the window and end the program cleanly. Now, the program opens and closes cleanly.

### Coding buttons

#### Initialisation

Each button will be an object from a button class, allowing for high modularity. The Button class will be written in a separate python file containing all UI elements, ui\_elements.py.

**try**:

**import** pygame

**from** pygame.**locals** **import** \*

**except** ImportError **as** e: # most likely a ModuleNotFoundError

**raise** Exception('Could not import a module: %s.' % e)

**class** Button:

**def** \_\_init\_\_(self, ctx, text, text\_size, position, dimensions, color, alt\_color, hover\_color, text\_color):

self.ctx = ctx

self.text = text

self.text\_size = text\_size

self.position = position

self.dimensions = dimensions

self.colors = {'primary':color ,

'alt' :alt\_color ,

'hover' :hover\_color,

'text' :text\_color }

Initialising the class, it has nine parameters;

* ctx: the Application instance required for rendering the button;
* text: the text to be displayed on the button;
* text\_size: text height in pixels;
* position: a tuple or list of two values stating the coordinates of the button;
* dimensions: a tuple or list of two values indicating the width and height the rectangular button should be;
* color: the colour of the centre of the button (tuple or list with three values: (R, G, B));
* alt\_color: the colour of the button border (tuple or list with three values: (R, G, B));
* hover\_color: the colour of the centre of the button upon mouse hover (tuple or list with three values: (R, G, B));
* text\_color: the colour of the button text (tuple or list with three values: (R, G, B)).

These are all the required information to make a button. The next step is to create a Rect object for the button. PyGame Rect objects are a way of storing rectangular coordinates and are very useful as hitboxes and for displaying rectangles on-screen. PyGame Rect objects are created as:

pygame.Rect( (top left coordinate of rectangle) , (width, height) )

Using the formulas:

Will produce the coordinates of the top left coordinate of the rectangle, using desired cords based on the centre of the rectangle. This allows Button instances to be centred around the position coordinates, which is generally more intuitive.

self.rect = pygame.Rect((**round**(position[0]-(dimensions[0]\*0.5)),

**round**(position[1]-(dimensions[1]\*0.5))),

(dimensions[0],dimensions[1]))

#### Rendering

To render the button is fairly simple, drawing a rectangle with the alt colour, then a slightly smaller rectangle inside with the primary colour, creating the border effect:

**def** render(self, color):

pygame.draw.rect(self.ctx.screen, self.colors['alt'], self.rect) # draw the button frame

pygame.draw.rect(self.ctx.screen, color, pygame.Rect((self.rect.x+5,self.rect.y+5),

(self.rect.width-10,self.rect.height-10)))

This will create a border of the alt colour 5 pixels wide around the middle section of the button.

Then, place the text onto the button:

font = pygame.font.Font('.\\assets\\font.ttf', self.dimensions[1]-self.text\_size)

label = font.render(self.text, True, self.colors['text'])

self.ctx.screen.blit(label, (self.position[0]-self.dimensions[0]\*.45,self.position[1]-self.dimensions[1]\*.45))

Finally, call the render function when initialising the button:

self.render(self.colors['primary'])

#### Checking for mouse hover

The final function the button needs is to detect whether the mouse is hovering over the button. If it is, render the button using the mouse hover colour, otherwise use the primary colour. To achieve this, a useful property of PyGame Rect objects can be used – the collidepoint function each Rect object has. This returns true if any point within the Rect boundaries is in contact with the point provided to the function:

**def** check(self, mouse\_pos):

**if** self.rect.collidepoint(mouse\_pos): # if the mouse is over the button

self.render(self.colors['hover'])

**return** True

**else**:

self.render(self.colors['primary'])

**return** False

It is expected that each loop in the main program, check will be called to render the button correctly.

### Testing the buttons

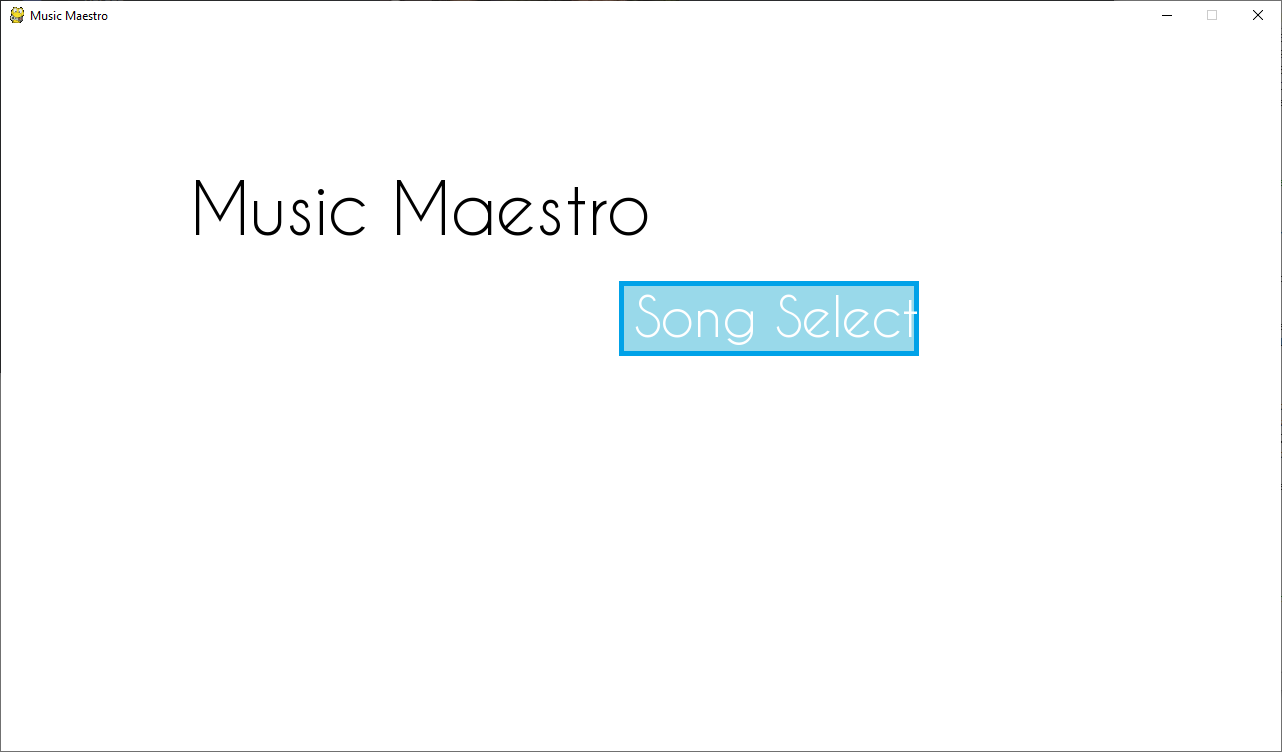
To test the Button class I will place one into the main program. Before the loop, a button is created using the following arguments:

* text = ‘Song Select’
* text size = 22
* position = x: 768, y: 288
* dimensions = width: 300, height: 75
* colour = light blue
* alt colour = darker blue
* hover colour = middle blue
* text colour = white

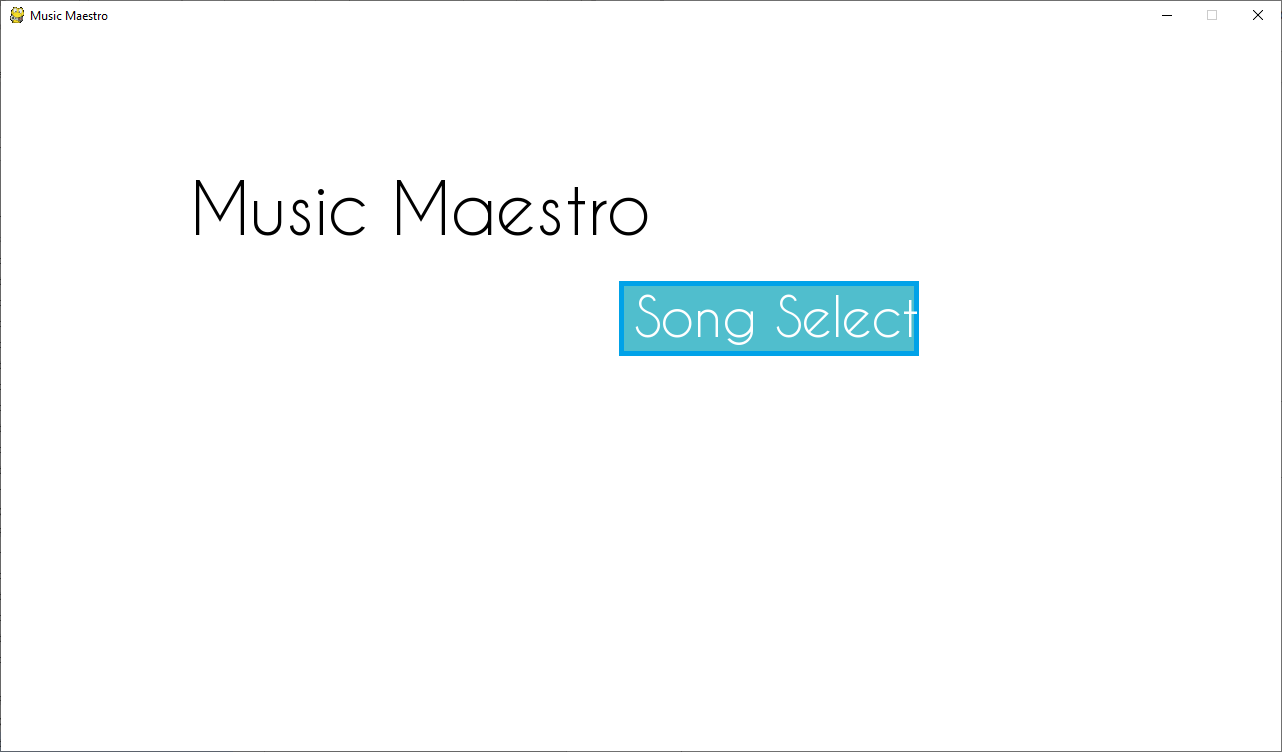
# ui elements

song\_select\_button = Button(self, text='Song Select', text\_size=22, position=[768,288], dimensions=[300,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

Placing a check function call into the loop and running the program will yield:



With the hover colour being applied when the mouse hovers over the button:



To make the button clickable, we can look to the MOUSEBUTTONDOWN event. This occurs when the mouse button is clicked, and only occurs once per mouse held down. Then, by performing another check call, if it returns positive, we know the button was clicked as the mouse was down and hovering over said button, and can perform some task.

**if** event.**type** == MOUSEBUTTONDOWN:

**if** song\_select\_button.check(mouse\_pos):

**print**('Button clicked!')

Which then clicking the button four times produces:

Button clicked!

Button clicked!

Button clicked!

Button clicked!

In the console.

To finish, the rest of the menu buttons are added, and to make things more compact are placed in a list with their corresponding click actions:

# ui elements

song\_select\_button = Button(self, text='Song Select', text\_size=22, position=[768,288], dimensions=[300,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

options\_button = Button(self, text='Options' , text\_size=22, position=[768,378], dimensions=[300,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

login\_button = Button(self, text='Log In', text\_size=22, position=[698,468], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

quit\_button = Button(self, text='Quit' , text\_size=22, position=[856,468], dimensions=[125,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

buttons = [[song\_select\_button, self.\_songSelectScreen],

[options\_button , self.\_optionsScreen ],

[login\_button , self.\_logInScreen ],

[quit\_button , self.quit ]]

**while** True:

mouse\_pos = pygame.mouse.get\_pos()

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

**if** event.**type** == MOUSEBUTTONDOWN:

**for** button **in** buttons:

**if** button[0].check(mouse\_pos):

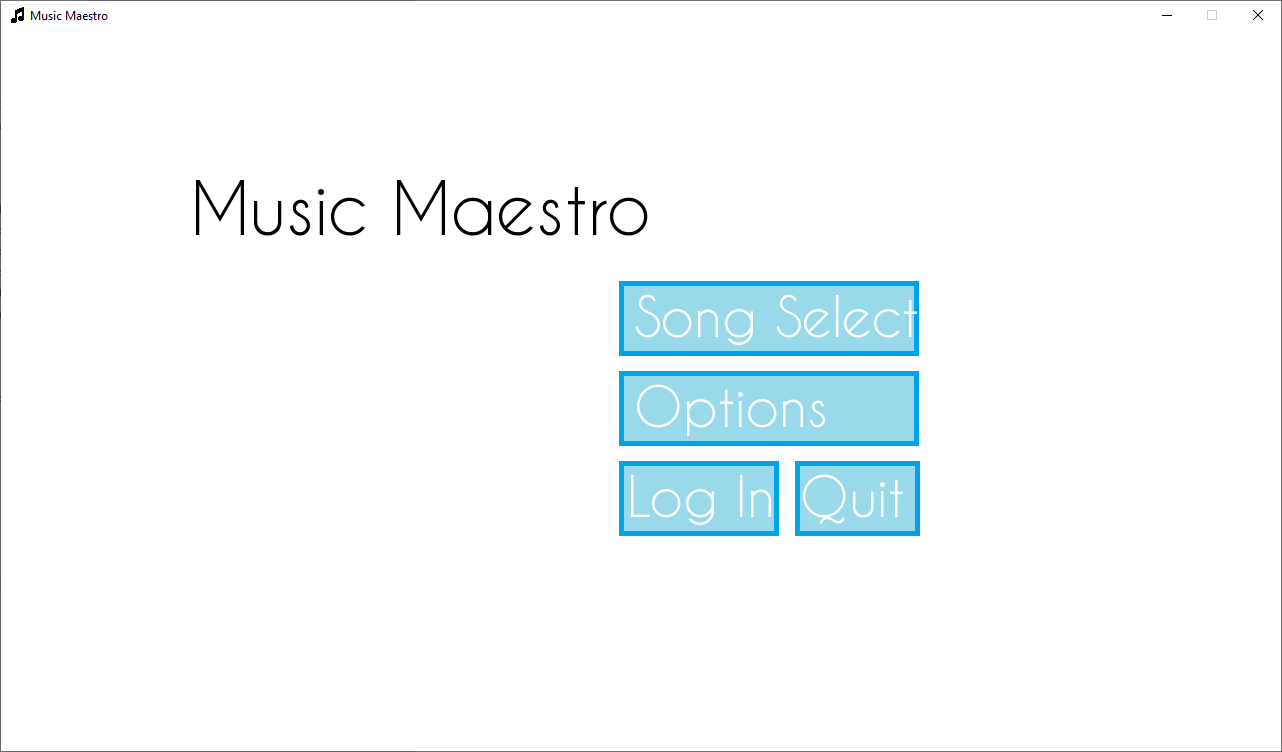
button[1]()

**return**

**for** button **in** buttons:

button[0].check(mouse\_pos)

Which gives the final result:



### Stakeholder feedback

I sent the following email to the stakeholders:

*Hello*

*Attached below the code for a functioning main menu. Please have a go and let me know of any criticisms you may have.*

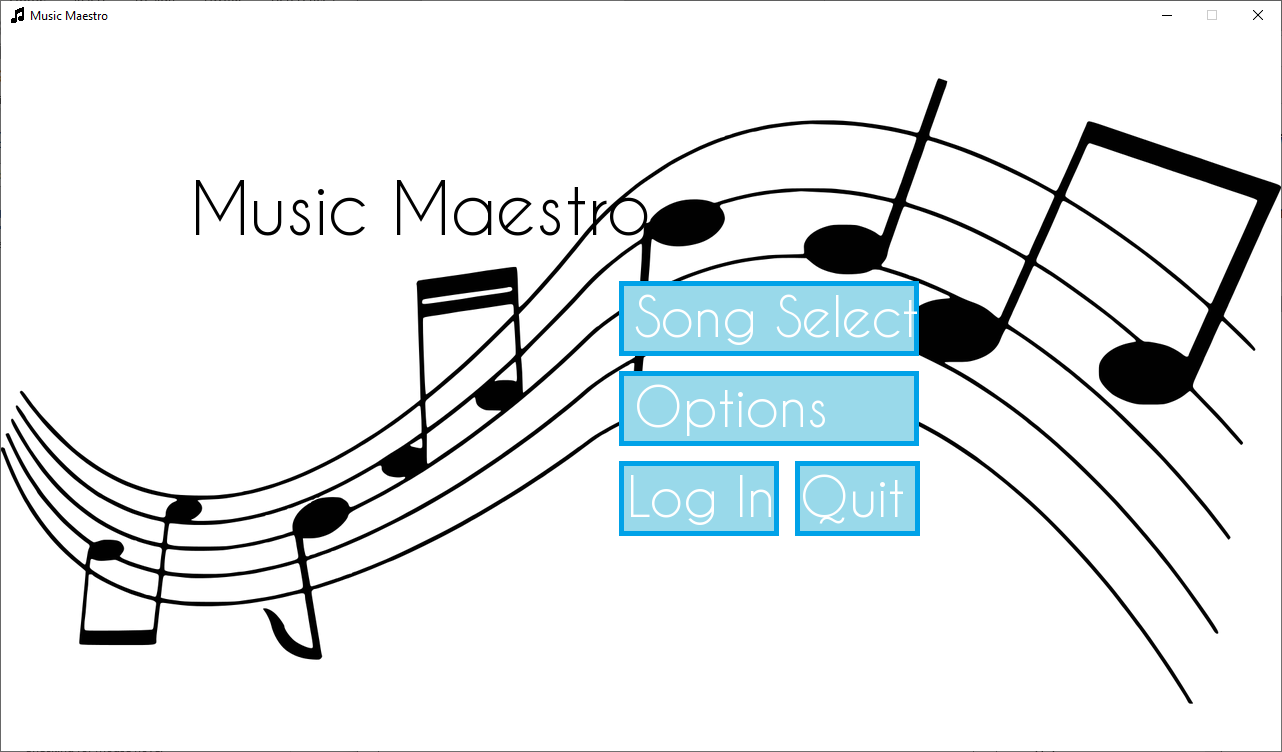
As a response, I found that the screen was very plain and uninteresting to look at. Whilst I would like to keep the program aesthetics very minimalistic, having a hint of character with a background image would not go amiss. So, before the loop, I added the following code;

# background image

background\_image = pygame.image.load('.\\assets\\images\\menu\_background.png')

self.screen.blit(background\_image, (0,40))

Which set the background image nicely:



## Creating the song select screen

The main challenge of this menu screen is the scroll bar. It is necessary for displaying the information cleanly however posed many issues. I decided to start this function with the scroll bar as once that was done the other components would follow easily.

### Coding the scroll bar

I started by making a new scroll bar class in the ui\_elements.py file. This would make for easier portability if needed as well as making debugging and testing much more efficient.

#### Initialisation

The data needed to form the scroll bar is quite simple – position, dimensions, the scroll length the thumb will need to traverse, as well as the colour of the track and thumb clicked and unclicked.

**class** ScrollBar:

**def** \_\_init\_\_(self, ctx, dimensions, position, scroll\_length, color, alt\_color, clicked\_color, scroll\_position=0):

self.ctx = ctx

self.dimensions = dimensions

self.position = position

**if** scroll\_length <= dimensions[0]: # if the scrollable area is less than the width of the scroll bar track, set to the scroll bar track width (no scrolling needed)

self.scroll\_length = dimensions[0]

**else**:

self.scroll\_length = scroll\_length

self.scroll\_position = scroll\_position

self.colors = {'primary':color ,

'alt' :alt\_color ,

'clicked':clicked\_color}

#### Track and thumb

Creating the track requires taking the position and dimensions and placing it correctly (the position coordinates are inputted as based centre of the scroll bar, which has to be accounted for).

The scroll bar thumb, or the movable part that is clicked, has a slightly more complex formula in order to be correctly sized. Using an example, if the viewable area (the data on-screen) is 40% of the entire scrollable area, then the thumb needs to be 40% the size of the track. This is done by:

Unfortunately, this was not the first formula I used. I began by using the following formula:

Which yielded correct results for small scroll lengths, such as the distance needed to travel for five song tabs, but for larger scroll distances it would become larger than the scroll bar track, making it unusable.



Scroll bar with 5 song tabs worth of scroll length.



Scroll bar upon scroll length of 25 song tabs worth of length.

The more song tabs there are, the smaller the thumb should be, not the other way around. Although the initial formula would be suitable for the program if there was a low amount of songs, I wanted it to work for any number, including if there were few enough song tabs that no scrolling was needed (in which case the thumb should be the size of the track).

self.scroll\_bar\_track = pygame.draw.rect(ctx.screen, color, pygame.Rect((**round**(position[0]-(dimensions[0]\*0.5)),**round**(position[1]-(dimensions[1]\*0.5))),(dimensions[0],dimensions[1])))

self.scroll\_bar\_thumb = pygame.draw.rect(ctx.screen, alt\_color, pygame.Rect((self.scroll\_bar\_track.x+scroll\_position,self.scroll\_bar\_track.y),(self.scroll\_bar\_track.width/(scroll\_length+1000)\*self.scroll\_bar\_track.width,self.scroll\_bar\_track.height)))

#### Rendering

Displaying the scroll bar was fairly easy, drawing two rectangles based on the scroll bar track Rect object and the thumb Rect object with the correct colours. I did this in a separate function with thumb colour as a parameter to facilitate the notch being clicked and changing colour:

**def** render(self, thumb\_color):

pygame.draw.rect(self.ctx.screen, self.colors['primary'], self.scroll\_bar\_track)

pygame.draw.rect(self.ctx.screen, thumb\_color, self.scroll\_bar\_thumb)

#### Click detection

Next was click detection and moving the thumb accordingly. I did this by keeping track of whether the thumb was clicked *last* loop in order to track how much it needs to move *this* loop. Upon initialisation of the scroll bar object, a boolean variable was\_scroll\_bar\_clicked is created as false. Then in a new function I began seeing whether the scroll bar thumb needs to be moved.

