Mp3 Project Report

Student:
Callista Roselynn

Teacher:
Sir Jude Martinez

Course:
Program Design Method

Project Specifications:

Purpose:

This document is intended as a guide to overall gist and further information regarding the recreation of this media player. This project will make use of Python coding in hopes of producing a working program.

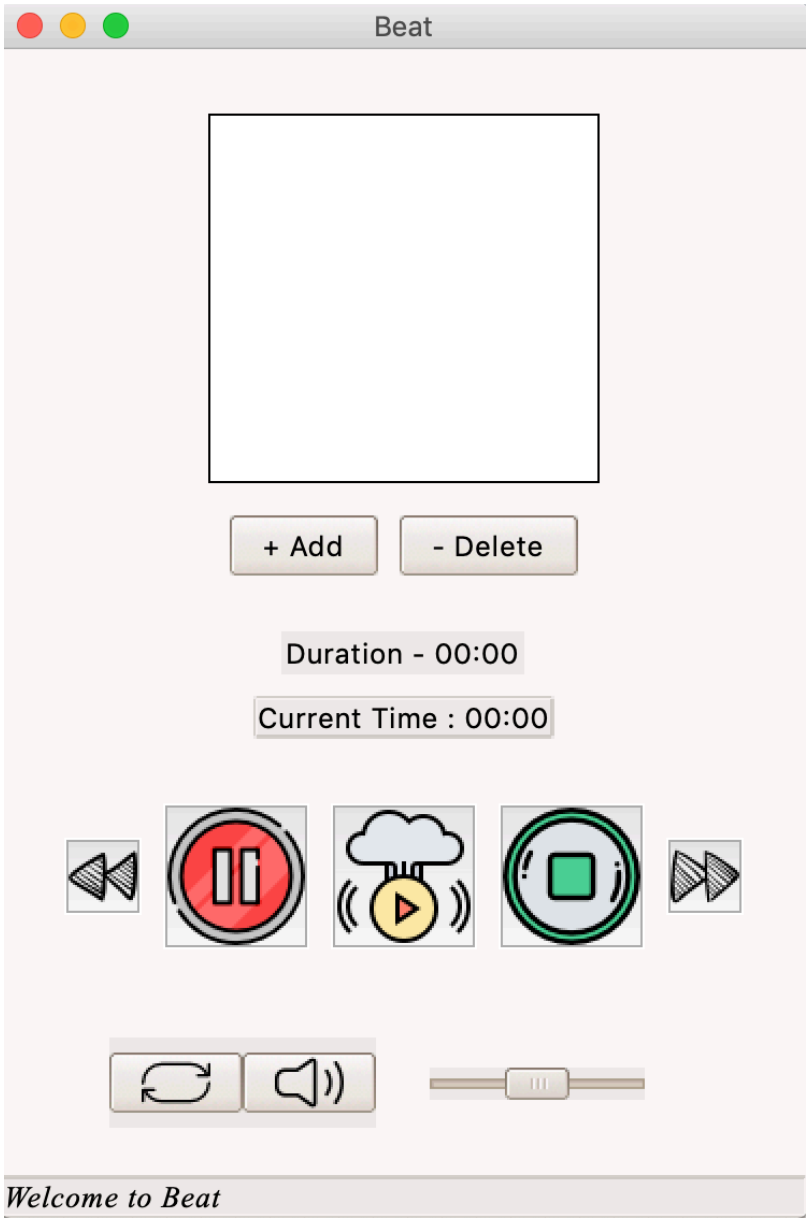
Scope:

In this project, I will be recreating the Mp3 players called Beat. This system should utilize Python libraries and modules to do the actual processing of media files on the system. Beats should function just like a Mp3 Player should. It must be capable of loading audio files and playing those files as well as display the menu which includes fully functioning buttons in a pop-up window. By implementing these functionality using Python libraries, I hope to gain new knowledge regarding some of the various modules that Python has to offer.

Modules used within the project:

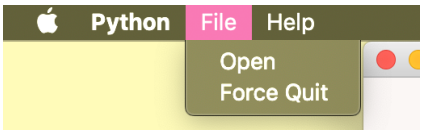
- Tkinter – A framework which is used to create the overall GUI of the MP3 player
    - Tk – creates a window
    - Mainloop – makes sure to keep creating the window so the window doesn't disappear
    - Destroy – destroys the window to exit
    - Filedialog –
    - Messagebox - creates a pop-up window containing the message you wish to display
- Ttkthemes – A package used to integrate themes into the GUI
- Pygame – A framework for games used to handle audio files mostly in .wav format.
    - Music.load – loads the music
    - Music.pause – pauses the music
    - Music.play – plays the music
    - Music.unpause – resumes the music
- Mutagen – Used to deal with big files such as (.mp3) file formats
- Threading – To let the current time run at the same time as the other code throughout the program
- Time – Used to display the duration and remaining time as well as let the program buffer
- OS - Used to display filename without path as well as to determine the file type.