If it needed to be moved, then I utilised PyGame’s inbuilt Rect.move() function which moves Rect objects effortlessly.

**def** check(self, mouse\_pos, mouse\_clicked):

**if** self.was\_scroll\_bar\_clicked:

**if** mouse\_clicked:

self.scroll\_bar\_thumb = self.scroll\_bar\_thumb.move(mouse\_pos[0]-self.last\_mouse\_pos[0], 0)

This moves the thumb along the x-axis by the change in the mouse’s x coordinates from the last loop cycle. This will be travelling right if positive and left if negative. Then, if the thumb is not inside the track anymore (it is over as far left or right as it is supposed to go) this change is undone.

**if** self.scroll\_bar\_track.contains(self.scroll\_bar\_thumb):

self.render(self.colors['clicked'])

**else**:

self.scroll\_bar\_thumb = self.scroll\_bar\_thumb.move(-(mouse\_pos[0]-self.last\_mouse\_pos[0]), 0)

This is all done assuming the mouse is still clicked. If it is not clicked but *was* clicked last cycle, then it goes back to its default colour and was\_scroll\_bar\_clicked is set to false. If the mouse wasn’t clicked last cycle, then a check is done to see if it now is by taking the mouse position and whether the mouse is clicked.

**if** self.was\_scroll\_bar\_clicked:

**if** mouse\_clicked:

# ...

**else**:

self.render(self.colors['alt'])

self.was\_scroll\_bar\_clicked = False

**elif** self.scroll\_bar\_thumb.collidepoint(mouse\_pos) **and** mouse\_clicked:

self.was\_scroll\_bar\_clicked = True

self.last\_mouse\_pos = mouse\_pos

### Testing the scroll bar

To test the scroll bar functionality, I will populate the \_songSelectScreen function in the Application class:

**def** \_songSelectScreen(self):

# background color

self.screen.fill((255,255,255))

# title text

title = self.font['50'].render('Song select', True, (0,0,0))

self.screen.blit(title, (490,36))

# ui elements

back\_button = Button(self, text='Back', text\_size=22, position=[128,648], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

scroll\_bar = ScrollBar(self, dimensions=[1000,20], position=[640,504], scroll\_length=10000, color=(153,217,234), alt\_color=(0,162,232), clicked\_color=(0,131,187))

# event loop

**while** True:

mouse\_pos = pygame.mouse.get\_pos()

mouse\_clicked = pygame.mouse.get\_pressed()[0]

scroll\_bar.check(mouse\_pos, mouse\_clicked)

back\_button.check(mouse\_pos)

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

**if** event.**type** == MOUSEBUTTONDOWN:

**if** back\_button.check(mouse\_pos):

**return** self.\_menuScreen()

pygame.display.flip()

**return**

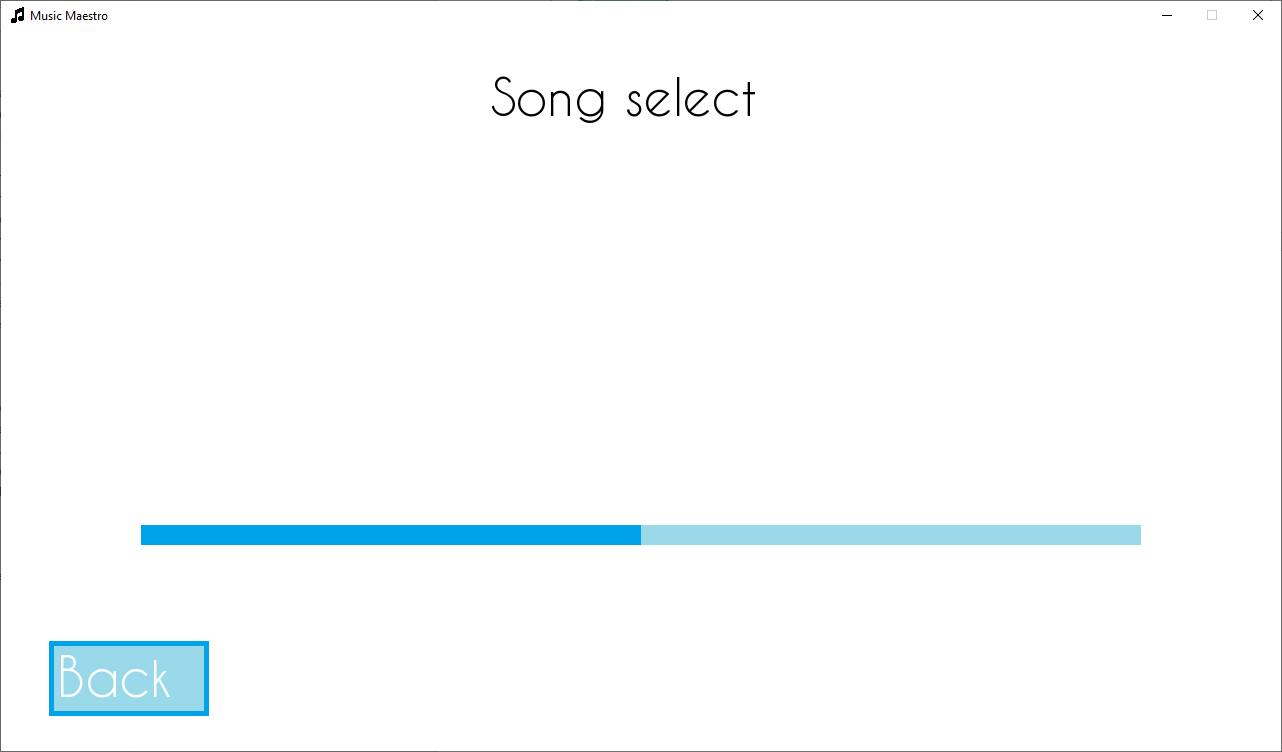
This will create a scroll bar of width 1000 and scroll length of 10000. The notch should be an eleventh (1000 pixels on screen and 10000 off-screen that can be scrolled to) of the track length and move fluidly across it.



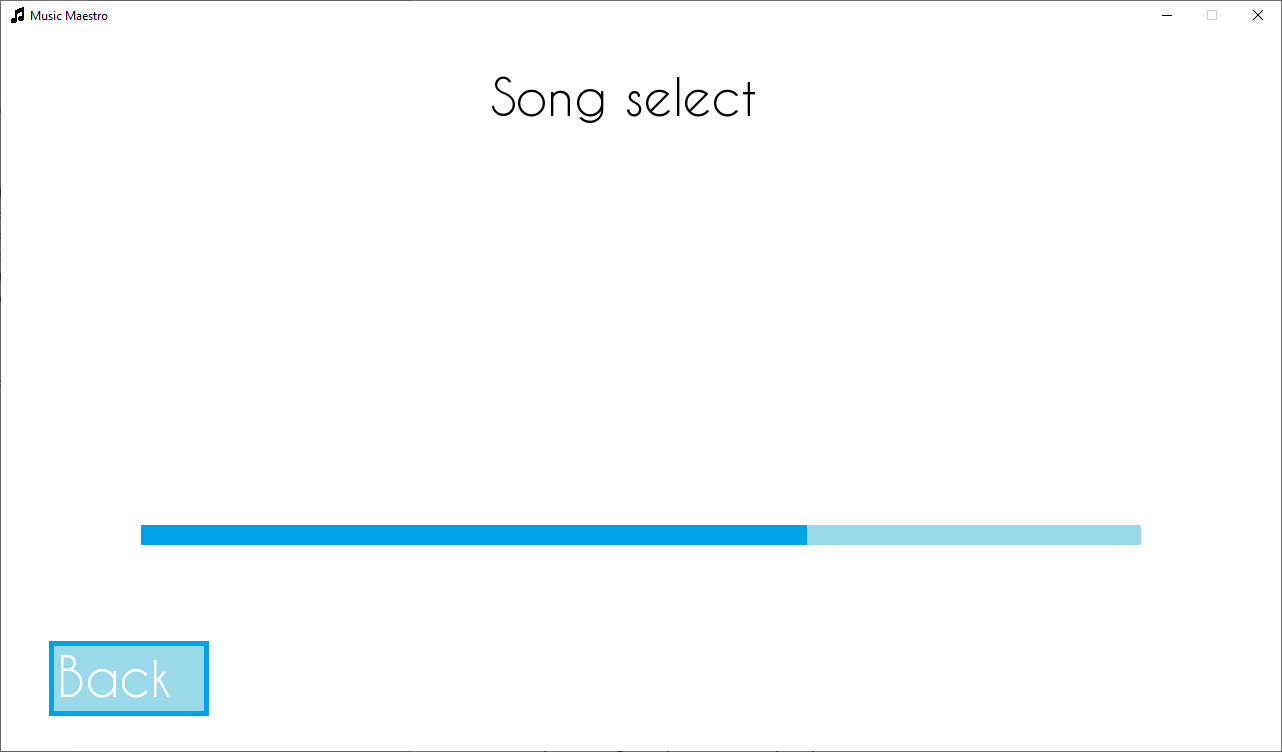
This is done as intended, with the scroll bar moving smoothly. To make sure the thumb size is definitely correct, I will also test:

Scroll length = 1000

Thumb should be half the track length:

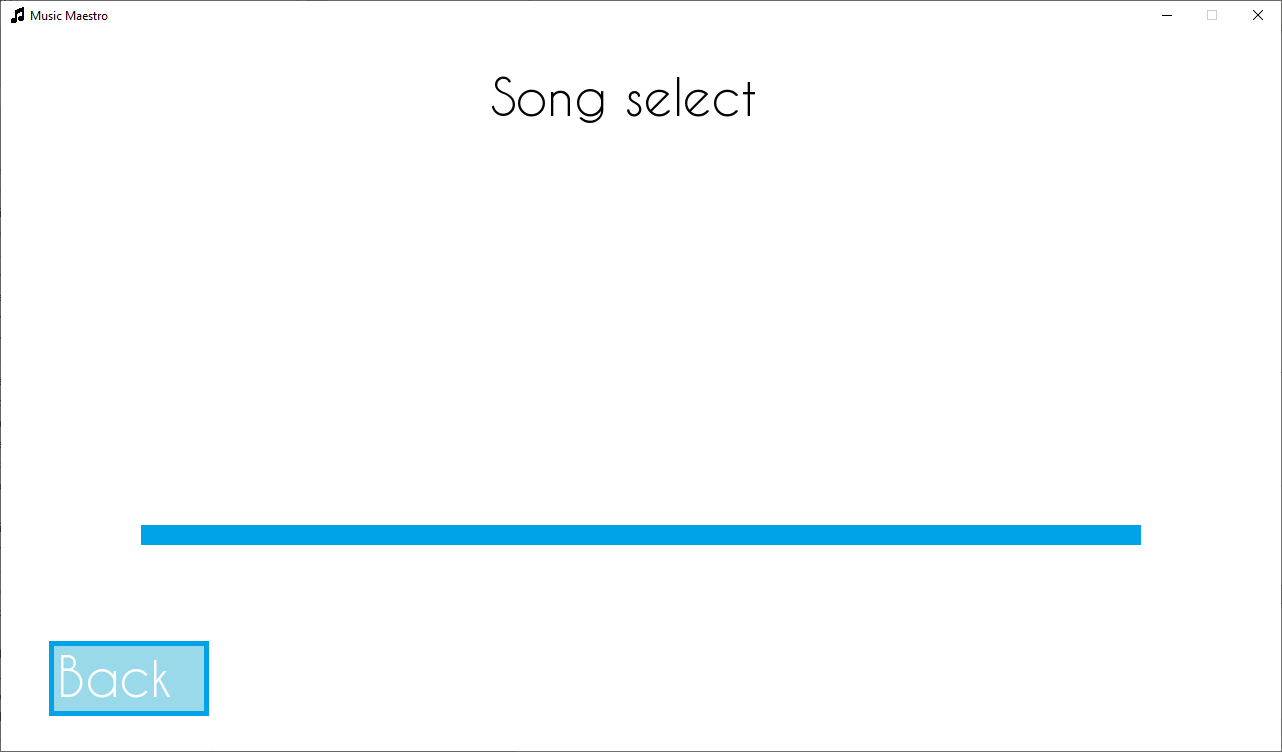


Scroll length = 500

Thumb should be two thirds of the track length: 

Scroll length = 0

Thumb should be the size of the track



### Coding the song tabs

Now the scroll bar was complete, I added in the song tabs. These were fairly straight forward to add in – being comprised of only text and a button. I again created a class in the ui\_elements.py file to allow for e

#### Initialisation

The parameters required for the song tab are the start position of the tab (used for scrolling calculations) and the song it is representing.

A button object is created for each song tab.

**class** SongTab:

**def** \_\_init\_\_(self, ctx, start\_pos, song=['name','difficulty','location']):

self.ctx = ctx

self.start\_pos = start\_pos

self.position = start\_pos

self.song = song

self.button = Button(self.ctx, text='play', text\_size=22, position=[self.position+125,400], dimensions=[160,75], color=(213,240,247), alt\_color=(0,162,232), hover\_color=(58,186,218), text\_color=(0,0,0))

#### Rendering

The tab is rendered as two rectangles (one five pixels smaller to create a border) and then text indicating the song name and difficulty. I will later add the high score when the database is set up.

**def** render(self):

pygame.draw.rect(self.ctx.screen, (0,162,232), (self.position, 115, 250, 350))

pygame.draw.rect(self.ctx.screen, (153,217,234), (self.position+5, 120, 240, 340)) # 5 pixels smaller to create a border

font\_40 = pygame.font.Font('.\\assets\\font.ttf', 40) # size 40 text

font\_20 = pygame.font.Font('.\\assets\\font.ttf', 20) # size 20 text

name = font\_40.render(self.song[0], True, (0,0,0))

diff = font\_20.render(self.song[1], True, (0,0,0))

self.ctx.screen.blit(name, (self.position+10,120))

self.ctx.screen.blit(diff, (self.position+10,170))

This creates a tab that looks like this:



#### Tab movement

To move the tab, I created the set\_x function which changes the position of the tab by its start position added to a modifier (x). The button position also needs to be updated, using the set\_position function in the button class.

**def** set\_x(self, x):

self.button.set\_position([self.start\_pos+125+x,400])

self.position = self.start\_pos + x

### Coding the load files function

The purpose of this sub-routine is to load all files related to the program ready for use in a dedicated dictionary structure. This will be very useful for displaying musical notation during the user performance and loading all the song files into a convenient location.

**def** \_loadFiles(self, t=['png','jpg']):

dir\_contents = [[files,root] **for** root, dirs, files **in** os.walk('.')]

The sub-routine is based entirely around os.walk(), which traverses all the child directories from the argument directory down, returning a generator containing all files and directories inside. This is very useful as then after narrowing down the results to just the whitelisted extension files using a nested and enumerated loop the details of the file can then be added to a dictionary ready for use throughout the Application instance.

The benefit of using os.walk is that as long as the files are in the same directory (or child of it) as the main Python file, denoted by the ‘.’ argument, they will be identified and loaded. Furthermore, new files can be added into the directory or old ones removed or modified and the code will not need any changes – it is fully self-sufficient.

The parameter t signifies all the file extensions to be whitelisted, aka the file extensions that are wanted.

Result of dir\_contents:

[[['main.pyw', 'readme.md'], '.'],

[['buffer.wav', 'font.ttf'], '.\\assets'],

[['icon.png', 'menu\_background.png'], '.\\assets\\images'],

[['Megolovania - Hard.txt', 'Ode to Joy - Easy.txt', 'Scales - Medium.txt', 'Short n Sweet - Easy.txt', "Up 'n Down - Medium.txt"], '.\\assets\\songs'],

[['.audio.py.swp', 'audio.py', 'song.py', 'ui\_elements.py', 'user.py'], '.\\scripts'],

[['audio.cpython-36.pyc', 'audio.cpython-37.pyc', 'ui\_elements.cpython-36.pyc', 'ui\_elements.cpython-37.pyc', '\_\_init\_\_.cpython-37.pyc'], '.\\scripts\\\_\_pycache\_\_']]

Then, this list is iterated through with two for loops – one for each directory and one for each file in each directory:

location = []

**for** x,dirs **in** **enumerate**(dir\_contents):

**for** y,**file** **in** **enumerate**(dirs[0]):

**file** = **file**.split('.')

**if** **file**[1] **in** t:

path = dirs[1] + '\\' + dir\_contents[x][0][y]

location.append([path,**file**[0]])

The enumerate function creates an extra variable during the iteration through the list, x, which acts as a counter. Each iteration, starting from 0, it will increase by 1.

For each file, it is split into two components – it’s filename and file extension: 'picture.jpg' becomes ['picture' , 'jpg']. Then, if its extension is whitelisted, it’s path and filename is added to the location list (of which is returned at the end of the function).

Now this function has been implemented, image loading can be made easier. In the initialisation of Application, a dictionary of all the images can be created:

self.images = {}

images = self.\_loadFiles()

**for** image **in** images:

self.images[image[1]] = pygame.image.load(image[0])

Then, to access an image, for example with the main menu background, the code can be changed to:

# background image

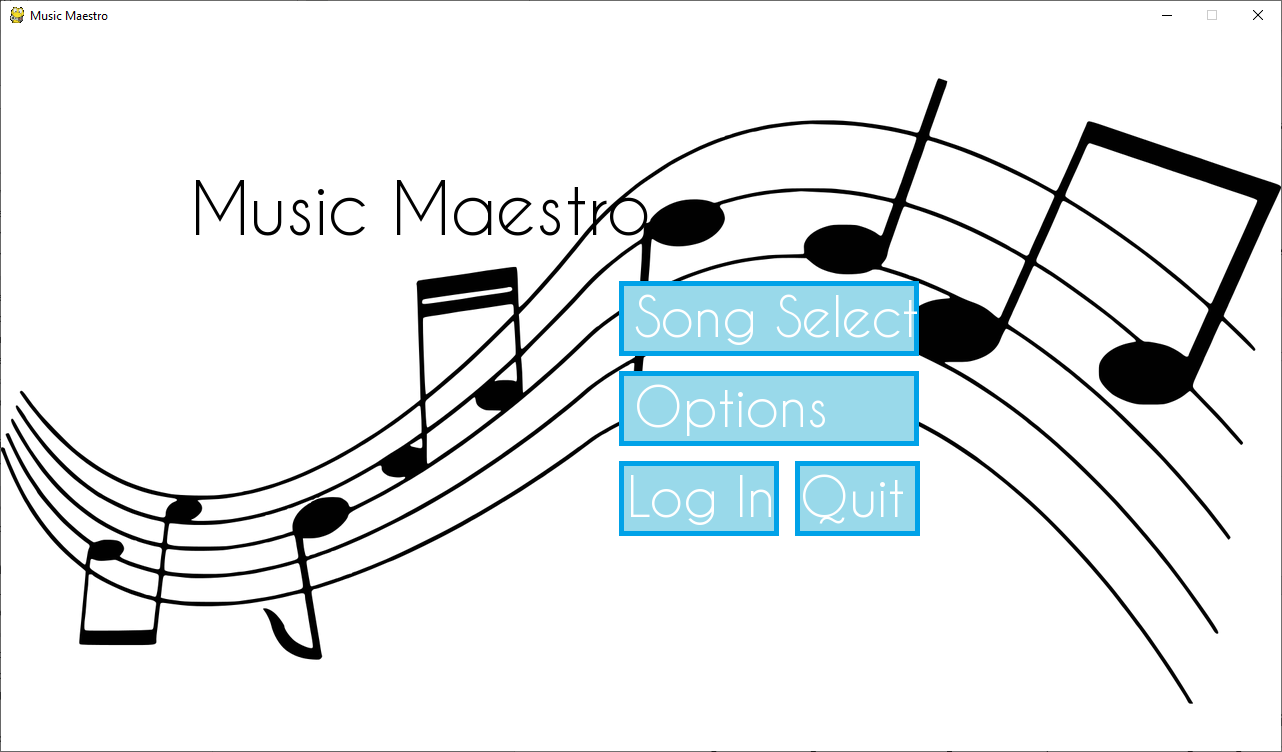
background\_image = self.images['menu\_background']

self.screen.blit(background\_image, (0,40))

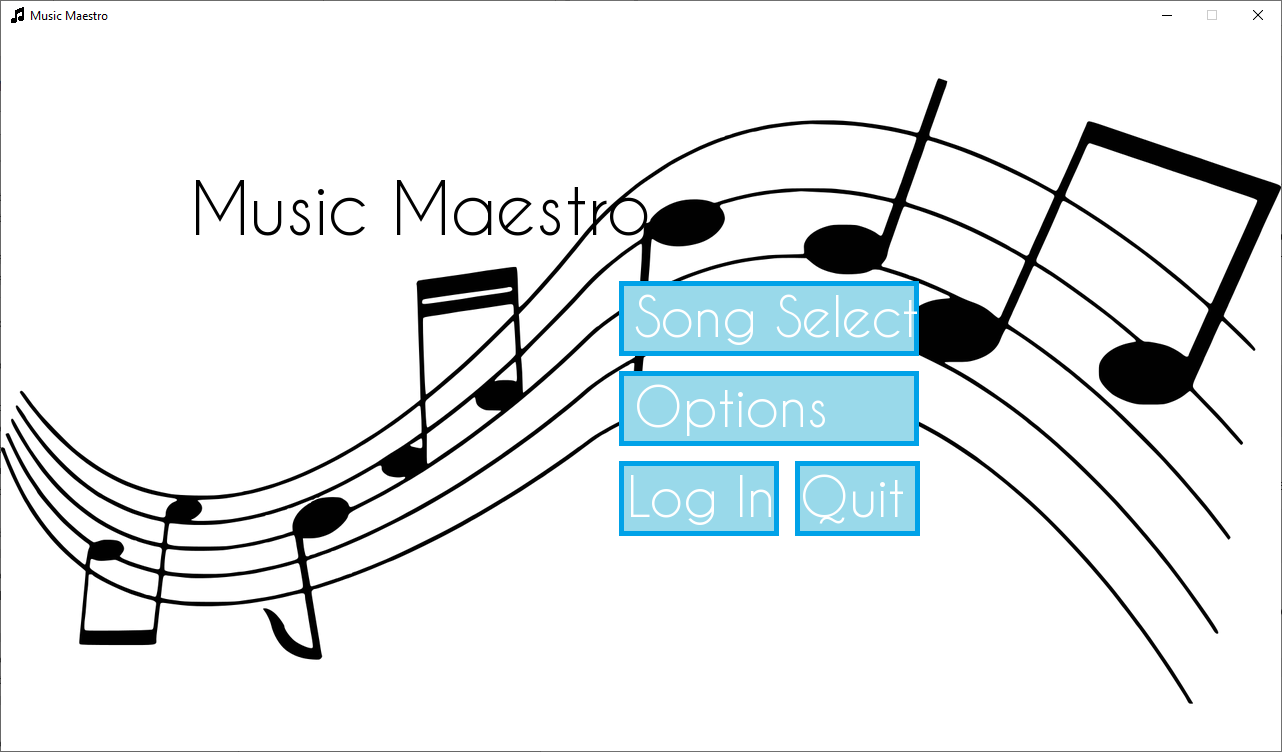
At this point I added an icon for the program window, i.e. the picture displayed in the top left of the window:

pygame.display.set\_icon(self.images['icon'])

Default:

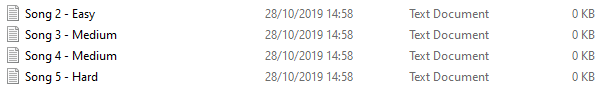


New:



### Loading song files

The songs will be stored as text files in the program folder and will have the name format of “name – difficulty”.



This is then used, before the loop, to split the song files into name and difficulty in one list:

songs = self.\_loadFiles(t=['txt'])

**for** song **in** songs:

song[1] = song[1].split(' - ')

This creates a list with each element in the format:

['.\\assets\\songs\\Song 3 - Medium.txt', ['Song 3', 'Medium']]

Then, a list of all the song tabs is made. The starting position is calculated by taking the width of the tab (250 pixels) added to the gap between each tab (25 pixels) and multiplying that by the place in the songs list. Using the enumerate function it is very easy to do this as it cycles through the list with a counter (which I start at 1 rather than the default of 0 so the first tab isn’t placed at x=0 on screen).

# ui elements

song\_tabs = []

**for** place,song **in** **enumerate**(songs,start=1):

song\_tabs.append(SongTab(self, start\_pos=**round**(place\*(250+25))-135, song=[songs[place-1][1][0],songs[place-1][1][1],songs[place-1][0]]))

Also created is the back button which would take the user back to the main menu, and the scroll bar which has a scroll length of where 275 is the number of pixels a tab and the spacing take up and 1025 is 1000 pixels for the already viewable area and 25 pixels for the last tab’s unneeded space at the end.

back\_button = Button(self, text='Back', text\_size=22, position=[128,648], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

scroll\_bar = ScrollBar(self, dimensions=[1000,20], position=[640,504], scroll\_length=**round**(275\***len**(song\_tabs))-1025, color=(153,217,234), alt\_color=(0,162,232), clicked\_color=(0,131,187))

Taking nothing off scroll length:



Taking 1000 off the scroll length:



Taking 1025 off the scroll length:



### Coding the main loop

Creating the loop for the screen will consist of four parts. The first part will be getting the mouse position and whether it is clicked or not, as well as refreshing the parts of the screen which move (by drawing a white rectangle over the area). Then, each UI element will be rendered using their check functions, making sure that they are displaying the correct colour depending on if they are being hovered over or not. Each song tab will be repositioned based on the scroll bar’s position however only the tabs that are in the viewable area will be rendered and checked to keep performance of the program high with a large number of songs.

# event loop

**while** True:

mouse\_pos = pygame.mouse.get\_pos()

mouse\_clicked = pygame.mouse.get\_pressed()[0]

pygame.draw.rect(self.screen, (255,255,255), (0,92,1280,400))

scroll\_bar.check(mouse\_pos, mouse\_clicked)

back\_button.check(mouse\_pos)

**for** tab **in** song\_tabs:

tab.set\_x(scroll\_bar.get\_notch\_position())

**if** tab.get\_x() > -275 **and** tab.get\_x() < 1280:

tab.render()

tab.button.check(mouse\_pos)

Then, each button will be checked to see if it has been clicked on and if so, run the corresponding function. This includes each tab button, the back button and the quit button. Finally, two white rectangles will be drawn either side of the viewable area between the sides of the window and the scroll bar to hide the tabs that have gone past the viewable area:

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

**if** event.**type** == MOUSEBUTTONDOWN:

**if** back\_button.check(mouse\_pos):

**return** self.\_menuScreen()

**for** tab **in** song\_tabs:

**if** tab.button.check(mouse\_pos):

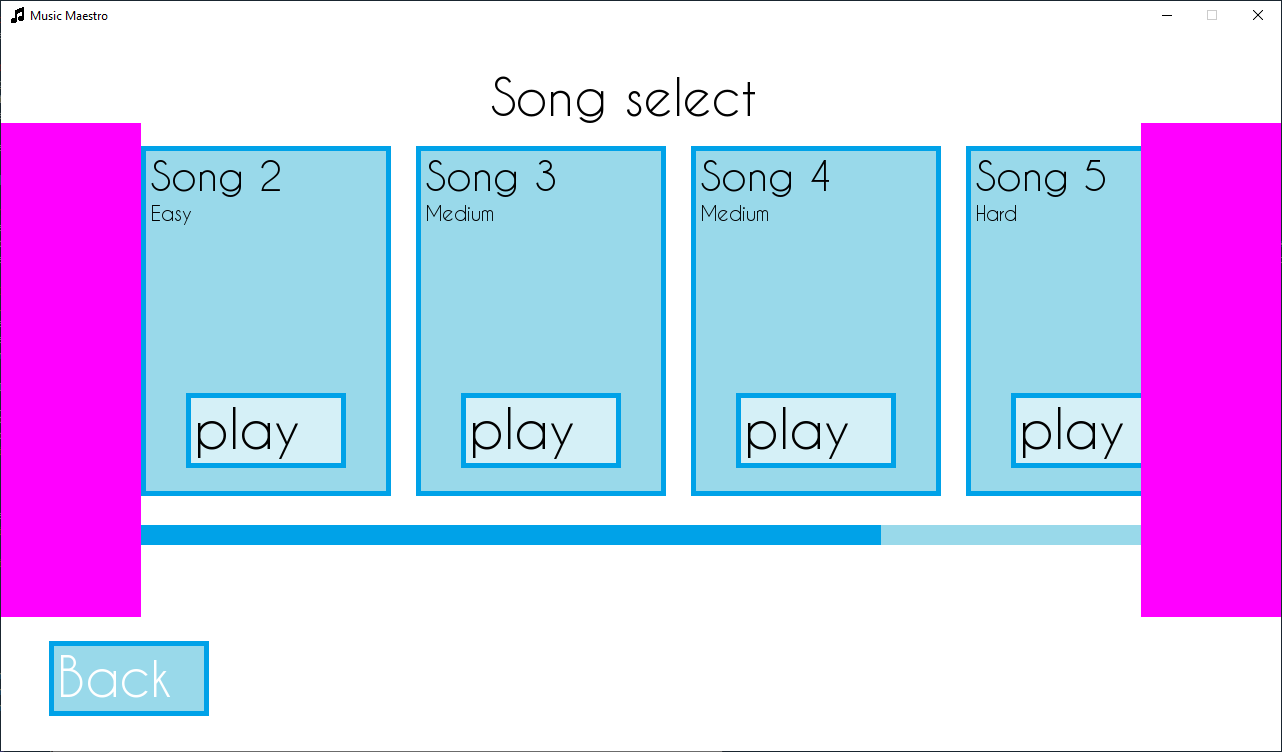
**return** self.\_gameScreen(tab.song)

pygame.draw.rect(self.screen, (255,255,255,255), (0,115,140,365))

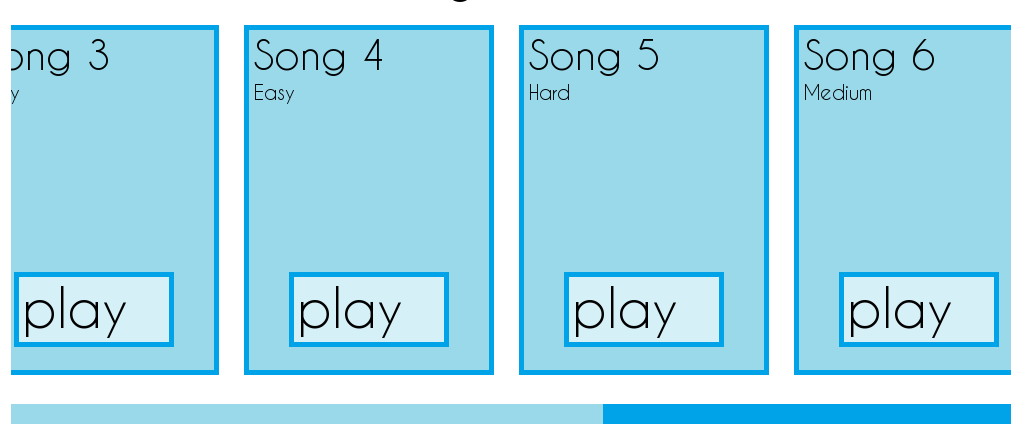
pygame.draw.rect(self.screen, (255,255,255,255), (1140,115,1280,365))

pygame.display.flip()

Which produces:



A logical error I discovered fairly quickly was that the scroll bar would not scroll all the way through the tabs; instead reaching its end too promptly.



self.scroll\_bar\_thumb = self.scroll\_bar\_thumb.move(mouse\_pos[0]-self.last\_mouse\_pos[0], 0)

self.scroll\_position -= (mouse\_pos[0]-self.last\_mouse\_pos[0])

Out of 9 songs, only 6 are displayed before the thumb reaches the end. This is due to the scroll bar position variable being changed at the same rate as the change in mouse position. As there are more pixels in the total scrollable area than there are in the scroll bar track, a relation between how many pixels the thumb has moved and how much to scroll by must be created (rather than a 1:1 ratio).

The formula for how many pixels to scroll by per pixel the thumb has moved is as follows:

Or in other words, the scrolling length divided by the number of pixels the scroll bar *can* travel.

**try**:

self.scroll\_amount = scroll\_length/((self.scroll\_bar\_track.right-self.scroll\_bar\_thumb.width)-self.scroll\_bar\_track.x)

**except** ZeroDivisionError:

self.scroll\_amount = 0

The division by zero error that occurs when the scroll length is zero is caught.

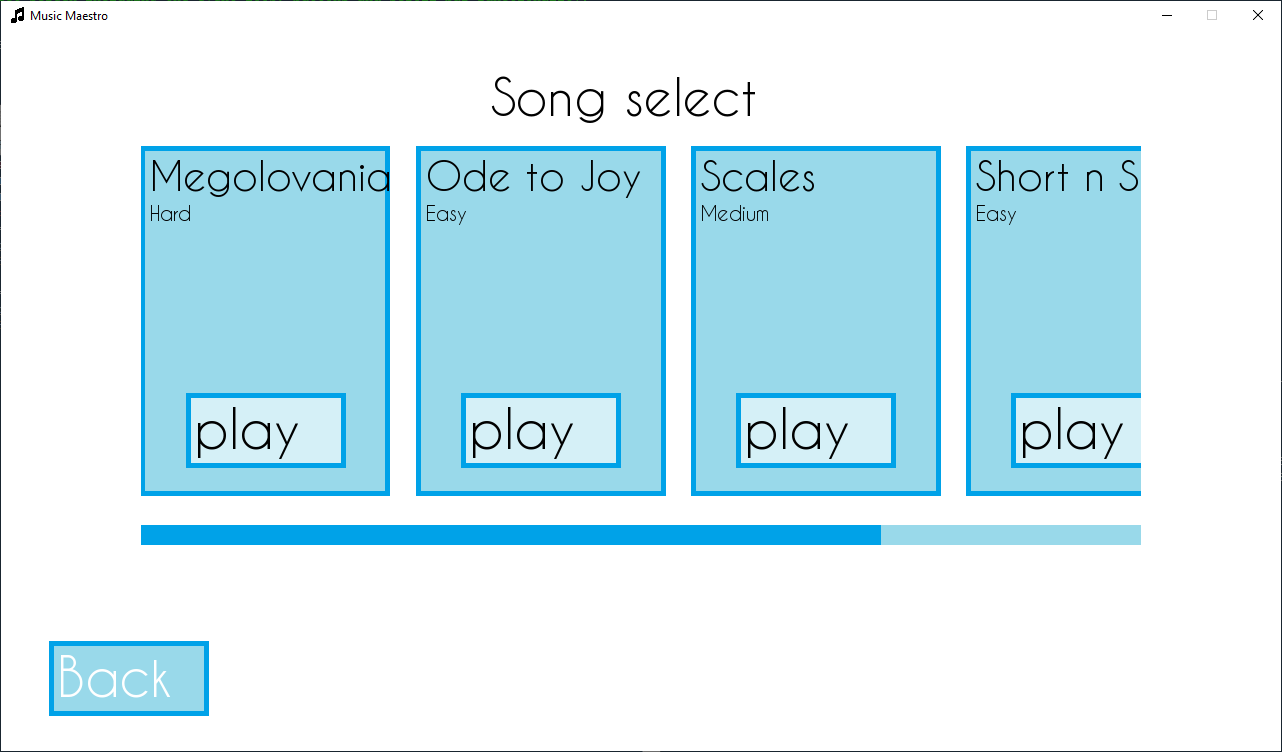
Now, this scroll amount can be multiplied with the change in mouse x to find how many pixels to scroll by.

self.scroll\_position -= self.scroll\_amount\*(mouse\_pos[0]-self.last\_mouse\_pos[0])

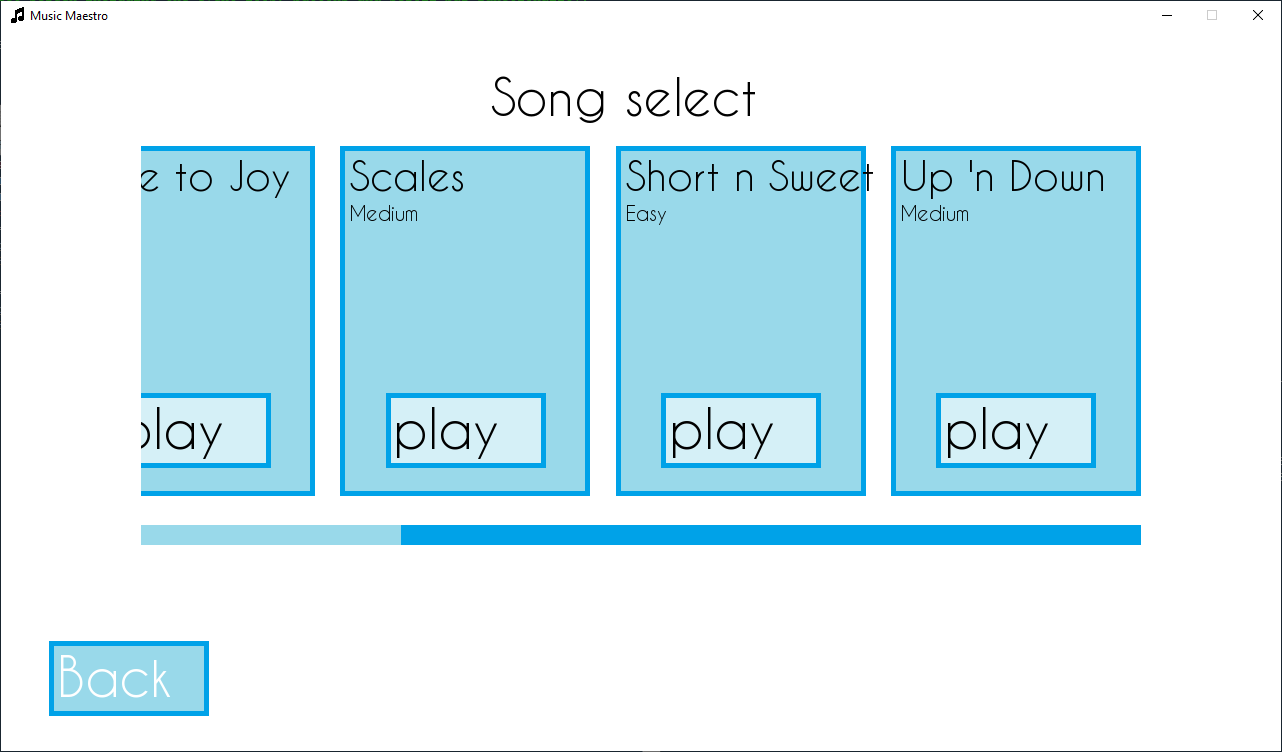
This fixes the issue, meaning any number of song tabs can be scrolled to correctly.

### Testing the song selection screen

Upon entering the song selection screen, all components are there and drawn correctly:



Upon clicking the scroll bar and dragging the thumb, the tabs move correctly, and all tabs can be displayed fully:



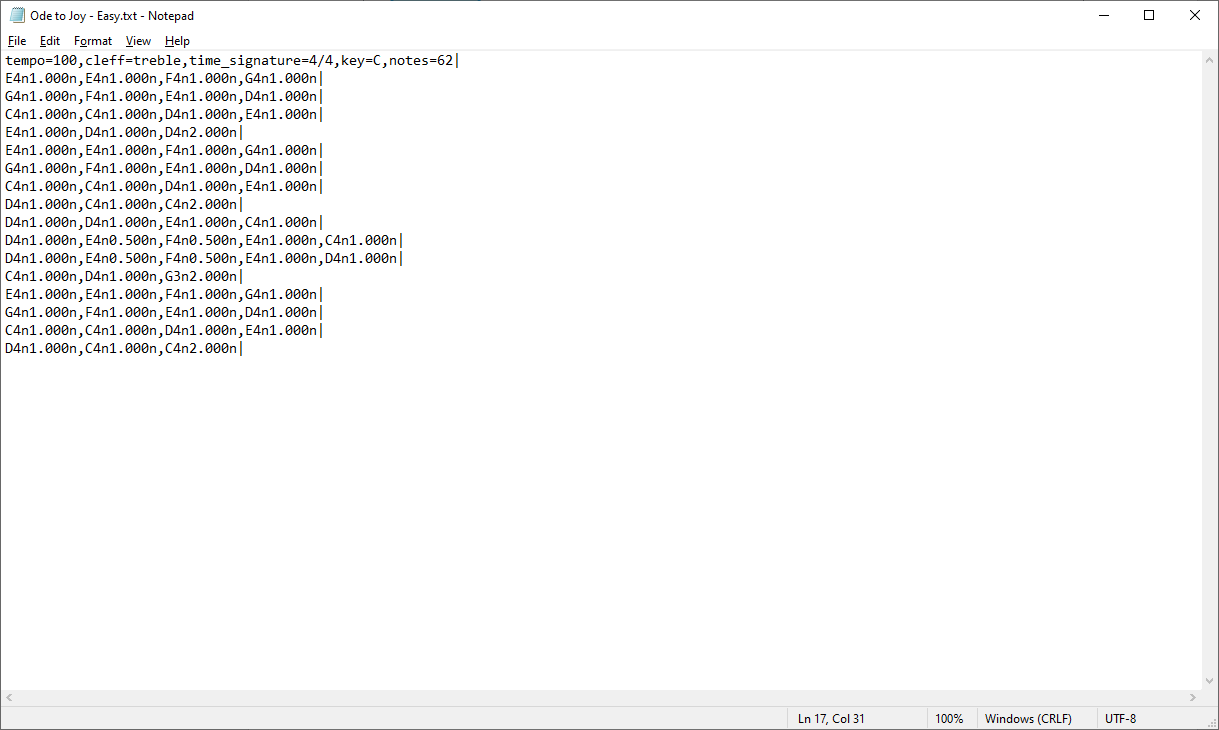
The song tab buttons are all functional and lead to the correct location with the correct data.