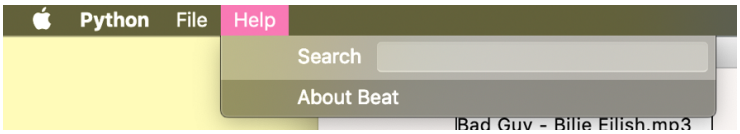
Evidence of Working Program:



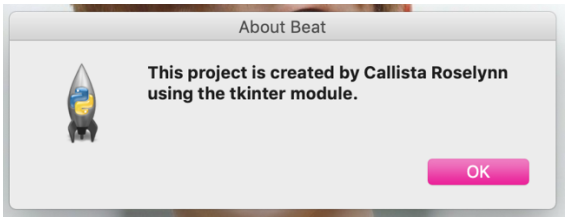
This is how the overall layout of what the app looks like

The menubar






```
"""creating the menubar"""
menubar = Menu(window)
window.config(menu=menubar)
# ensures that the menubar appears at the top and prepares the menubar so it can receive submenus
```

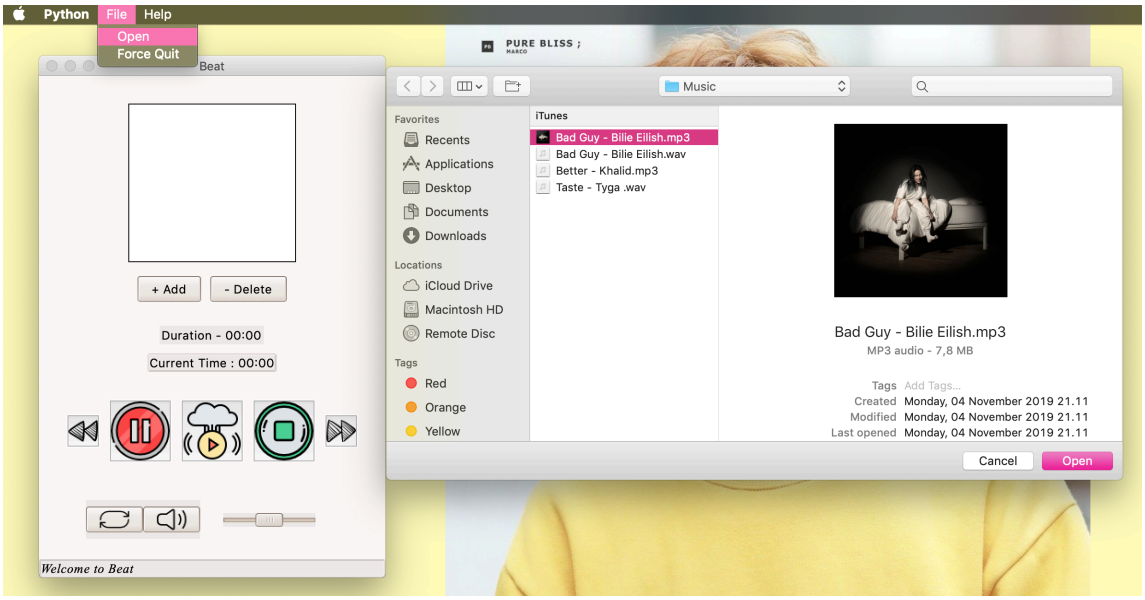



```
"""creating the submenu"""
subMenu = Menu(menubar, tearoff=0)  # using menubar instead of window to differentiate the two types of menu
menubar.add_cascade(label="File", menu=subMenu)  # cascade = dropdown menu
subMenu.add_command(label="Open", command=browse)
subMenu.add_command(label="Force Quit", command=window.destroy)


def about_beat():
    tkinter.messagebox.showinfo("About Beat", "This project is created by Callista Roselynn using the tkinter module.")


subMenu = Menu(menubar, tearoff=0)  # using menubar instead of window to differentiate the two types of menu
menubar.add_cascade(label="Help", menu=subMenu)  # cascade = dropdown menu
subMenu.add_command(label="About Beat", command=about_beat)
```

The menubar is created using the Menu class available on the tkinter package. It receives a parameter of window to indicate where the menubar should appear. I then congigured this feature into a menubar to ensure that it appears at the top of the window. The submenu is created in a miliar way as the menubar with only slight differences. First of all, instead of typing window in the Menu paramter for the submenu, I instead typed menubar and gave a tearoff value of 0. This differentiates the two types of menu and makes sure that the submenu does appear on the menubar.