Finally, the back button takes the user back to the main menu as expected.

## Creating the performance screen

### Coding the song parser

The song parser will be a class in its own song.py file. Each song file is formatted the same:



With the first line being the metadata of the song. Each note is:

tone + octave + flat/sharp/neutral + duration + rest/note

And each bar is separated by a ‘|’.

#### Initialisation

The SongParser class takes two arguments, song and images. Song is a list containing all the information in the song file;

[['tempo=100', 'cleff=treble', 'time\_signature=4/4', 'key=C', 'notes=62'],

['E4n1.000n', 'E4n1.000n', 'F4n1.000n', 'G4n1.000n'],

['G4n1.000n', 'F4n1.000n', 'E4n1.000n', 'D4n1.000n'],

['C4n1.000n', 'C4n1.000n', 'D4n1.000n', 'E4n1.000n'],

...

['C4n1.000n', 'C4n1.000n', 'D4n1.000n', 'E4n1.000n'],

['D4n1.000n', 'C4n1.000n', 'C4n2.000n'],

['']]

And images is the dictionary of all images the program uses. Each note, rest and tilt values (sharp/flat) have their own image:

  
To start with, the metadata and empty entry at the end of the song list must be removed. The SongParser object will have a next\_note function that is called each time the next note is required. To provide this, the note will be popped off the list. For this reason, the entire list must be reversed. To start, each bar in the list is reversed:

**class** SongParser:

**def** \_\_init\_\_(self, song, images):

self.song = song[1:-1][::-1]

And then each bar of notes is reversed as well so the stack can be popped and get the correct note order:

**for** i,bar **in** **enumerate**(self.song):

self.song[i] = bar[::-1]

Next, three dictionaries containing all the data needed to translate a note string into an image and details are created:

self.y\_pos = {i:**int**(count\*20+60) **for** count,i **in** **enumerate**([y+**str**(x) **for** x **in** **range**(7,0,-1) **for** y **in** ['G','F','E','D','C','B','A']])}

self.tilt = {'#':images['sharp'],

'b':images['flat'],

'n':None}

self.durations = {0.25:'sixteenth\_',

0.5:'eighth\_',

1.0:'quarter\_',

2.0:'half\_',

4.0:'whole\_'}

self.y\_pos can be broken down into:

notes = []

**for** x **in** **range**(7,0,-1):

**for** y **in** ['G','F','E','D','C','B','A']:

notes.append(y+**str**(x))

self.y\_pos = {}

**for** count,i **in** **enumerate**(notes):

self.y\_pos[i] = **int**(count\*20+60)

In which a full list of notes is created in descending order;

['G7', 'F7', 'E7', ... , 'D1', 'C1', 'B1', 'A1']

And then iterated through as the key of the self.y\_pos dictionary entry, along with the corresponding on-screen y-position:

y\_pos = **int**(count \* (0.5\*(distance between stave lines)) + (height of note image **file**))

#### Parsing

The \_parse function has a string note as a parameter. This will look something like:

'E4n1.000n'

The image dictionary will contain all the information needed about the note and will be created with its x-position at the far right of the Application window, x=1280.

**def** \_parse(self, note):

image = {'pos':1280}

Its duration is derived from the 4th character to the 8th in the string:

image['note\_length'] = **float**(note[3:-1]) = 1.000

The note name will be characters one to three:

image['note\_name'] = note[:3] = 'E4n'

The number of loop cycles the microphone detected note must match before the note is successfully played:

image['played'] = 5

This will help smooth out any errors or inconsistencies in the detected note and help prevent the note from being successfully played by chance when it shouldn’t have.

The duration, y\_pos and tilt are all looked up into separate variables:

duration = self.durations[image['note\_length']]

y\_pos = self.y\_pos[note[:2]]

tilt = self.tilt[note[2]]

Next, the correct image can be found using the duration value:

**if** note[-1] == 'n':

image['note\_img\_offset'] = 118

image['note\_img'] = [self.images[duration+'note'],y\_pos-image['note\_img\_offset']]

**elif** note[-1] == 'r':

image['note\_img'] = [self.images[duration+'rest'],350]

note\_img\_offset is to take into consideration the height of the image file when placing the image on the stave at the correct y-position.

Finally, the tilt image can be added if it is needed before returning the image dictionary:

**if** tilt:

image['tilt'] = [tilt,y\_pos-115]

#### Getting the next note

To return the next note of the song on the function call, the current bar being popped from must be tracked.

self.current\_bar = self.song.pop()

Is performed during the initialisation of the instance, removing the first bar of the song from self.song and setting self.current\_bar equal to it. This means self.current\_bar will look something like:

self.current\_bar = ['E4n1.000n', 'E4n1.000n', 'F4n1.000n', 'G4n1.000n']

Then, the next note is parsed and returned for use by the main program.

self.note = self.\_parse(self.current\_bar.pop())

**return** self.note

However, as .pop() removes the result from the list, eventually the bar will be empty, and the next bar should be popped into self.current\_bar.

**def** next\_note(self):

**if** **not** self.current\_bar:

**if** self.song:

self.current\_bar = self.song.pop()

**else**:

**return** None

When there are no more bars left in the song, and no more notes left in the last bar, “if self.song” will evaluate as False, and a special case of “None” is returned to let the main program know there are no more notes left in the song.

### Testing the song parser

To test the song parser, first a loop will be created to display the string value of the notes in the song “Ode to Joy”:

[['tempo=100', 'cleff=treble', 'time\_signature=4/4', 'key=C', 'notes=62'],

['E4n1.000n', 'E4n1.000n', 'F4n1.000n', 'G4n1.000n'],

['G4n1.000n', 'F4n1.000n', 'E4n1.000n', 'D4n1.000n'],

['C4n1.000n', 'C4n1.000n', 'D4n1.000n', 'E4n1.000n'],

...

['C4n1.000n', 'C4n1.000n', 'D4n1.000n', 'E4n1.000n'],

['D4n1.000n', 'C4n1.000n', 'C4n2.000n'],

['']]

Using the following driver code, each note is expected to be displayed in the correct order:

**with** **open**('..\\assets\\songs\\Ode to Joy - Easy.txt', mode='r') **as** **File**:

song\_contents = **File**.read().split('|') # Read the song file into a list

**for** i,bar **in** **enumerate**(song\_contents):

song\_contents[i] = bar.strip('\n').split(',')

Song = SongParser(song\_contents, {'images here'})

note = Song.next\_note()

**while** note:

note = Song.next\_note()

**print**(note)

This produces:

E4n1.000n

F4n1.000n

G4n1.000n

G4n1.000n

F4n1.000n

E4n1.000n

D4n1.000n

C4n1.000n

C4n1.000n

D4n1.000n

E4n1.000n

…

E4n1.000n

D4n1.000n

C4n1.000n

C4n2.000n

None

To test to make sure a note is parsed correctly, a manual call can be done on the \_parse function:

Song = SongParser(['song here'],{'images here'})

Song.\_parse('C4#2.000n')

Which returns:

{'pos':1280,

'note\_length':2.0,

'note\_name':'C4#',

'played':5,

'note\_img\_offset':118,

'note\_img':[<Surface(52x138x32 SW)>, 442],

'tilt':[<Surface(26x138x32 SW)>, 445]}

### Coding variable declaration

To make the code more digestible and easier to understand I have separated key parts of the performance function into their own sub-routines. Due to this, certain variables will be made global to avoid long function returns and arguments.

**def** get\_song\_vars(song):

**global** Song, score, tempo, cleff, time\_signature, key, song\_length

**global** metronome, current\_note, current\_mic\_note, tick, event, beats, note\_buffer, note\_played\_data

The first set of variables relate to the information of the song, and the second set to do with the backend handling of the performance.

To start with, the song text file will be read into a variable song\_contents, and split into individual components:

**with** **open**(song[2], mode='r') **as** **File**:

song\_contents = **File**.read().split('|')

This will change the string value of the text file into a list of each bar (which in the text file was denoted with a “|”).

This will change:

'A5n1.000n,B5n1.000n,C5#1.000n,D5n1.000n|E5n1.000n,F5#1.000n,G5#1.000n,A6n1.000n|'

To:

['A5n1.000n,B5n1.000n,C5#1.000n,D5n1.000n','E5n1.000n,F5#1.000n,G5#1.000n,A6n1.000n']

Then, each item in the created list will be again separated into a list of individual notes, representing a bar in the music.

**for** i,bar **in** **enumerate**(song\_contents):

song\_contents[i] = bar.strip('\n').split(',')

This will give the final result:

[['A5n1.000n','B5n1.000n','C5#1.000n','D5n1.000n'],

['E5n1.000n','F5#1.000n','G5#1.000n','A6n1.000n']]

This can then be placed as the argument for a SongParser object to allow for the song to be played. The first item in song\_contents contains all the metadata about the song including: tempo, clef, time signature, key and song length (number of notes in the song).

To obtain these values, simple string manipulation can be done to convert each string into a list, with the second value in the list being the desired one:

Song = SongParser(song\_contents, self.images)

# song\_contents[0] = ['tempo=90', 'cleff=treble', 'time\_signature=4/4', 'key=C', 'notes=32']

tempo = **int**(song\_contents[0][0].split('=')[1])

# 'tempo=90' to ['tempo', '90'] to '90' to 90

cleff = self.images[song\_contents[0][1].split('=')[1]+'\_cleff']

# 'cleff=treble' to ['cleff', 'treble'] to 'treble'

time\_signature = song\_contents[0][2].split('=')[1].split('/')

# 'time\_signature=4/4' to ['time\_signature', '4/4'] to '4/4' to ['4', '4']

key = song\_contents[0][3].split('=')[1]

# 'key=C' to ['key', 'C'] to 'C'

song\_length = **int**(song\_contents[0][4].split('=')[1])

# 'notes=32' to ['notes', '32'] to '32' to 32

Then, key is changed to the image of the key the song is in:

**if** key **not** **in** ['C','Am']:

key = self.images[key+'\_key\_signature']

The key of C and Am have no special notation, so the image is only loaded if it is not C or Am.

Then finally the necessary variables needed for running the performance are declared:

metronome = 'left'

current\_mic\_note = 'X'

current\_note = 'Y'

score = 0

tick = 0

event = None

beats = 0

note\_buffer = [Song.next\_note()]

note\_played\_data = []

current\_mic\_note and current\_note have placeholder values that cannot physically be achieved by the microphone or played during the song as they are not musical notes.

### Coding background rendering

The background will consist of four main components; text showing the song name and score, note hitbox indicating when a note is expected to be played, note fadeout gradient so notes that were not played gradually fade to white as they leave the stave to the left, and the musical components that will stay fixed, such as the stave, metronome, key and time signature and clef.

#### Text

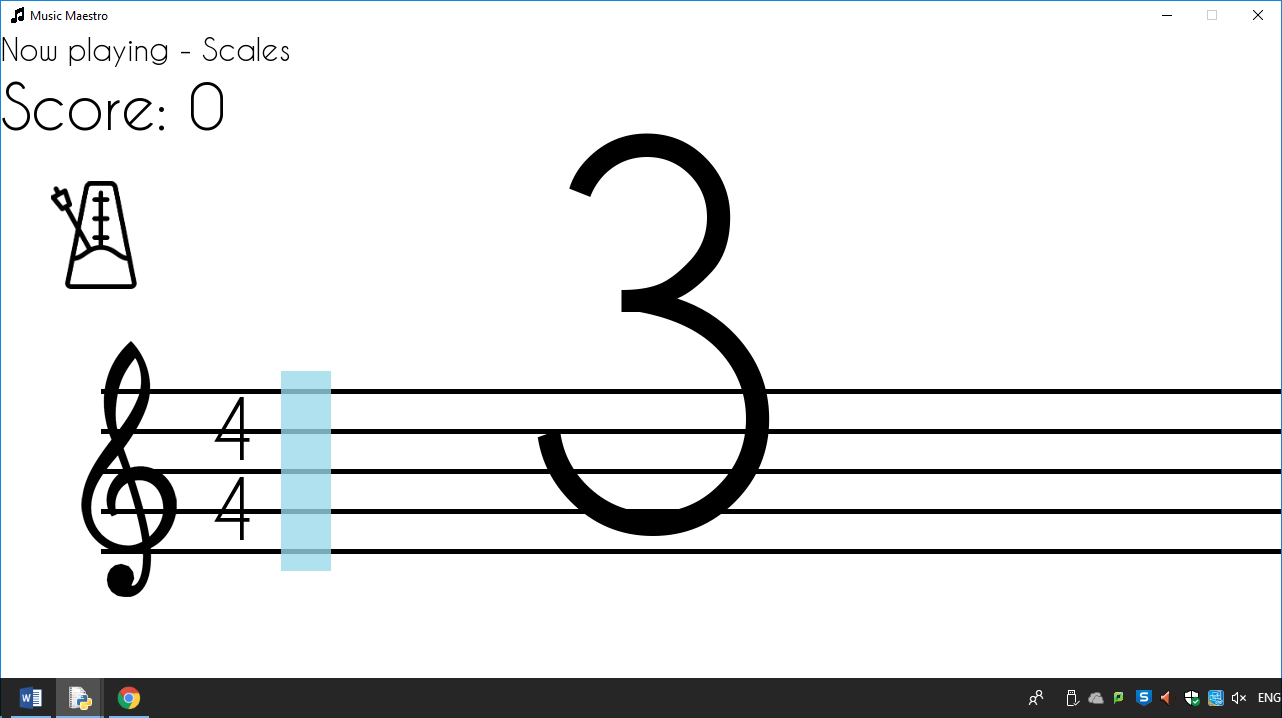
title = self.font['30'].render('Now playing - {}'.format(song[0]), True, (0,0,0)) # Render the "Now playing" text

score\_text = self.font['60'].render('Score: {}'.format(score), True, (0,0,0)) # Render the score text

self.screen.blit(title, (0,0)) # Display the "Now playing" and score text

self.screen.blit(score\_text, (0,40))

Which produces:



#### Note fadeout

To create a fadeout gradient for the notes, PyGame’s ability to give alpha values (opacity values) to Surface objects will be used. The alpha value is out of 255, and the closer to 255 the alpha value is, the more transparent the object is.

fadeout\_steps = 50

fadeout\_gradient = []

**for** i **in** **range**(fadeout\_steps):

fadeout\_gradient.append(pygame.Surface((**int**(275/fadeout\_steps),720)))

This creates a list of Surface objects with equally sized widths that cover the correct amount of distance. The more items in the list the smoother the gradient will appear. Each Surface object is given a unique alpha value that gradually decreases across the list. To perform this, the fadeout\_gradient list is iterated over and each Surface object is given the correct alpha value before being displayed on-screen.

**for** i,part **in** **enumerate**(fadeout\_gradient):

part.set\_alpha(260-**int**((255/**len**(fadeout\_gradient))\*(i+1)))

part.fill((255,255,255))

self.screen.blit(part, (part.get\_rect().width\*i,0))

#### Note hitbox

This will be achieved by using PyGame’s alpha value once again.

hitbox = pygame.Surface((50,200)) # Create the hitbox for note detection

hitbox.set\_alpha(200)

hitbox.fill((153,217,234))

hitbox = self.screen.blit(hitbox, (time\_signature\_rect.right+25,340))

The hitbox is then filled blue and rendered on screen next to the time signature image.

#### Music notation components

Placing all the components is fairly straight forward – simply aligning the images and text in the correct location. Starting from left to right, the clef is placed first:

cleff\_rect = self.screen.blit(cleff, (0,310))

Then, the time signature is rendered (but not yet displayed):

beats\_text = self.font['80'].render(time\_signature[0], True, (0,0,0))

per\_bar = self.font['80'].render(time\_signature[1], True, (0,0,0))

Which is done with text rather than an image. If the key signature is needed to be displayed, it is done so first and then the time signature, otherwise the time signature is displayed straight away. Each component is placed at the right-most pixel of the last component to be placed:

**if** key **not** **in** ['C','Am']:

key\_rect = self.screen.blit(key,(cleff\_rect.right-75,300))

time\_signature\_rect = self.screen.blit(beats\_text, (key\_rect.right,350))

self.screen.blit(per\_bar, (key\_rect.right,430))

**else**:

time\_signature\_rect = self.screen.blit(beats\_text, (cleff\_rect.right-50,350))

self.screen.blit(per\_bar, (cleff\_rect.right-50,430))

The last part is to draw the stave. I decided not to use an image for this and simply draw five lines 20 pixels apart down the screen:

**for** i **in** **range**(0,200,40):

pygame.draw.line(self.screen, (0,0,0), (100,360+i), (1280,360+i), 5)

The reasons for not using an image for the stave is that this way it can easily be resized or modified if necessary, as well as the program overall taking up less storage and memory space due to there not being an extra image file to handle.

### Coding the entering and countdown of the performance screen

Upon entering a performance from the Song Select screen, I wanted a smooth transition and a slow start to give the user time to prepare to play, rather than the song simply starting immediately. I found the best way to do this was to use a fade-in and then countdown to give the user plenty of time to be ready. To start with, the variables are declared and the fade-in function is called:

**def** \_performanceScreen(self,song):

get\_song\_vars(song)

fade\_from\_white()

The fade\_from\_white sub-routine works by looping some amount of times, gradually decreasing the opacity of a PyGame Surface object that covers the entire screen.

The equation:

Is used to perform this task, as 255 is the maximum value for the opacity (alpha value) of a Surface object. (255÷(total number of loop iterations)) will create a fraction of equal parts so that on the last iteration of the loop it would result in 255, thus creating an alpha value of (255-255), or 0.

**def** fade\_from\_white():

overlay = pygame.Surface((1280,720))

overlay.fill((255,255,255)) # Set the colour to white

**for** i **in** **range**(200):

draw\_background()

overlay.set\_alpha(**round**(255-(255/200)\*i))

self.screen.blit(overlay, (0,0)) # Place at (0,0) on screen

pygame.display.flip()

Then, we want to start the main loop of the performance screen. It will be imperative to keep track of how many ticks (iterations of the loop) the program is currently on as this will allow the program to do timing, such as with keeping the notes in time with the tempo or displaying each number of the countdown after precisely a second. Furthermore, the PyGame clock will be limited to 60 frames per second so we know that there will always be 60 ticks each second, helping with the timing of parts of the performance.

The event variable, which is declared as null to begin with, keeps track of where the performance is at, and will have three states – before the performance starts, during the performance, and after the performance. This allows the program to know not display the countdown repeatedly or during the performance, and to stop the song from starting during the countdown.

**while** True:

self.clock.tick(60)

self.screen.fill((255,255,255)) # Fill the screen completely white

tick += 1

**if** event != 'playing' **and** tick % 60 == 0:

event = self.font['500'].render(**str**(**int**(4-(tick/60))), True, (0,0,0)) # Render countdown text

**if** **not** **float**(4-(tick/60)):

tick = 0

event = 'playing'

To display the text, event is set to a font render of the string value of:

As tick increases over the period of a second, the following calculations are made:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tick | 1 (1st second) | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| tick÷60 | 0.02 | 0.03 | 0.05 | 0.07 | 0.08 | 0.1 | 0.12 | 0.13 | 0.15 | 0.17 | 0.18 | 0.2 |
| 4-(above) | 3.98 | 3.97 | 3.95 | 3.93 | 3.92 | 3.9 | 3.88 | 3.87 | 3.85 | 3.83 | 3.82 | 3.8 |
| Floor(above) | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| … | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 (2nd second) |
| - | 0.82 | 0.83 | 0.85 | 0.87 | 0.88 | 0.9 | 0.92 | 0.93 | 0.95 | 0.97 | 0.98 | 1 | 1.02 |
| - | 3.18 | 3.17 | 3.15 | 3.13 | 3.12 | 3.1 | 3.08 | 3.07 | 3.05 | 3.03 | 3.02 | 3 | 2.98 |
| - | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 |

This is a very elegant way of solving the task, as otherwise three if-statements would need to be used.

When this equation ends up equalling 0 (at tick=240, ), the event variable is set to ‘playing’ and tick is reset to 0 to signify that the song should start.

### Testing the starting of the performance screen

#### Test plan

Upon clicking the ‘Play’ button on a song tab in the Song Select screen, the following should occur in order:

1. The song performance screen is called with the correct song as the input.
2. The screen should start completely white, and then fade into view over a short timeframe (less than 3 seconds). The background, including stave, clef, time and key signature, metronome, song name text and score text should always be visible during this process.
3. The countdown should begin once the fade-in is complete. The text should be clear and centred on the screen and countdown from 3 each second. One second after the 1 text is displayed the program should continue on and start the performance.

#### Results

1. To test whether the correct song is used when clicking the ‘Play’ button on the tab in the Song Select screen, the song argument is printed to the console as soon as the \_performanceScreen function is called:

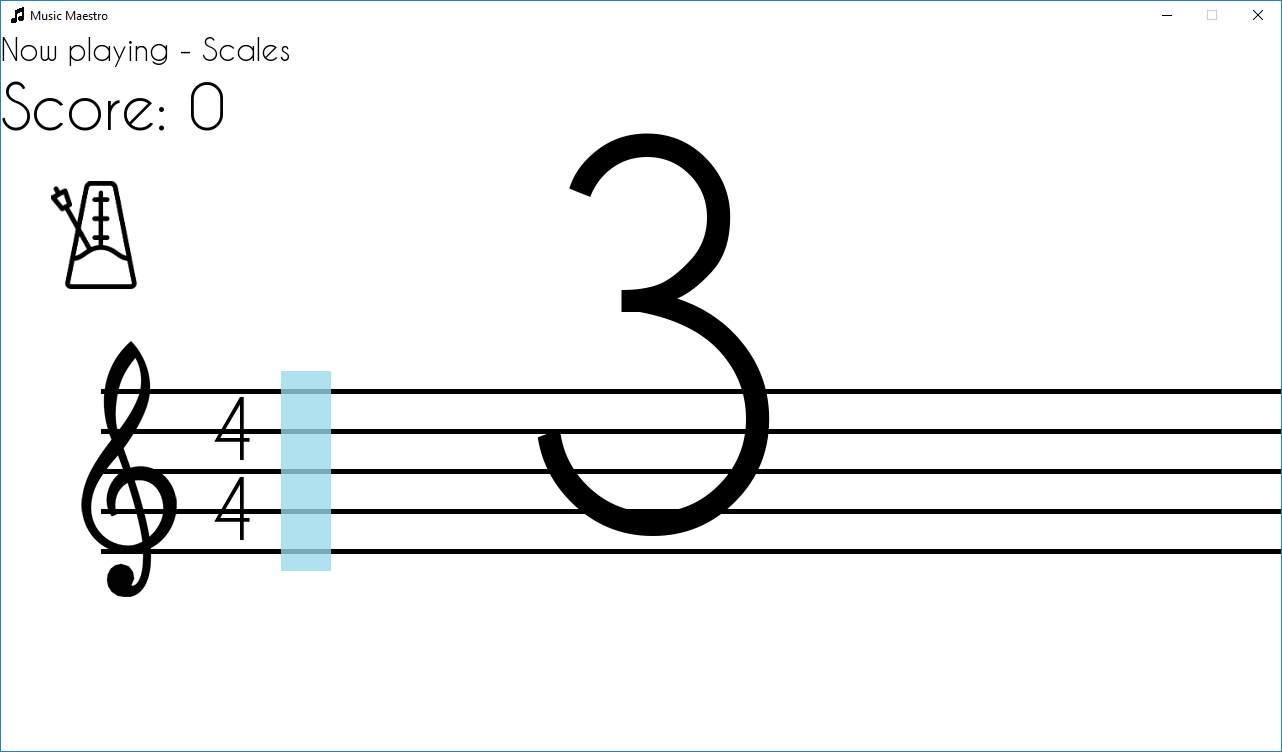
**def** \_performanceScreen(self,song):

**print**(song)

Upon clicking the ‘Scales’ song play button the following result is returned:

['Scales', 'Medium', '.\\assets\\songs\\Scales - Medium.txt']

And furthermore, the song name can be seen displayed in the top left of the screen:



1. The screen starts completely white, and gradually fades-in as expected. To test the time it takes, the following code is added:

**import** time

get\_song\_vars(song)

start\_time = time.perf\_counter()

fade\_from\_white()

**print**(time.perf\_counter()-start\_time)

time.perf\_counter is simply a floating point number that increases at the rate of the system clock, allowing an accurate time measurement in seconds to be drawn by subtracting a current reading of this value from the read value before the fade\_from\_white sub-routine.

This code returns:

2.0413451080000002

2.0102369520000014

2.082289038999999

After three attempts, which is under the desired three second maximum time.

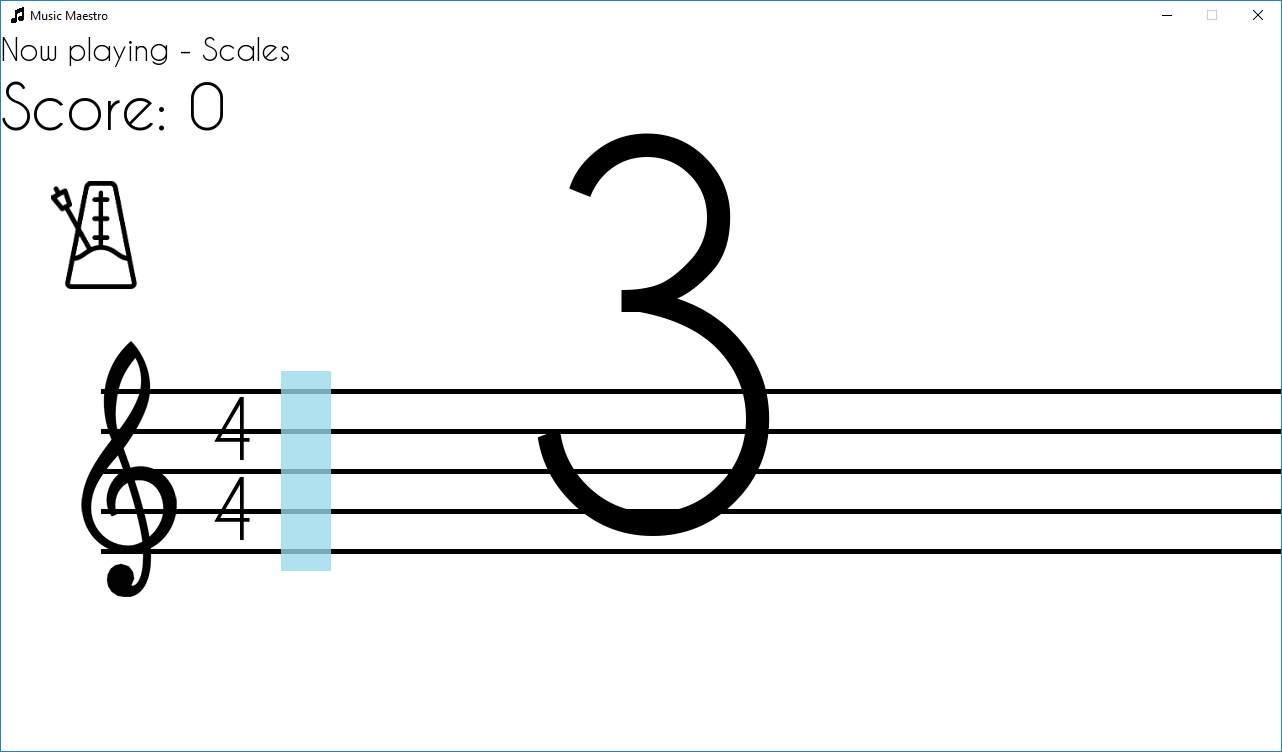
An error I became aware of was that trying to close the program during the fade-in would result in it freezing and becoming unresponsive, having to be closed using the system Task Manager program. This was because during the loop in fade\_from\_white, there was no checks to safely handle QUIT events. Adding the following code into the loop fixed this issue:

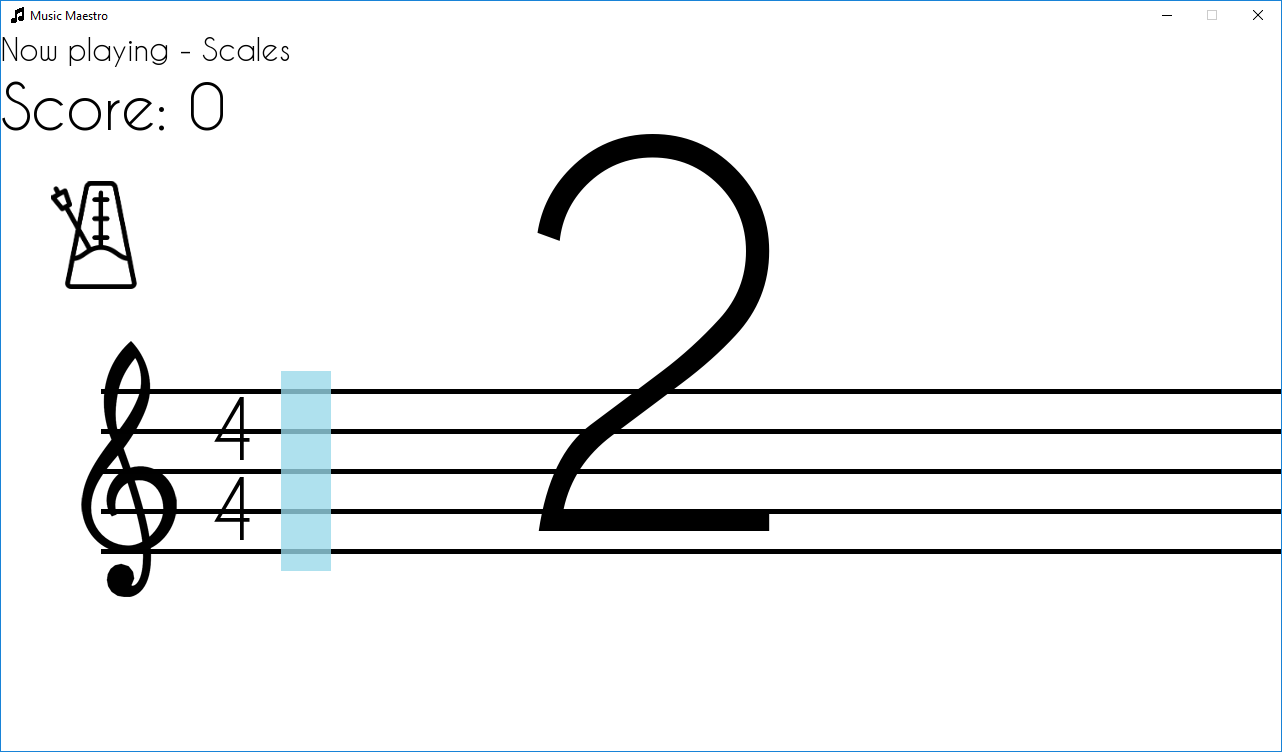
**for** event **in** pygame.event.get():

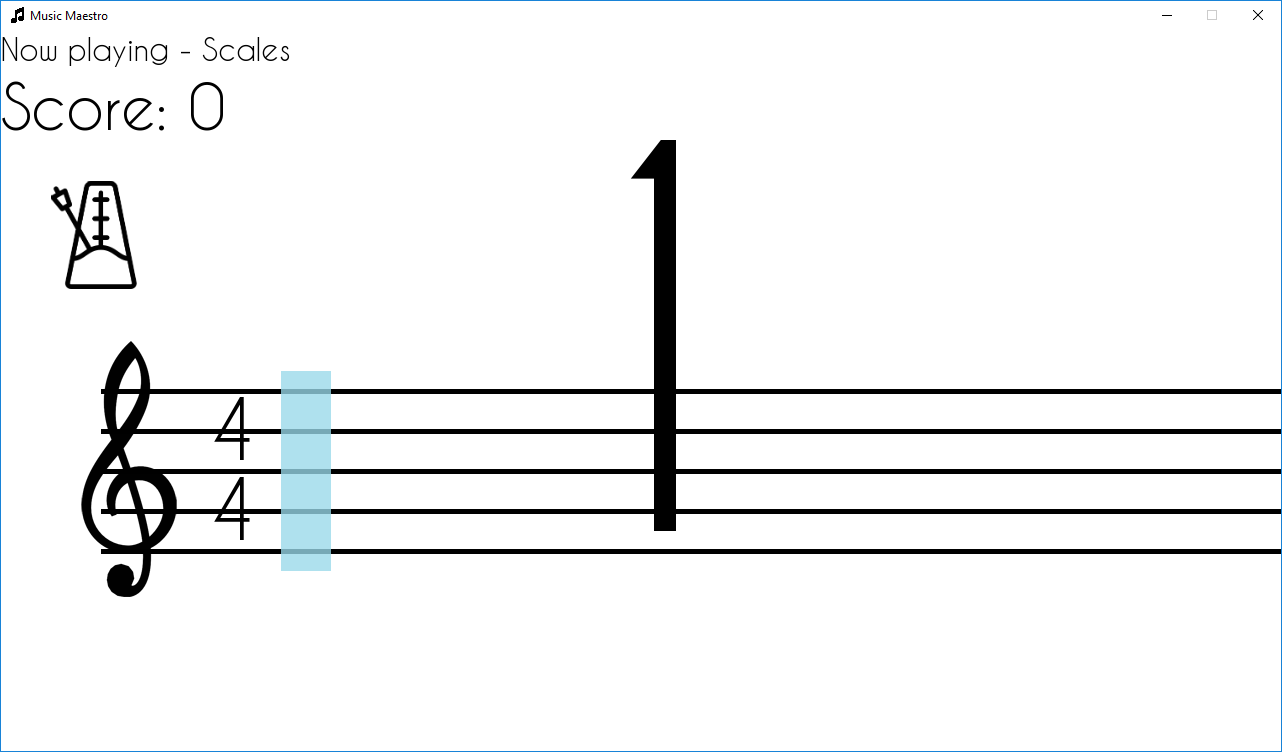
**if** event.**type** == QUIT:

**return** self.quit()

1. The text is large and centred, counting down from 3 correctly:







# Evaluation

## Usability Features

### Questionnaire

|  |  |  |
| --- | --- | --- |
| Question | Feedback | Comment |
| Before loading the program, what were you expecting your first actions in the program to be? | I was intending to go straight into a song and begin playing. | - |
| Upon loading the program, which button were you inclined to interact with first? | The song select button stood out as I wanted to go straight into playing. I was later interested in the log in feature however it was not the first thing I looked for. | This is okay, although it may be worth giving the user a notification to remind them progress won’t be saved if they do not first make an account when they first enter the song select menu. |
| Did you find the UI elements across the program easy and intuitive to use? | Yes, they were very familiar from other programs and I had no issue using the scroll bar or buttons. | - |
| Was the text across the program easy to read? | I had no issues reading text, the contrast between the text and background was always very clear. | - |
| How was performing – did you know what to do at the start? | Once I saw the notes moving left across the screen, I guessed what I had to do. It took a second or two to realise you had to play the note as it hit the blue bar and not immediately as you saw it, though. | Perhaps a tutorial could be shown the first time a user plays a song, to let them know what to expect during the performance. This would help reduce the guessing required the first time using the software. |
| How easy was it to interpret the post-performance review? | I liked how it was scored – it showed exactly how much I needed to improve. I think it should show which notes I missed and hit specifically though, that would be cool to see. | Logging which notes were hit and missed specifically to show the user after the performance would not be tricky to do and would make the review even more beneficial to the user. |
| How did you find creating and using an account? | I was a bit confused as to where to create an account, as the pop-up box indicated that it was for signing in, not creating an account. However, once I created the account it was very easy to log in and get all the high scores. I think it should remain logged in if you close and reopen the program, and a manual log out feature should be used instead. | Separating creating accounts and signing in is something that would be a high priority to do in the future, as it is the least intuitive part of the program. Making the user remain signed in and adding a log out feature would make accessing high scores much less irritating. |
| Overall, did you find the program easy to use? | Yes, very. I understood what I was doing for the most part and never got blindingly stuck by an aspect of the program. | This is the desired answer I wanted to receive from the client. |

## Success Criteria

|  |  |
| --- | --- |
| Criteria | Met? |
| A clean and intuitive, consistent design. | Yes |
| Main window with start menu UI including buttons to navigate to other menus. | Yes |
| Main window with options menu UI with buttons to navigate to log in screen and back to start menu. | Yes |
| Main window with song selects UI including button to go back to start menu, song tabs with buttons to start the song and a slider to navigate through the songs. | Yes |
| Create and sign into accounts. Text shown on screen of which account is currently signed in. | **No** |
| Song selection screen with a range of songs to choose from at varying difficulties. | Yes |
| Notes of the song shown clearly and accurate to music theory, large and unambiguous. | Yes |
| Playing a note into the microphone gets registered by the program. | Yes |
| Get on the fly feedback of each note played in the song. Shown by an accuracy mark out of 10 in text at the top of the screen. | Yes |
| After-performance analysis of how well the song was played, with a rating based from how other users performed. Should be concise and easily interpretable. | Yes |
| The song score to the account so it can be improved later. The user will be able to see the score of each song in the song select. | Yes |

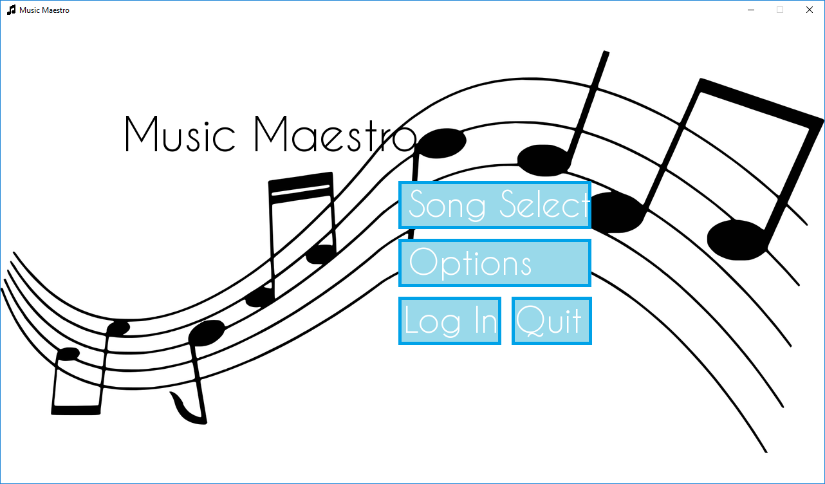
## Evidence

### A clean and intuitive, consistent design.

I decided to go with a clean and minimalistic look, with no screen clutter or complex visual designs. There are primarily only four colours: black, white, and two shades of blue which significantly helps keep the program easy to navigate and use. Blue objects usually refer to interactive parts of the program, whilst black and white are for text and design. This is consistent throughout the program, making it easy to know what to do when faced with a new screen for the first time.

### Main window with start menu UI including buttons to navigate to other menus.

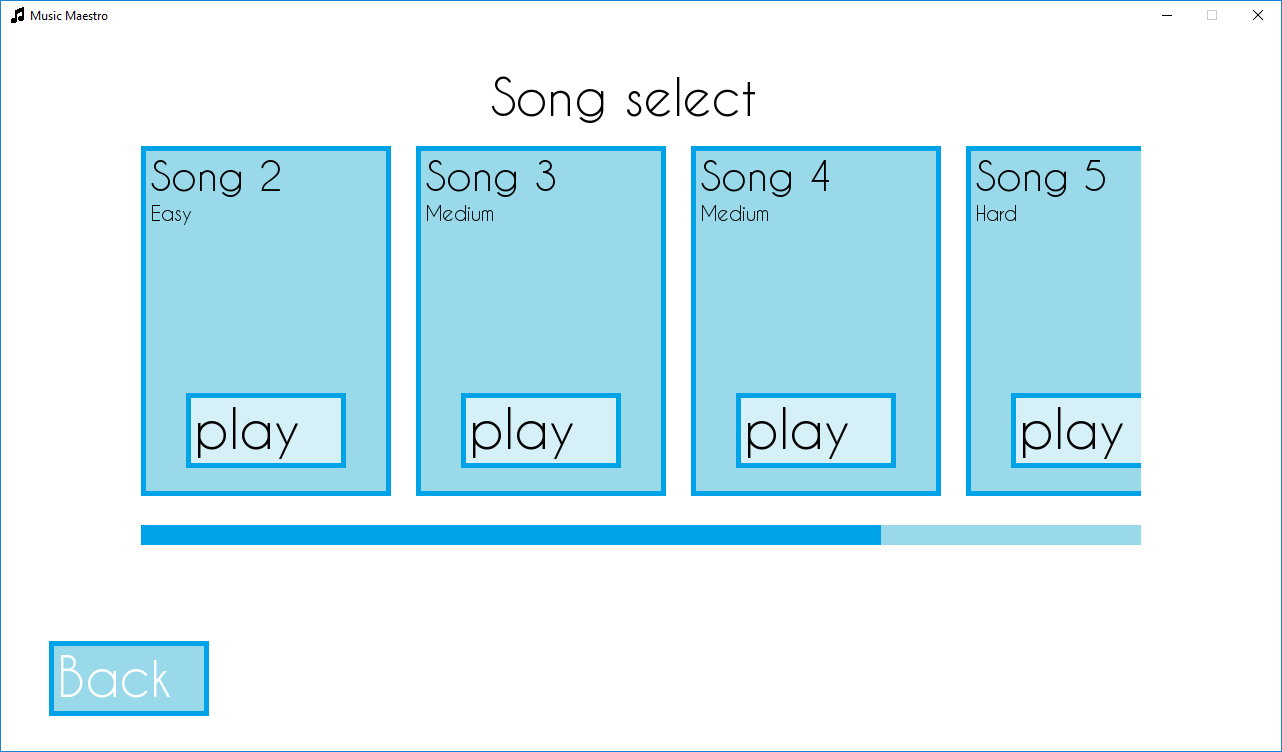
Buttons are clear and large and show exactly where they will take the user upon being clicked.