The .add_cascae() function creates a dropdown type of menu like typically seen on any window. The .add_command() function links the dropdown menu with a certain action to tell the menu what to do when the user clicks on it. In the case of “Force Quit”, the command window.destroy() literally destroys the window and immediately closes your window.

Opening File




```
def open_file():
    global filename
    # you can now use the variable name wherever you want throughout the code
    filename = filedialog.askopenfilename()
    add_music(filename)
```

The opening of a file is initiated through the function called browse. Here, the user is prompted to choose a file on their device and will automatically retrieve the name of the file. It will then call upon the add_music () function which essentially adds and displays the music in the playlist listbox.

Add and Delete button

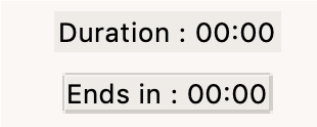
```
index = 0


def add_music(fname):

    global index
    # creating a listbox
    fname = os.path.basename(fname)  # fname = filename without path
    playlist.insert(index, fname)  # adding song to playlist
    play_List.insert(index, filename)
    playlist.pack(pady=15)


def delete_music():
    selection = playlist.curselection()
    selection = int(selection[0])
    playlist.delete(selection)  # deletes music from listbox
    play_List.pop(selection)  # deletes music from list of songs because it takes up space
    #  .remove needs the fname
```

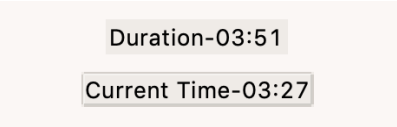
As mentioned before, the add_music() button is used to add songs to the listbox by using.insert(). It also adds the song to a list called play_List. The difference of the list and listbox lies in the name of the file. Whereas the list contains the full path of the file, the listbox only contains and displays the filename.

The delete method is similar to that of the play functionality where it takes into account the song being selected the listbox and deleting that certain song from both the listbox by using .delete() and .pop() in the list. .remove() is not being used in this case for the list this function requires 1 parameter called the fname, however I only take into account the index of the song being selected I the listbox and not the name of the song.

TIME CLASS

Duration and Current Time






```python
class Time:
    def __init__(self):
        self.paused = False

    def music_duration(self, play_selection):

        file_type = os.path.splitext(play_selection)
        # splits the file path into a list, 1st index in list is the extension

        if file_type[1] == ".mp3":  # for .mp3 file formats:
            audio = MP3(play_selection)
            total_duration = audio.info.length

        else:  # for .wav file formats:
            x = mixer.Sound(play_selection)  # loads the music file
            total_duration = x.get_length()  # calculates the value of the music in seconds

        # getting the length in minutes and seconds
        minutes, seconds = divmod(total_duration, 60)
        minutes = round(minutes)
        seconds = round(seconds)  # round function rounds up the values
        duration = "{:02d}:{:02d}".format(minutes, seconds)
        # formatting how the minutes and seconds will appear
        # 2d = 2 decimal places, 0 is added in case of 1 digit
        durationLabel["text"] = "Duration" + "-" + duration

        timee = threading.Thread(target=self.count_time, args=(total_duration,))
        # args needs to be in tuple/list form
        # threading enables the function to work simultaneously as the other functions
        timee.start()
        # print(file_type[1]) just to show the extension is ghe first index

    def count_time(self, current_time):

        global durationLabel

        # mixer.music.get_busy(): Returns false value when we stop, exits the while loop (music stops)
        # get_busy also causes the thread to quit, stop and destroy itself
        while current_time and mixer.music.get_busy():
            if self.paused:
                continue
                # continue ignores all the statement under else
            else:
                minutes, seconds = divmod(current_time, 60)
                minutes = round(minutes)
                seconds = round(seconds)
                duration = "{:02d}:{:02d}".format(minutes, seconds)
                remaining_Label["text"] = "Ends in" + "-" + duration
                time.sleep(1)  # subtract per second
                current_time -= 1  # subtracts 1 second from the total duration when music plays
```

The Duration of the music is controlled by the music_duration() function while the Current Time is counted through the count_time() function.