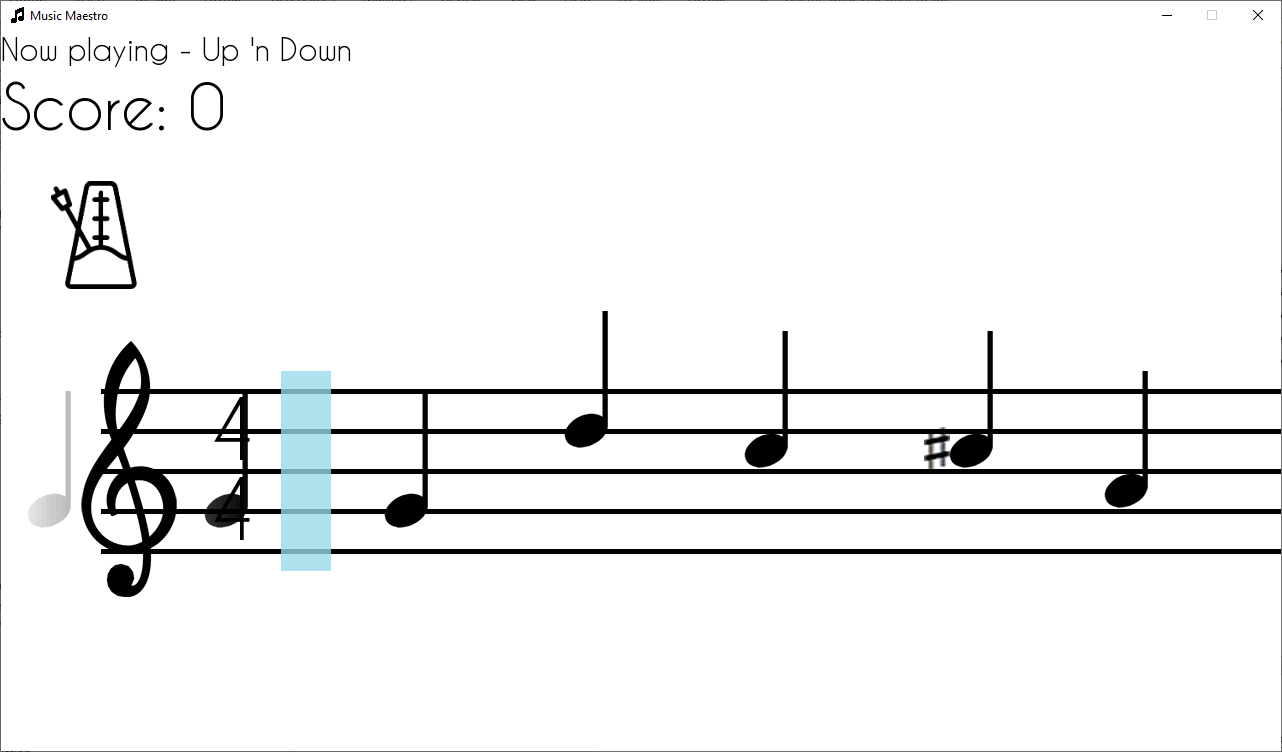
### Main window with song selects UI

Song tabs show song name, difficulty and high score, and have individual play buttons to start a song performance.

Scroll bar to allow the song tabs to be large and clear, without losing information when there are many songs to choose from.



### Notes of the song shown clearly and accurate to music theory, large and unambiguous.



The stave and notes take the majority of the screen space, with wide gaps between notes to show exactly what to play next. The notes fade to white after passing the hitbox, allowing view of the clef, time and key signature at all times. Sharp and flat notations are slightly larger than traditional sheet music also, to make it clear when it is expected to be played. A metronome is included above the clef, to aid with keeping in time and knowing when to play notes.

## Limitations and Further Additions

The greatest limitation of the program is how the largest factor to the accuracy of parsing notes played is the user’s microphone. If the user does not have a microphone, they would have to acquire one before being able to use the program. Furthermore, if the user’s microphone is low quality then the noise produced can interfere with what note is found. This issue will artificially deflate the user’s performance score for a reason out of their control.

To remedy this, the next steps of the program would be to create a denoising algorithm, using dynamic learning functions to isolate the desired frequencies away from the static and noise produced by most low-end microphones. The further the sound processing system can be taken, the more accurate it will be allowing for better feedback to the user. This could all be accomplished with further development time, and even multiple developers working on the software.

Other additions to make to the program would be to add even more supported notation as right now it is very basic. It was necessary to reduce the amount of supported notation with the time constraints in place, however support for instrument-specific notation and less common notation would be beneficial to the utility of the program.

Furthermore, perhaps a feature that allows the user to listen to a song, or part of it, along with the notation would be very good. With the limitations of the PyGame module, something like this would not be possible so an alternative solution would have to be found. However, it would allow users to know what exactly they should be sounding like if they are unsure of a complex rhythm scheme, for example.

## Maintenance

The project is extremely modular, making modifications very easy. By splitting the project over multiple files; song parsing, UI elements, database management, microphone parsing and the main file, it allows for changes to be made to one aspect of the program without affecting the others. Not only that, but the main file is also heavily modular, with each menu and performance screen in separate functions within a class structure, again allowing independence between them whilst remaining coherent as a whole.

This means that if stakeholders wish for a change in the design of the program, this can be done without affecting the functions such as how the database works, or components of the performance part of the program.

The code is thoroughly annotated and organised, making it very accessible to any new developers that may work on it in the future. Each function is described and has inputs and outputs commented and algorithms are described to help keep track of code that may be unfamiliar.

Future versions of the software could implement a resizable window and a full screen mode, to make it easier to use on a variety of monitors, as well as new notation and quality of life features such as joining quarter notes together to make it more accurate to traditional sheet music.

# Full Program

## Tree

**C:.**

│ main.pyw

│ readme.md

│

├───**assets**

│ │ buffer.wav

│ │ font.ttf

│ │

│ ├───**images**

│ │ │ icon.png

│ │ │ menu\_background.png

│ │ │ menu\_background\_a.png

│ │ │ menu\_background\_b.png

│ │ │ menu\_background\_c.png

│ │ │ menu\_background\_d.png

│ │ │ menu\_background\_e.png

│ │ │ menu\_background\_f.png

│ │ │ metronome\_left.png

│ │ │ metronome\_right.png

│ │ │

│ │ └───**notation**

│ │ Ab\_key\_signature.png

│ │ Bb\_key\_signature.png

│ │ Cb\_key\_signature.png

│ │ Db\_key\_signature.png

│ │ Eb\_key\_signature.png

│ │ eighth\_note.png

│ │ eighth\_rest.jpg

│ │ flat.jpg

│ │ Gb\_key\_signature.png

│ │ half\_note.jpg

│ │ half\_note.png

│ │ half\_rest.png

│ │ quarter\_note.png

│ │ quarter\_rest.png

│ │ sharp.png

│ │ sixteenth\_note.jpg

│ │ sixteenth\_note.png

│ │ sixteenth\_rest.png

│ │ temp\_key\_signature.png

│ │ treble\_cleff.png

│ │ whole\_note.jpg

│ │ whole\_note.png

│ │ whole\_rest.png

│ │

│ ├───**songs**

│ │ Megolovania - Hard.txt

│ │ Ode to Joy - Easy.txt

│ │ Scales - Medium.txt

│ │ Short n Sweet - Easy.txt

│ │ Up 'n Down - Medium.txt

│ │

│ └───**users**

│ ben.json

│

├───**scripts**

│ audio.py

│ song.py

│ ui\_elements.py

│ user.py

│

└───**write-up**

BFI01 - Music Maestro.docx

Music Maestro Demo.mp4

## Dependencies

- Python 3.7 (<https://www.python.org/downloads/>)

- pygame (<https://pypi.org/project/pygame/>)

- numpy (<https://pypi.org/project/numpy/>)

- scipy (<https://pypi.org/project/scipy/>)

- pyaudio (<https://pypi.org/project/PyAudio/>)

## main.pyw

#!/usr/bin/env python3

**try**:

**import** os

**import** threading

**import** pygame

**from** random **import** choice

**from** string **import** ascii\_letters, digits

**from** numpy **import** ceil

**from** pygame.**locals** **import** \*

**from** scripts.song **import** SongParser

**from** scripts.audio **import** SoundData

**from** scripts.ui\_elements **import** Button, ScrollBar, SongTab, TextInput

**from** scripts.user **import** User

**except** ImportError **as** e: # most likely a ModuleNotFoundError

**raise** Exception(f'Could not import a module: {e}.')

**class** Application:

**def** \_\_init\_\_(self):

'''

        Initialize an Application instance.

        '''

pygame.init() # initialize all imported pygame modules

pygame.mouse.set\_cursor(\*pygame.cursors.arrow)

self.images = {}

images = self.\_loadFiles()

**for** image **in** images:

self.images[image[1]] = pygame.image.load(image[0]) # load the image onto a new Surface object into a dictionary

self.backgrounds = [self.images['menu\_background\_a'],

self.images['menu\_background\_b'],

self.images['menu\_background\_c'],

self.images['menu\_background\_d'],

self.images['menu\_background\_e']]

self.overlay = pygame.Surface((1280,720))

self.overlay.set\_alpha(50)

self.overlay.fill((255,255,255))

self.user = User()

self.screen = pygame.display.set\_mode((1280, 720)) # initialize a window

pygame.display.set\_caption('Music Maestro') # set the text in the window caption (top left)

pygame.display.set\_icon(self.images['icon'])

self.clock = pygame.time.Clock()

self.audio = SoundData()

self.notes = {'A' :[440],

'A#/Bb':[466],

'B' :[493],

'C' :[523],

'C#/Db':[554],

'D' :[587],

'D#/Eb':[622],

'E' :[659],

'F' :[698],

'F#/Gb':[739],

'G' :[783],

'G#/Ab':[830]}

self.font = {**str**(i):pygame.font.Font('.\\assets\\font.ttf', i) **for** i **in** **range**(10,510,10)} # Load various font sizes

**return**

**def** \_loadFiles(self, t=['png','jpg']):

'''

        Load files from the directory the program is run from and all child directories.

        Args:

            t (list) : list of all file extensions to whitelist during the directory walk

                              default: ['png','jpg']

        Returns:

            location (list) : contains all file names and addresses of files that have the whitelisted extensions

        '''

dir\_contents = [[files,root] **for** root, dirs, files **in** os.walk('.')] # os.walk() will traverse all child directories from the argument returning a generator

# dir\_contents is a 2D array of all files (and their root suffix, e.g: .png) in all child directories

location = []

**for** x,dirs **in** **enumerate**(dir\_contents): # enumerate allows for iteration over an object with an automatic counter (in this case variable x)

# for each child directory in dir\_contents

**for** y,**file** **in** **enumerate**(dirs[0]): # for each file in each child directory

**file** = **file**.split('.') # split a string file name at each '.' into a list: 'picture.jpg' --> ['picture' , 'jpg']

**if** **file**[1] **in** t:

path = dirs[1] + '\\' + dir\_contents[x][0][y]

location.append([path,**file**[0]]) # add location details of a found file to location var where file[0]=name of file

**return** location

**def** \_menuScreen(self):

'''

        Handle the main menu screen including the event loop and button functionality.

        '''

# clear screen and set background color

self.screen.fill((255,255,255))

# set background image

background\_image = self.images['menu\_background']

self.screen.blit(background\_image, (0,40))

# create ui elements

song\_select\_button = Button(self, text='Song Select', text\_size=22, position=[768,288], dimensions=[300,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

options\_button = Button(self, text='Options' , text\_size=22, position=[768,378], dimensions=[300,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

buttons = [[song\_select\_button, self.\_songSelectScreen], # add button and the corresponding function to a list

[options\_button , self.\_optionsScreen ]]

**if** self.user.get\_username(): # if the user is signed in there is no need to have a login button

text = self.font['30'].render(f'Logged in as: {self.user.get\_username()}', True, (0,0,0)) # show logged in as text

self.screen.blit(text, (15, 675))

quit\_button = Button(self, text='Quit', text\_size=22, position=[768,468], dimensions=[300,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

buttons.append([quit\_button, self.quit])

**else**:

login\_button = Button(self, text='Log In', text\_size=22, position=[698,468], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

quit\_button = Button(self, text='Quit' , text\_size=22, position=[856,468], dimensions=[125,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

buttons.append([login\_button, self.\_logInScreen])

buttons.append([quit\_button , self.quit ])

# render title text

title = self.font['70'].render('Music Maestro', True, (0,0,0))

self.screen.blit(title, (190, 135))

# event loop

**while** True:

mouse\_pos = pygame.mouse.get\_pos() # get mouse position as a tuple

**for** event **in** pygame.event.get(): # iterate through all current pygame events

**if** event.**type** == QUIT:

**return** self.quit() # end the program if the quit event is performed

**if** event.**type** == MOUSEBUTTONDOWN:

**for** button **in** buttons:

**if** button[0].check(mouse\_pos):

button[1]()

**return**

**for** button **in** buttons:

button[0].check(mouse\_pos)

pygame.display.flip() # update screen

**return**

**def** \_songSelectScreen(self):

'''

        Handle the song select screen including the event loop, buttons and scroll bar functionality.

        '''

# background color

self.screen.fill((255,255,255))

# background image

background\_image = self.images['menu\_background\_f']

self.screen.blit(background\_image, (**round**(640-(background\_image.get\_size()[0]\*.5)),**round**(360-(background\_image.get\_size()[1]\*.5))))

self.screen.blit(self.overlay, (0,0))

# title text

title = self.font['50'].render('Song select', True, (0,0,0))

self.screen.blit(title, (490,36))

songs = self.\_loadFiles(t=['txt'])

**for** song **in** songs:

song[1] = song[1].split(' - ')

**del** songs[-1] # remove pseudocode.txt

# ui elements

song\_tabs = []

**for** place,song **in** **enumerate**(songs,start=1):

**if** self.user.get\_username():

user\_data = self.user.get\_data()

**try**:

song\_tabs.append(SongTab(self, start\_pos=**round**(place\*(250+25))-135, song=[songs[place-1][1][0],songs[place-1][1][1],songs[place-1][0]], highscore=user\_data[songs[place-1][1][0]]))

**except** KeyError:

song\_tabs.append(SongTab(self, start\_pos=**round**(place\*(250+25))-135, song=[songs[place-1][1][0],songs[place-1][1][1],songs[place-1][0]], highscore=0))

**else**:

song\_tabs.append(SongTab(self, start\_pos=**round**(place\*(250+25))-135, song=[songs[place-1][1][0],songs[place-1][1][1],songs[place-1][0]]))

back\_button = Button(self, text='Back', text\_size=22, position=[128,648], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

scroll\_bar = ScrollBar(self, dimensions=[1000,20], position=[640,504], scroll\_length=**round**(275\***len**(song\_tabs))-1025, color=(153,217,234), alt\_color=(0,162,232), clicked\_color=(0,131,187))

# event loop

**while** True:

mouse\_pos = pygame.mouse.get\_pos()

mouse\_clicked = pygame.mouse.get\_pressed()[0]

pygame.draw.rect(self.screen, (255,255,255), (0,92,1280,400))

scroll\_bar.check(mouse\_pos, mouse\_clicked)

back\_button.check(mouse\_pos)

**for** tab **in** song\_tabs:

tab.set\_x(scroll\_bar.get\_notch\_position())

**if** -275 < tab.get\_x() < 1280:

tab.render()

tab.button.check(mouse\_pos)

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

**if** event.**type** == MOUSEBUTTONDOWN:

**if** back\_button.check(mouse\_pos):

**return** self.\_menuScreen()

**for** tab **in** song\_tabs:

**if** tab.button.check(mouse\_pos):

**return** self.\_performanceScreen(tab.song)

pygame.draw.rect(self.screen, (255,255,255,255), (0,115,140,365))

pygame.draw.rect(self.screen, (255,255,255,255), (1140,115,1280,365))

pygame.display.flip()

**return**

**def** \_optionsScreen(self):

'''

        Handle the main menu screen including the event loop, buttons and slider functionality.

        '''

**def** render\_screen():

# background color

self.screen.fill((255,255,255))

# background image

self.screen.blit(background\_image, (**round**(640-(background\_image.get\_size()[0]\*.5)),**round**(360-(background\_image.get\_size()[1]\*.5))))

self.screen.blit(self.overlay, (0,0))

# title text

title = self.font['50'].render('Options', True, (0,0,0))

self.screen.blit(title, (490,36))

title = self.font['40'].render('Calibrate Microphone', True, (0,0,0))

self.screen.blit(title, (100,100))

title = self.font['40'].render('User Account', True, (0,0,0))

self.screen.blit(title, (750,100))

# ui elements

back\_button = Button(self, text='Back', text\_size=22, position=[128,648], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

calibration\_button = Button(self, text='Calibrate', text\_size=20, position=[300,200], dimensions=[200,60], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

buttons = [[back\_button,self.\_menuScreen],

[calibration\_button, calibrate\_microphone]]

**if** self.user.get\_username():

logout\_button = Button(self, text='Log Out', text\_size=20, position=[875,200], dimensions=[180,60], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

delete\_account\_button = Button(self, text='Delete Account', text\_size=26, position=[875,275], dimensions=[250,60], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

buttons.append([logout\_button , user\_logout ])

buttons.append([delete\_account\_button, delete\_account])

**else**:

login\_button = Button(self, text='Log In', text\_size=22, position=[875,200], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

buttons.append([login\_button, self.\_logInScreen])

**return** buttons

**def** calibrate\_microphone():

**for** note **in** self.notes.keys():

title\_text = self.font['40'].render('Please play the following note once:', True, (0,0,0))

note\_text = self.font['400'].render(note, True, (0,0,0))

self.screen.fill((255,255,255))

self.screen.blit(title\_text, (490,36))

self.screen.blit(note\_text, (0,25))

pygame.display.flip()

tick = 0

buffering = True

buffer = {}

**while** buffering:

self.clock.tick(60)

tick += 1

**if** tick < 40:

self.audio.stream()

**for** frequency **in** self.audio.get\_dominant\_frequencies().tolist():

buffer[**str**(frequency)] = buffer.get(**str**(frequency),0) + 1 # Creates a dictionary of how many times each dominant frequency appears in the timeframe

**else**:

buffering = False

**try**:

**del** buffer['1.0'] # Erroneous value that should be removed

**except** KeyError:

**pass** # Ignore error raised if the value is not found

buffer\_mode = **sorted**(**zip**(buffer.values(),buffer.keys()),reverse=True)[:3] # zip([A,B,C],[x,y,z])=[(A,x),(B,y),(C,z)] , sorted([2,87,1,9,2,4,56,8,0])=[0,1,2,2,4,8,9,56,87] --> Extract top 3 most frequent frequencies

self.notes[note] = [**int**(**float**(freq[1])) **for** freq **in** buffer\_mode]

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

self.\_optionsScreen()

**def** user\_logout():

self.user = User()

buttons = render\_screen()

**return** buttons

**def** delete\_account():

self.user.remove()

**return** user\_logout()

background\_image = choice(self.backgrounds)

buttons = render\_screen()

# event loop

**while** True:

mouse\_pos = pygame.mouse.get\_pos()

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

**if** event.**type** == MOUSEBUTTONDOWN:

**for** button **in** buttons:

**if** button[0].check(mouse\_pos):

buttons = button[1]()

**break**

**for** button **in** buttons:

button[0].check(mouse\_pos)

pygame.display.flip()

**return**

**def** \_logInScreen(self):

'''

        Handle the log in screen including the event loop, buttons and text box functionality.

        Returns:

            self.\_menuScreen (bound method) : returns to the main menu

        '''

**def** render\_screen():

# background color

self.screen.fill((255,255,255))

# background image

self.screen.blit(background\_image, (**round**(640-(background\_image.get\_size()[0]\*.5)),**round**(360-(background\_image.get\_size()[1]\*.5))))

self.screen.blit(self.overlay, (0,0))

# title text

title = self.font['50'].render('Log in', True, (0,0,0))

self.screen.blit(title, (565,36))

username\_text = self.font['40'].render('Username', True, (0,0,0))

self.screen.blit(username\_text, (340,125))

username\_text = self.font['40'].render('Password', True, (0,0,0))

self.screen.blit(username\_text, (340,235))

**def** login(username, password):

error = self.user.validate(username, password)

**if** **not** error:

**return** self.\_menuScreen()

**else**:

render\_screen()

error\_text = self.font['30'].render(error, True, (0,0,0))

self.screen.blit(error\_text, (340,325))

**return**

**def** create\_account(username, password):

error = self.user.create(username, password)

**if** **not** error:

**return** self.\_menuScreen()

**else**:

render\_screen()

error\_text = self.font['30'].render(error, True, (0,0,0))

self.screen.blit(error\_text, (340,325))

**return**

background\_image = choice(self.backgrounds)

render\_screen()

# ui elements

username\_input = TextInput(self, dimensions=(575, 40), position=(340,175), character\_limit=16, allowed\_characters=ascii\_letters+digits+'\_', color=(153,217,234), active\_color=(113,203,225))

password\_input = TextInput(self, dimensions=(225, 40), position=(340,285), character\_limit=16, allowed\_characters=ascii\_letters+digits+' ', input\_hidden=True, color=(153,217,234), active\_color=(113,203,225))

login\_button = Button(self, text='Log in', text\_size=22, position=[640,395], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

create\_account\_button = Button(self, text='Create account', text\_size=35, position=[640,475], dimensions=[320,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

back\_button = Button(self, text='Back', text\_size=22, position=[128,648], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

# event loop

**while** True:

mouse\_pos = pygame.mouse.get\_pos()

mouse\_clicked = pygame.mouse.get\_pressed()[0]

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

**if** event.**type** == MOUSEBUTTONDOWN:

**if** login\_button.check(mouse\_pos):

login(username\_input.get\_value(), password\_input.get\_value())

**if** create\_account\_button.check(mouse\_pos):

create\_account(username\_input.get\_value(), password\_input.get\_value())

**if** back\_button.check(mouse\_pos):

**return** self.\_menuScreen()

**if** event.**type** == KEYDOWN:

**if** event.**unicode**:

username\_input.key\_press(event.**unicode**, key\_type='char')

password\_input.key\_press(event.**unicode**, key\_type='char')

**else**:

username\_input.key\_press(event.key, key\_type='action')

password\_input.key\_press(event.key, key\_type='action')

login\_button.check(mouse\_pos)

create\_account\_button.check(mouse\_pos)

back\_button.check(mouse\_pos)

username\_input.check(mouse\_pos,mouse\_clicked)

password\_input.check(mouse\_pos,mouse\_clicked)

pygame.display.flip()

**return** self.\_menuScreen()

**def** \_performanceScreen(self,song):

'''

        Run the song performance of a selected song.

        Args:

            song (list) : details of the song to be run, consisting of song name, difficulty, and file address

        '''

# Declare global variables, used in the sub-routines

**global** Song, score, tempo, cleff, time\_signature, key, song\_length

**global** metronome, current\_note, current\_mic\_note, tick, event, beats, note\_buffer, note\_played\_data

**global** hitbox, metronome\_offset

**def** get\_song\_vars(song):

**global** Song, score, tempo, cleff, time\_signature, key, song\_length

**global** metronome, current\_note, current\_mic\_note, tick, event, beats, note\_buffer, note\_played\_data

# Variables from song file

**with** **open**(song[2], mode='r') **as** **File**:

song\_contents = **File**.read().split('|') # Read the song file into a list

**for** i,bar **in** **enumerate**(song\_contents):

song\_contents[i] = bar.strip('\n').split(',') # Strip all newlines and split into individual notes

Song = SongParser(song\_contents, self.images) # Load SongParser object

tempo = **int**(song\_contents[0][0].split('=')[1]) # Identify song variables

cleff = self.images[song\_contents[0][1].split('=')[1]+'\_cleff']

time\_signature = song\_contents[0][2].split('=')[1].split('/')

key = song\_contents[0][3].split('=')[1]

**if** key **not** **in** ['C','Am']: # The key of C and Am have no special notation, so only load the key image if it is not C or Am

key = self.images[key+'\_key\_signature']

song\_length = **int**(song\_contents[0][4].split('=')[1])

# Miscellaneous variables

metronome = 'left'

current\_mic\_note = 'X'

current\_note = 'Y'

score = 0

tick = 0

event = None

beats = 0

note\_buffer = [Song.next\_note()]

note\_played\_data = []

**return**

**def** fade\_from\_white():

**for** i **in** **range**(200):

draw\_background()

overlay = pygame.Surface((1280,720))

overlay.set\_alpha(**round**(255-(255/200)\*i))

overlay.fill((255,255,255))

self.screen.blit(overlay, (0,0))

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

pygame.display.flip()

**return**

**def** draw\_background():

**global** hitbox, metronome\_offset

# Draw note fadeout gradient

fadeout\_steps = 50

fadeout\_gradient = [pygame.Surface((**int**(275/fadeout\_steps),720)) **for** i **in** **range**(fadeout\_steps)] # 275 is the distance between left of screen to left of hitbox

**for** i,part **in** **enumerate**(fadeout\_gradient):

part.set\_alpha(260-**int**((255/**len**(fadeout\_gradient))\*(i+1)))

part.fill((255,255,255))

self.screen.blit(part, (part.get\_rect().width\*i,0))

# Draw text

title = self.font['30'].render('Now playing - {}'.format(song[0]), True, (0,0,0)) # Render the "Now playing" text

score\_text = self.font['60'].render('Score: {}'.format(score), True, (0,0,0)) # Render the score text

self.screen.blit(title, (0,0)) # Display the "Now playing" and score text

self.screen.blit(score\_text, (0,40))

# Draw metronome

self.screen.blit(self.images['metronome\_'+metronome], (50,150)) # Draw the metronome

# Draw stationary music components

cleff\_rect = self.screen.blit(cleff, (0,310)) # Display the cleff

beats\_text = self.font['80'].render(time\_signature[0], True, (0,0,0)) # Display the time signature

per\_bar = self.font['80'].render(time\_signature[1], True, (0,0,0))

**if** key **not** **in** ['C','Am']: # Display the key signature (the key of C and Am do not have any special notation so only do this if it is any other key signature)

key\_rect = self.screen.blit(key,(cleff\_rect.right-75,300))

time\_signature\_rect = self.screen.blit(beats\_text, (key\_rect.right,350))

self.screen.blit(per\_bar, (key\_rect.right,430))

**else**:

time\_signature\_rect = self.screen.blit(beats\_text, (cleff\_rect.right-50,350))

self.screen.blit(per\_bar, (cleff\_rect.right-50,430))

**for** i **in** **range**(0,200,40): # Draw the stave

pygame.draw.line(self.screen, (0,0,0), (100,360+i), (1280,360+i), 5)

hitbox = pygame.Surface((50,200)) # Create the hitbox for note detection

hitbox.set\_alpha(200)

hitbox.fill((153,217,234))

hitbox = self.screen.blit(hitbox, (time\_signature\_rect.right+25,340))

metronome\_offset = **round**(((self.screen.get\_width()-hitbox.left)/(tempo/20))/(60\*\*2/tempo))

**return**

**def** run\_audio\_stream():

**while** True:

self.audio.stream()

**if** event != 'playing':

**break**

**return**

get\_song\_vars(song)

fade\_from\_white()

**while** True:

self.clock.tick(60)

self.screen.fill((255,255,255)) # Fill the screen completely white

tick += 1

**if** event != 'playing' **and** tick % 60 == 0:

event = self.font['500'].render(**str**(4-(tick//60)), True, (0,0,0)) # Render countdown text

**if** **not** 4-(tick//60):

tick = 0

event = 'playing'

stream\_thread = threading.Thread(target=run\_audio\_stream)

stream\_thread.start()

**if** event **and** event != 'playing':

self.screen.blit(event, (500,10))

**elif** event == 'playing':

**if** tick % (60\*\*2/tempo) == 0:

note\_buffer[-1]['long\_duration\_bool'] += 1

**if** (tick-(3\*metronome\_offset)) % (60\*\*2/tempo) == 0:

beats += 1

**if** metronome == 'left':

metronome = 'right'

**else**:

metronome = 'left'

**if** tick:

**if** tick % **round**((60\*\*2/tempo)\*note\_buffer[-1]['note\_length']) == 0 **and** note\_buffer[0]:

**if** Song.end\_of\_bar:

note\_buffer.append({'pos': 1280,

'note\_length': note\_buffer[-1]['note\_length'],

'note\_name': 'X',

'long\_duration\_bool': 0,

'played': 5,

'note\_img\_offset':0,

'note\_img': [None, 0]})

Song.end\_of\_bar = False

**else**:

**if** note\_buffer[-1]['note\_length'] > 1:

**if** note\_buffer[-1]['long\_duration\_bool'] == note\_buffer[-1]['note\_length']:

note\_buffer.append(Song.next\_note())

**else**:

tick += 60\*\*2/tempo

**else**:

note\_buffer.append(Song.next\_note())

**if** Song.end\_of\_bar:

note\_buffer[-1]['note\_length'] /= 2

**if** note\_buffer[-1] == None: # End of the song is reached

**del** note\_buffer[-1]

note\_buffer.insert(0,None)

**if** tick > 10:

current\_mic\_frequencies = self.audio.get\_dominant\_frequencies()

current\_mic\_note = self.audio.get\_note\_from\_frequency(self.notes, current\_mic\_frequencies)

note = self.font['30'].render(f'Detected note: {current\_mic\_note}', True, (0,0,0))

self.screen.blit(note, (975,675))

**if** current\_note **in** current\_mic\_note.split('/'):

**for** note **in** note\_buffer:

**if** note:

**if** note['note\_name'][-1] == 'n':

note\_name = note['note\_name'][0]

**else**:

note\_name = note['note\_name'][0] + note['note\_name'][-1]

**if** note\_name == current\_note:

note['played'] -= 1

**break**

**for** i,note **in** **enumerate**(note\_buffer):

**if** note:

note['pos'] -= tempo/20

**if** hitbox.left < note['pos'] < hitbox.right:

**if** note['note\_name'][-1] == 'n':

current\_note = note['note\_name'][0]

**else**:

current\_note = note['note\_name'][0] + note['note\_name'][-1]

**elif** note['pos'] < hitbox.left-40:

note['played'] = 2\*\*40

**if** note['played'] <= 0:

note['pos'] = -40

current\_note = 'X'

score += 1

**for** j,part **in** **enumerate**(**list**(note.keys())[6:]):

**if** j:

self.screen.blit(note[part][0], (note['pos']-25,note[part][1]))

**else**:

**if** note[part][0]:

self.screen.blit(note[part][0], (note['pos'],note[part][1]))

**if** note[part][1]+note['note\_img\_offset'] > 540:

**for** k **in** **range**(((note[part][1]+note['note\_img\_offset'])-520)//40):

pygame.draw.line(self.screen, (0,0,0), (note['pos']-10,560+(40\*k)), (note['pos']+60,560+(40\*k)), 5)

**elif** note[part][1] < 340:

**for** k **in** **range**(((360-(note[part][1]+note['note\_img\_offset']))//40)):

pygame.draw.line(self.screen, (0,0,0), (note['pos']-10,320-(40\*k)), (note['pos']+60,320-(40\*k)), 5)

**else**:

pygame.draw.line(self.screen, (0,0,0), (note['pos'],360), (note['pos'],520), 4)

**if** note['pos'] <= -40:

**del** note\_buffer[i] # Delete the note when it's x-position is off screen

**if** note['note\_name'] != 'X':

note\_played\_data.append(**round**(100\*(score/(**len**(note\_played\_data)+1)))) # Percent of notes played correctly out of all the notes so far

**else**:

note\_buffer[i] = note

**if** note\_buffer == [None]:

event = 'complete'

stream\_thread.join()

**return** self.\_analysisScreen([song[0], score, **round**(100\*(score/song\_length)), song\_length, note\_played\_data])

**for** e **in** pygame.event.get():

**if** e.**type** == QUIT:

**return** self.quit()

**if** e.**type** == KEYDOWN:

**if** e.key == K\_ESCAPE:

**if** event == 'playing':

event = 'complete'

stream\_thread.join()

**return** self.\_menuScreen()

draw\_background()

pygame.display.flip()

# As each note collides with the hitbox, set "current note" to the note value, or None if a rest

# If some function current\_note\_being\_played() == current\_note, then they successfully played it

# else, they did not

# could be an analogue thing to allow for more precise timings (based on how close the note is to the center of the hitbox)

# current\_note\_being\_played() should also have a None value if no note is detected

**return**

**def** \_analysisScreen(self, score):

'''

        Display song performance scores, and graph to show where the errors in the performance occurred.

        Args:

            score (list) :

        '''

**def** graph(total\_notes, percents):

# Draw graph details

pygame.draw.line(self.screen, (0,0,0), (640,125), (640,625), 4) # Draw y-axis

pygame.draw.line(self.screen, (0,0,0), (640,625), (1140,625), 4) # Draw x-axis

graph\_underlay = pygame.Surface((498,499)) # Create a translucent underlay under the graph area to obscure potentially distracting background image

graph\_underlay.set\_alpha(200)

graph\_underlay.fill((255,255,255))

self.screen.blit(graph\_underlay, (643,125))

# Convert x and y value to x and y on-screen coordinates

y = **lambda** k: **int**(ceil((625-10)-(((500/(100-min\_percent))\*(k-min\_percent))))) # Ceiling of: (origin\_y-offset) - ((length of axis/range of percents)\*i)

x = **lambda** k: **int**(ceil((640-5)+(((500/(total\_notes-1))\*k)))) # Ceiling of: (origin\_x) + ((total length of axis - text height/range of percents)\*i)

min\_percent = **int**(**min**(percents)\*.1)\*10

**if** min\_percent == 100: # Prevent axis from having only one value on it

min\_percent = 90

**for** i **in** **range**(0,(100-min\_percent)+10,10): # Calculate positions of and display graph y-axis label text

text = self.font['20'].render(f'{i+min\_percent}%', True, (0,0,0))

self.screen.blit(text, (590,y(i+min\_percent)))

x\_positions = []

**for** i **in** **range**(total\_notes): # Calculate positions of and display graph x-axis label text

**if** total\_notes <= 20 **or** i==0 **or** i==total\_notes-1:

text = self.font['20'].render(**str**(i+1), True, (0,0,0))

self.screen.blit(text, (x(i),640))

x\_positions.append(x(i)+6)

y\_label = self.font['30'].render('Accuracy', True, (0,0,0)) # Label text for y-axis

self.screen.blit(y\_label, (450,350))

x\_label = self.font['30'].render('Note count', True, (0,0,0)) # Label text for x-axis

self.screen.blit(x\_label, (830,660))

# Plot points

percents = [y(percent)+10 **for** percent **in** percents]

**for** point **in** **zip**(x\_positions,percents):

**try**:

pygame.draw.aaline(self.screen, (0,0,0), point, last\_point, 4)

**except** UnboundLocalError: # On the first iteration of the loop, catch the error caused by last\_point not yet being defined

**pass**

last\_point = point

**for** point **in** **zip**(x\_positions,percents):

pygame.draw.circle(self.screen, (255,0,0), point, 4)

# background color

self.screen.fill((255,255,255))

# background image

background\_image = choice(self.backgrounds)

self.screen.blit(background\_image, (**round**(640-(background\_image.get\_size()[0]\*.5)),**round**(360-(background\_image.get\_size()[1]\*.5))))

self.screen.blit(self.overlay, (0,0))

# title text

title = self.font['50'].render('Song Performance', True, (0,0,0))

song\_name\_text = self.font['40'].render(f'Song name: {score[0]}', True, (0,0,0))

score\_text = self.font['40'].render(f'Score: {score[1]}', True, (0,0,0))

percent\_text = self.font['40'].render(f'Percent: {score[2]}%', True, (0,0,0))

self.screen.blit(title, (490,36))

self.screen.blit(song\_name\_text, (25,86))

self.screen.blit(score\_text, (25,126))

self.screen.blit(percent\_text, (25,170))

# ui elements

back\_button = Button(self, text='Back', text\_size=22, position=[128,648], dimensions=[160,75], color=(153,217,234), alt\_color=(0,162,232), hover\_color=(113,203,225), text\_color=(255,255,255))

# update user data

**if** self.user.get\_username():

user\_data = self.user.get\_data()

**try**:

**if** score[2] > user\_data[score[0]]: # score > high score (%)

user\_data[score[0]] = score[2] # high score = score (%)

**except** KeyError:

user\_data[score[0]] = score[2]

**finally**:

self.user.save(user\_data)

highscore\_text = self.font['40'].render('High score: {0}%'.format(user\_data[score[0]]), True, (0,0,0))

self.screen.blit(highscore\_text, (25,210))

**if** score[2] > 1: # Only show graph if song contains more than one note to avoid division by zero error

graph(score[3],score[4])

# event loop

**while** True:

mouse\_pos = pygame.mouse.get\_pos()

mouse\_clicked = pygame.mouse.get\_pressed()[0]

**for** event **in** pygame.event.get():

**if** event.**type** == QUIT:

**return** self.quit()

**if** event.**type** == MOUSEBUTTONDOWN:

**if** back\_button.check(mouse\_pos):

**return** self.\_menuScreen()

back\_button.check(mouse\_pos)

pygame.display.flip()

**return**

**def** run(self):

'''

        Higher level initilization of the program.

        '''

self.\_menuScreen()

**return**

**def** quit(self):

'''

        End all PyGame processes and close the PyGame window.

        '''

pygame.font.quit()

pygame.quit()

**return**

**if** \_\_name\_\_ == '\_\_main\_\_':

MusicMaestro = Application()

MusicMaestro.run()

**raise** SystemExit

## audio.py

**try**:

**import** numpy **as** np

**import** wave

**from** scipy.io **import** wavfile

**from** pyaudio **import** PyAudio, paInt16

**except** ImportError **as** e: # most likely a ModuleNotFoundError

**raise** Exception(f'Could not import a module: {e}.')

**class** SoundData:

**def** \_\_init\_\_(self, chunk=1024, rate=44100):

'''

        Initialize a SoundData object.

        Args:

            chunk (int) : number of samples grouped together

                          default: 1024

            rate (int) : sampling frequency in Hz

                          default: 44100

        '''

self.chunk = chunk

self.rate = rate

self.buffer = None

self.audio\_stream = PyAudio().**open**(format=paInt16, # Create an audio stream object from the microphone using PyAudio

channels=1,

rate=rate,

**input**=True,

frames\_per\_buffer=chunk)

**def** \_write\_stream\_to\_file(self, filename, data):

'''

        Write contents of data to a Wave file.

        Args:

            filename (str) : name of Wave file to be written to

            data (list) : mono audio signal

        '''

wave\_file = wave.**open**(f'./assets/{filename}.wav', 'wb') # Open the Wave file in binary write mode

wave\_file.setnchannels(1) # Set details of the data being written

wave\_file.setsampwidth(PyAudio().get\_sample\_size(paInt16))

wave\_file.setframerate(self.rate)

wave\_file.writeframes(b''.join(data)) # Convert the list into a binary string and (over)write to the Wave file

wave\_file.close()

**def** \_framing(self, data):

'''

        Transform audio signal into a series of overlapping frames.

        A frame (sample) is the amplitude at a point in time.

        Args:

            data (list) : mono audio signal

        Returns:

            frames (list) : all the frames

            frame\_length (int) : length of each frame

        '''

frame\_length = **int**(.025 \* self.rate) # Frame length = (window length) \* (rate), .025 secs chosen arbitrarily

frame\_step = **int**(.01 \* self.rate) # Used to convert from seconds to samples, .01 secs between windows chosen arbitrarily

signal\_length = **len**(data)

number\_of\_frames = **int**(np.ceil(**abs**(signal\_length-frame\_length)/frame\_step)) # Check there is at least one frame

# Find indices

index\_a = np.tile(np.arange(0, frame\_length), (number\_of\_frames, 1)) # numpy.arange(start,stop,step) returns evenly spaced values between start & stop

# numpy.tile(array, repeats) constructs an array by repeating the given array in each given axis (repeats)

index\_b = np.tile(np.arange(0, number\_of\_frames\*frame\_step, frame\_step), (frame\_length, 1))

index\_b = np.transpose(index\_b) # Rearrange the array so rows become columns and colums become rows

indices = index\_a + index\_b

# Pad out the signal to ensure the frames have at least the same length as the indices array

padding\_amount = number\_of\_frames \* frame\_step + frame\_length

padding = np.zeros((padding\_amount-signal\_length)) # Creates a numpy array filled entirely of zeros

padded\_buffer = np.append(data, padding) # Merges two arrays into one

frames = padded\_buffer[indices.astype(np.int32, copy=False)] # .astype(dtype, copy=False) changes the type of the indices array to int32

**return** frames, frame\_length

**def** \_get\_dominant\_frequency(self, frame):

'''

        Find the dominant frequency of a single frame.

        Args:

            frame (numpy.ndarray) : amplitude information at a point in time

        Returns:

                          (float) : dominant frequency in Hz

        '''

nfft = 2\*\*14 # Fast fourier transform points to be calculated

fourier\_transform = np.fft.rfft(frame, nfft) # Perform a fast fourier transform on a real input

magnitude\_spectrum = (1/nfft) \* **abs**(fourier\_transform)

power\_spectrum = (1/nfft)\*\*2 \* magnitude\_spectrum\*\*2

frequencies = np.fft.fftfreq(**len**(power\_spectrum), 1/self.rate) # Gives the frequencies associated with the coefficients: .fftfreq(window\_length,sampling\_spacing) where sampling\_spacing is the inverse of sampling rate

frequencies = (frequencies[np.where(frequencies >= 0)] // 2) + 1 # Filter out negative frequencies and return the floor division of 2 for each frequency. Finally, add 1 to each frequency

power\_spectrum = power\_spectrum[:**len**(frequencies)] # Take only the first half of the spectra as only the first part contains useful data

maxiumum\_index = np.argmax(power\_spectrum) # .argmax() returns the maximum values along an axis

**return** frequencies[maxiumum\_index] # Convert the dominant frequency to Hz

**def** stream(self, time=.1):

'''

        Update audio stream buffer.

        Args:

            time (float) : length of audio stream buffer in seconds

                           default: 0.1

        '''

# To record (time) seconds into the buffer, we must take (rate)\*(time) samples.

# In each iteration (chunk) samples are taken, so we must loop (rate)\*(time)/(chunk) times.

buffer\_hex = [self.audio\_stream.read(self.chunk) **for** i **in** **range**(**int**(self.rate/self.chunk\*time))]

self.\_write\_stream\_to\_file('buffer', buffer\_hex)

self.rate, self.buffer = wavfile.read('./assets/buffer.wav')

**def** get\_dominant\_frequencies(self):

'''

        Analyse the buffer data to find the dominant frequencies.

        Returns:

            dominant\_frequencies (list) : list of the dominant frequencies identified

        '''

# Perform framing on the signal

frames, frame\_length = self.\_framing(self.buffer)

# Perform Hamming window function on the frames

windows = frames \* np.hamming(frame\_length) # w(n) = .54 - .46\*cos((2\*(pi)\*n)/(M-1)) , 0 <= n <= M-1 where M = number of points in the output window

dominant\_frequencies = np.array([self.\_get\_dominant\_frequency(window) **for** window **in** windows]) # Find the dominant frequency for each frame

dominant\_frequencies = np.**round**(dominant\_frequencies, 3) # Round to three decimal places

dominant\_frequencies = np.unique(dominant\_frequencies) # Remove all duplicate values

**return** dominant\_frequencies

**def** get\_note\_from\_frequency(self, notes\_dict, frequencies):

'''

        Convert a list of frequencies into their likeliest music note.

        Args:

            notes\_dict (dict) : dictionary of notes and their associated frequencies

            frequencies (list) : list of frequencies

        Returns:

            note (str) : single note or None if no note identified

        '''

**if** 1.0 **in** frequencies.tolist():

**return** 'rest' # If 1.0 is a dominant frequency assume it is background noise

**for** note **in** notes\_dict.keys():

target = notes\_dict[note]

weight = 0

**for** freq **in** frequencies:

min\_distance\_from\_target = **min**([**abs**(100\***round**(np.sin((np.pi/np.log(2))\*np.log(freq/value)),4)) **for** value **in** target])

**if** **not** min\_distance\_from\_target:

min\_distance\_from\_target = -100

weight += min\_distance\_from\_target

**try**:

**if** weight < closest\_match[1]:

closest\_match = [note, weight]

**except** NameError: # On the first iteration closest\_match has not yet been declared

closest\_match = [note, weight]

**return** closest\_match[0]

## song.py

**try**:

**import** pygame

**except** ImportError **as** e: # most likely a ModuleNotFoundError

**raise** Exception(f'Could not import a module: {e}.')

**class** SongParser:

**def** \_\_init\_\_(self, song, images):

'''

        Class to handle the translation of song files into their components.

        Args:

            song (list) : the song file with all the song data

            images (dict) : contains all the image Surface objects and names the program uses

        '''

self.song = song[1:-1][::-1] # Remove the metadata at index 1 and blank entry at index -1 and reverse the list

**for** i,bar **in** **enumerate**(self.song):

self.song[i] = bar[::-1] # Reverse the notes in each bar

self.images = images

self.current\_bar = self.song.pop()

# Creates a list of note names by iterating through ['G','F','E','D','C','B','A'], each iteration of range(7,0,-1) (which counts down from 7 to 1). Then it adds both str values together into the list

# Finally, the list (['G7', 'F7', 'E7', ... , 'D1', 'C1', 'B1', 'A1']) is iterated through with each value becoming a key in the dictionary with a corresponing y-value

self.y\_pos = {i:**int**(count\*20+60) **for** count,i **in** **enumerate**([y+**str**(x) **for** x **in** **range**(7,0,-1) **for** y **in** ['G','F','E','D','C','B','A']])} # Dictionary of each note and its y-value on screen

self.tilt = {'#':images['sharp'],

'b':images['flat'],

'n':None}

self.durations = {0.25:'sixteenth\_',

0.5:'eighth\_',

1.0:'quarter\_',

2.0:'half\_',

4.0:'whole\_'}

**def** \_parse(self, note):

'''

        Translate a string note/rest name and duration into a dictionary of data.

        Args:

            note (str) : contains information about note/rest name, duration and whether it is sharp or flat or not

        Returns:

            image (dict) : all the data of the note, including loaded image files

        '''

image = {'pos':1280} # x-coord of where each note should start in the window

image['note\_length'] = **float**(note[3:-1]) # Append note length

image['note\_name'] = note[:3] # Append note as string

image['long\_duration\_bool'] = 0 # Used to track if a note longer than 1 beat has been spawned for its note duration

image['played'] = 5 # Number of times microphone detected note must match before the note is successfully played

duration = self.durations[image['note\_length']] # Lookup the image prefix in the self.durations dict

y\_pos = self.y\_pos[note[:2]]

tilt = self.tilt[note[2]]

**if** note[-1] == 'n':

**if** y\_pos < self.y\_pos['B5']: # Any note higher than B5 should have it's stem facing downwards

image['note\_img\_offset'] = 18 # Account for height of image file

image['note\_img'] = [pygame.transform.flip(self.images[duration+'note'], True, True),y\_pos-image['note\_img\_offset']] # Flip horizontally and vertically

**else**:

image['note\_img\_offset'] = 118

image['note\_img'] = [self.images[duration+'note'],y\_pos-image['note\_img\_offset']] # Take 118 from y\_pos to account for image height of notes

**elif** note[-1] == 'r':

image['note\_img'] = [self.images[duration+'rest'],350] # The image should always be at y=350 if it is a rest

**if** tilt:

image['tilt'] = [tilt,y\_pos-115]

**return** image

**def** next\_note(self):

'''

        Get the next note in the song as a dictionary of data.

        Returns:

            self.note (dict) : all the data of the note, including loaded image files

            None (NoneType) : special condition returned when self.song is empty

        '''

self.end\_of\_bar = **len**(self.current\_bar)==1 # Boolean of whether self.current\_bar has length of 1

**if** **not** self.current\_bar:

**if** self.song: # When the song has no more notes it will evaluate as False

self.current\_bar = self.song.pop()

**if** **len**(self.current\_bar) == 1:

self.end\_of\_bar = True

**else**:

**return** None

self.note = self.\_parse(self.current\_bar.pop()) # Parse the next note in the song

**return** self.note

**if** \_\_name\_\_ == '\_\_main\_\_':

**print**('Module not execuatable.')

**raise** SystemExit

## ui\_elements.py

**try**:

**import** string

**import** pygame

**from** pygame.**locals** **import** \*

**except** ImportError **as** e: # most likely a ModuleNotFoundError

**raise** Exception('Could not import a module: %s.' % e)

**class** Button:

**def** \_\_init\_\_(self, ctx, text, text\_size, position, dimensions, color, alt\_color, hover\_color, text\_color):

'''

        Create a Button instance.

        Args:

            ctx (\_\_main\_\_.Application) : context of the Application instance required for rendering the button

            text (str) : text to be displayed on the button

            text\_size (int) : text height in pixels

            position (list) : two values stating coordinates of the button

            dimensions (list) : a tuple or list of two values indicating the width and height the rectangular button should be

            color (tuple) : the color of the center of the button

            alt\_color (tuple) : the color of the button border

            hover\_color (tuple) : the color of the center of the button upon mouse hover

            text\_color (tuple) : the color of the button text

        '''

self.ctx = ctx

self.text = text

self.text\_size = text\_size

self.position = position

self.dimensions = dimensions

self.colors = {'primary':color ,

'alt' :alt\_color ,

'hover' :hover\_color,

'text' :text\_color }

# button coords based off the center of the button using (desired\_x - button\_width\*0.5 , desired\_y - button\_height\*0.5)

self.rect = pygame.Rect((**round**(position[0]-(dimensions[0]\*0.5)),**round**(position[1]-(dimensions[1]\*0.5))),

(dimensions[0],dimensions[1])) # create button as a Rect object

self.render(self.colors['primary'])

**def** render(self, color):

'''

        Draw the button onto the display.

        Args:

            color (tuple) : the color of the center of the button

        '''

pygame.draw.rect(self.ctx.screen, self.colors['alt'], self.rect) # draw the button frame

pygame.draw.rect(self.ctx.screen, color, pygame.Rect((self.rect.x+5,self.rect.y+5),

(self.rect.width-10,self.rect.height-10))) # draw the button inside (making a 10 pixel wide frame)

font = pygame.font.Font('.\\assets\\font.ttf', self.dimensions[1]-self.text\_size) # load the correct font

label = font.render(self.text, True, self.colors['text'])

self.ctx.screen.blit(label, (self.position[0]-self.dimensions[0]\*.45,self.position[1]-self.dimensions[1]\*.45)) # render the text in the middle of the button

**def** check(self, mouse\_pos):

'''

        Test whether the button is being hovered over and whether it has been clicked.

        Args:

            mouse\_pos (tuple) : mouse coordinates

        Returns:

            True (bool) : boolean value

            False (bool) : boolean value

        '''

**if** self.rect.collidepoint(mouse\_pos): # if the mouse is over the button

self.render(self.colors['hover'])

**return** True

**else**:

self.render(self.colors['primary'])

**return** False

**def** set\_position(self, position):

'''

        Set the position of a button.

        Args:

            position (list) : two values stating coordinates of the button

        '''

self.position = position

self.rect = pygame.Rect((**round**(position[0]-(self.dimensions[0]\*0.5)),**round**(position[1]-(self.dimensions[1]\*0.5))),

(self.dimensions[0],self.dimensions[1]))

**class** ScrollBar:

**def** \_\_init\_\_(self, ctx, dimensions, position, scroll\_length, color, alt\_color, clicked\_color, scroll\_position=0):

'''

        Create a scroll bar instance.

        Args:

            ctx (\_\_main\_\_.Application) : context of the Application instance required for rendering the scroll bar

            dimensions (list) : width and height of the scroll bar

            position (list) : two values stating the coordinates of the scroll bar

            scroll\_length (int) : number of pixels needed to be scrolled

            color (tuple) : slider track color

            alt\_color (tuple) : slider thumb when not clicked color

            clicked\_color (tuple) : slider thumb when clicked color

            scroll\_position (int) : how far along the scroll bar notch is (assumed to never be larger than the scroll length)

                                                     default: 0

        '''

self.ctx = ctx

self.dimensions = dimensions

self.position = position

**if** scroll\_length <= dimensions[0]: # if the scrollable area is less than the width of the scroll bar track, set to the scroll bar track width (no scrolling needed)

self.scroll\_length = dimensions[0] + 1 # add 1 to avoid a division by 0 error

**else**:

self.scroll\_length = scroll\_length

self.scroll\_position = scroll\_position

self.colors = {'primary':color ,

'alt' :alt\_color ,

'clicked':clicked\_color}

self.scroll\_bar\_track = pygame.draw.rect(ctx.screen, color, pygame.Rect((**round**(position[0]-(dimensions[0]\*0.5)),**round**(position[1]-(dimensions[1]\*0.5))), # adjust from coords based on middle of object to top left

(dimensions[0],dimensions[1])))

# the size of the scroll notch is: (size of scroll bar) / ( (scrolling length) \* (size of scroll bar) )

self.scroll\_bar\_thumb = pygame.draw.rect(ctx.screen, alt\_color, pygame.Rect((self.scroll\_bar\_track.x+scroll\_position,self.scroll\_bar\_track.y),

(self.scroll\_bar\_track.width/(scroll\_length+1000)\*self.scroll\_bar\_track.width,self.scroll\_bar\_track.height)))

self.was\_scroll\_bar\_clicked = False # track whether the thumb was clicked last cycle

# scroll amount per pixel change in thumb: (scrolling length) / ( ( (x coord of the right most wall of the scroll bar track)-(thumb width) )-(x position of scroll bar track) )

# or, the scrolling length divided by the number of pixels the scroll bar can travel

**try**:

self.scroll\_amount = scroll\_length/((self.scroll\_bar\_track.right-self.scroll\_bar\_thumb.width)-self.scroll\_bar\_track.x)

**except** ZeroDivisionError:

self.scroll\_amount = 0

**def** render(self, thumb\_color):

'''

        Draw the scroll bar and notch onto the display.

        Args:

            thumb\_color (tuple) : the color of the thumb

        '''

pygame.draw.rect(self.ctx.screen, self.colors['primary'], self.scroll\_bar\_track)

pygame.draw.rect(self.ctx.screen, thumb\_color, self.scroll\_bar\_thumb)

**def** check(self, mouse\_pos, mouse\_clicked):

'''

        '''

**if** self.was\_scroll\_bar\_clicked:

**if** mouse\_clicked:

self.scroll\_bar\_thumb = self.scroll\_bar\_thumb.move(mouse\_pos[0]-self.last\_mouse\_pos[0], 0) # move the thumb by the change in mouse x-axis position

self.scroll\_position -= self.scroll\_amount\*(mouse\_pos[0]-self.last\_mouse\_pos[0]) # change the scroll position by the scroll amount times the same change

**if** self.scroll\_bar\_track.contains(self.scroll\_bar\_thumb):

self.render(self.colors['clicked'])

**else**:

self.scroll\_bar\_thumb = self.scroll\_bar\_thumb.move(-(mouse\_pos[0]-self.last\_mouse\_pos[0]), 0) # undo the above thumb and scroll position changes

self.scroll\_position += self.scroll\_amount\*(mouse\_pos[0]-self.last\_mouse\_pos[0])

**else**:

self.render(self.colors['alt'])

self.was\_scroll\_bar\_clicked = False

**elif** self.scroll\_bar\_thumb.collidepoint(mouse\_pos) **and** mouse\_clicked: # if the scroll bar thumb has been clicked

self.was\_scroll\_bar\_clicked = True

self.last\_mouse\_pos = mouse\_pos # keep track of the mouse position of the last cycle

**def** get\_notch\_position(self):

**return** self.scroll\_position

**class** SongTab:

**def** \_\_init\_\_(self, ctx, start\_pos, song=['name','difficulty','location'], highscore=-1):

self.ctx = ctx

self.start\_pos = start\_pos

self.position = start\_pos

self.song = song

self.button = Button(self.ctx, text='play', text\_size=22, position=[self.position+125,400], dimensions=[160,75], color=(213,240,247), alt\_color=(0,162,232), hover\_color=(58,186,218), text\_color=(0,0,0))

self.highscore = highscore

**def** render(self):

pygame.draw.rect(self.ctx.screen, (0,162,232), (self.position, 115, 250, 350))

pygame.draw.rect(self.ctx.screen, (153,217,234), (self.position+5, 120, 240, 340)) # 5 pixels smaller to create a border

font\_40 = pygame.font.Font('.\\assets\\font.ttf', 40) # size 40 text

font\_20 = pygame.font.Font('.\\assets\\font.ttf', 20) # size 20 text

name = font\_40.render(self.song[0], True, (0,0,0)) # create the song name text

diff = font\_20.render(self.song[1], True, (0,0,0)) # create the song difficulty text

self.ctx.screen.blit(name, (self.position+10,120)) # render all the text

self.ctx.screen.blit(diff, (self.position+10,170))

**if** self.highscore >= 0:

highscore = font\_20.render(f'Highscore: {self.highscore}%', True, (0,0,0))

self.ctx.screen.blit(highscore, (self.position+10,200))

**def** set\_x(self, x):

self.button.set\_position([self.start\_pos+125+x,400])

self.position = self.start\_pos + x

**def** get\_x(self):

**return** self.position

**class** TextInput:

**def** \_\_init\_\_(self, ctx, dimensions, position, character\_limit, allowed\_characters, color, active\_color, input\_hidden=False):

self.ctx = ctx

self.dimensions = dimensions

self.position = position

self.character\_limit = character\_limit

self.colors = {'inactive':color ,

'active' :active\_color}

self.rect = pygame.Rect(position, dimensions)

self.allowed\_characters = allowed\_characters

self.input\_hidden = input\_hidden

self.is\_active = False

self.cursor = None

self.value = []

self.cursor\_position = 0

**def** render(self):

**if** self.is\_active:

pygame.draw.rect(self.ctx.screen, self.colors['active'], self.rect)

**else**:

pygame.draw.rect(self.ctx.screen, self.colors['inactive'], self.rect)

font = pygame.font.Font('.\\assets\\font.ttf', self.dimensions[1])

**if** self.input\_hidden:

text = font.render('•'\***len**(self.value), True, (0,0,0))

**else**:

text = font.render(''.join(self.value), True, (0,0,0))

self.ctx.screen.blit(text, (self.rect.left,self.rect.top-(.2\*self.dimensions[1])))

**def** check(self, mouse\_pos, mouse\_clicked):

**if** self.rect.collidepoint(mouse\_pos):

self.cursor = pygame.cursors.**compile**(pygame.cursors.textmarker\_strings)

pygame.mouse.set\_cursor((8,16),(0,0),\*self.cursor)

**if** mouse\_clicked:

self.is\_active = True

**else**:

**if** self.cursor:

pygame.mouse.set\_cursor(\*pygame.cursors.arrow)

self.cursor = None

#### **if** mouse\_clicked:

self.is\_active = False

self.render()

**def** key\_press(self, key, key\_type='char'):

**if** self.is\_active:

**if** key\_type == 'char':

**if** key == '\x08': # Backspace

**try**: # If value is empty an index error will be raised if a del is performed on it

**del** self.value[self.cursor\_position-1]

self.cursor\_position -= 1

**except** IndexError: # Catch redundant error

**pass**

**elif** **len**(self.value) < self.character\_limit: # Only continue if the current length of the input is less than the maximum allowed length

**if** key **in** self.allowed\_characters: # Make sure the character is valid for the input

self.value.insert(self.cursor\_position, key) # Insert the character at the cursor position

self.cursor\_position += 1

**if** key\_type == 'action':

**if** key == 276 **and** self.cursor\_position > 0: # Left arrow

self.cursor\_position -= 1

**elif** key == 275 **and** self.cursor\_position < **len**(self.value): # Right arrow

self.cursor\_position += 1

**elif** key == 127: # Del key

**try**: # If value is empty an index error will be raised if a del is performed on it

**del** self.value[self.cursor\_position]

**except** IndexError: # Catch redundant error

**pass**

**return**

**def** get\_value(self):

**return** ''.join(self.value)

**if** \_\_name\_\_ == '\_\_main\_\_':

**print**('Module not execuatable.')

**raise** SystemExit

## user.py

**try**:

**import** os

**import** json

**except** ImportError **as** e: # most likely a ModuleNotFoundError

**raise** Exception(f'Could not import a module: {e}.')

**class** User:

'''

    User class to handle song performance storing.

    Args:

        path (str) : indicates location of JSON user file

    Returns:

        self.data (dict) : all user and song data

    '''

**def** \_\_init\_\_(self):

self.data = None # Upon initializing create an empty dict

self.path = None

**return**

**def** \_load\_data(self):

**with** **open**(self.path, 'r') **as** **file**:

self.data = json.load(**file**)

**return**

**def** validate(self, username, password):

**if** **not** username **or** **not** password:

**return** 'Both fields must be filled in.'

directory = [[files,root] **for** root, dirs, files **in** os.walk('.\\assets\\users\\')]

**if** **len**(directory) == 1 **and** **not** directory[0][0]:

**return** 'User does not exist.'

**else**:

**for** **file** **in** directory[0][0]:

**if** username **in** **file**:

self.path = directory[0][1]+**file**

self.\_load\_data()

**if** password == self.data['password']:

self.username = username

**return**

**else**:

self.path = None

self.data = {}

**return** 'Invalid password.'

**return** 'User does not exist.'

**def** create(self, username, password):

**if** **not** username **or** **not** password:

**return** 'Both fields must be filled in.'

error = self.validate(username,password)

**if** error == 'User does not exist.':

self.path = f'.\\assets\\users\\{username}.json'

self.username = username

self.save({'password':password})

**return**

**else**:

**return** 'User already exists.'

**def** remove(self):

os.remove(self.path)

**return**

**def** save(self,data):

self.data = data

**with** **open**(self.path, 'w') **as** **file**:

json.dump(data, **file**, indent=4)

**return**

**def** get\_data(self):

**if** self.path:

self.\_load\_data()

**return** self.data

**else**:

**return** None

**def** get\_username(self):

**if** self.path:

**return** self.username

**else**:

**return** None