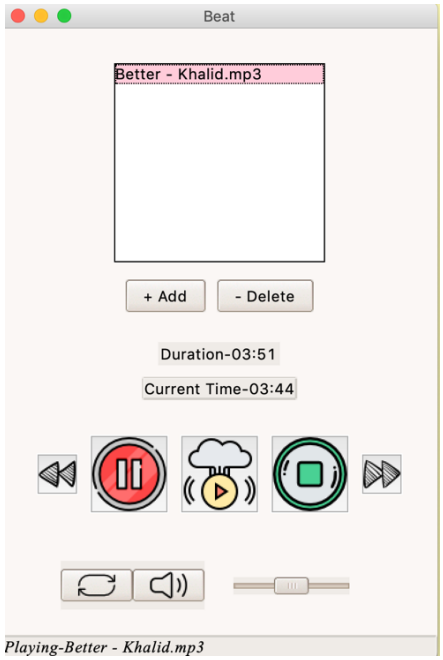
In music_duration(), the parameter self and play_selection is received. The variable file_type stores the extension of the file and determines whether the file being selected is a '.mp3' or a '.wav' file format. In this function, the if-else statement is being used. The if section is executed if the file selected is a '.mp3' file format while the else executes when it is a '.wav' format. The audio in the if section gets the metadata of the selected file and using audio.info.length(), the total length of the song was obtained. In the else statement, we use mixer in the pygame module as well as the .get_length() to do so. Aside from that, I also implemented threading in the mp3 player to be able to display how much time is remaining until the song finish.

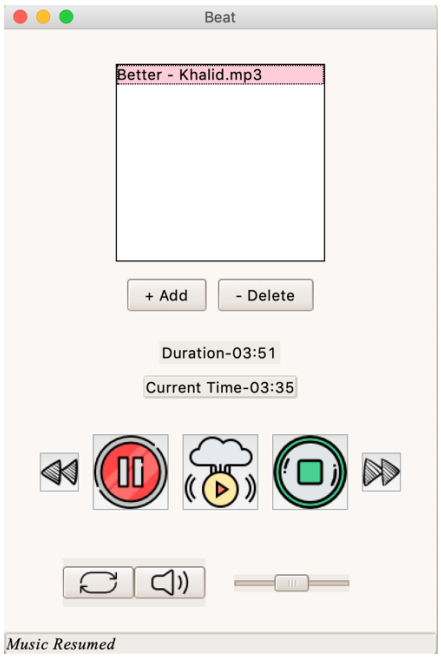
After getting the total duration of the song, I then format the length into minutes and seconds and updated the durationLabel to display the duration of the selected song in minutes and seconds.

In count_time(), we use the while statement as well as the if-else statement. The mixer.get_busy() in the while statement is used to return a false value when we click the stop button which therefore causes the code to exit the while loop. The get_busy() function also causes the thread to quit and destroy itself and hence the value displayed in the remaining_Label will no longer continue to count down. The if statement in this function checks whether the music is being paused or not. If the music is being paused, the code passes the function and goes to the next step. This action is done using the keyword 'continue' found in python. The else statement, will convert the value of the thread once again into minutes and seconds using divmod to get the remainder (seconds) and is once again formatted using "{02d}:{02d}" to properly display the value of the remaining duration of the song. However, instead of displaying this value in the durationLabel like in the music_duration(), this value will be diaplyed in the remaining_Label() instead. The line "time.sleep(1)" ensures that the value is decreasing by one second and the line "current_time -= 1" ensures that the value, 1 second, is being subtracted from the total_duration onf the selected song.

ACTION CLASS:

Play and Resume



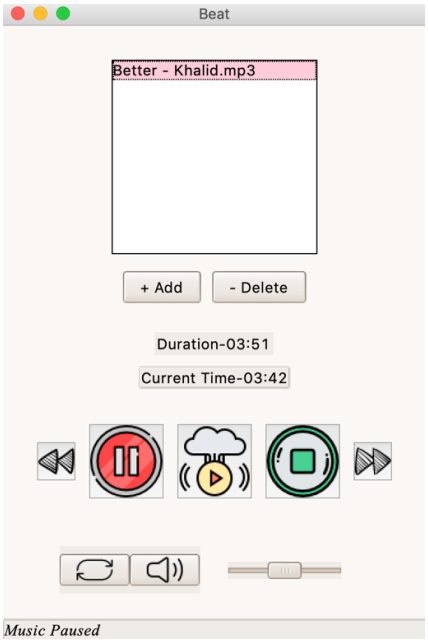



```python
class Action:

    def __init__(self):
        self.mute = False
        self.statusbar = statusbar
        self.index = index

    def play(self):

        global paused

        if paused:
            mixer.music.unpause()
            self.statusbar["text"] = "Rezoom"
            # executed when paused is initialized. It resumes the music from the moment of paused.
            paused = False
        else:  # executed when paused not initialized
            self.stop()
            time.sleep(1)
            # lets the mp3 player buffer for 1 second to switch music if we don't click stop b4 switching
            selection = playlist.curselection()
            # identifies which music you are selecting in the playlist
            # Returns the value of the index in the listbox in a tuple
            selection = int(selection[0])  # type casted the tuple into a int value to match index
            play_selection = play_List[selection]  # retrieve song and store it under play_selection
            mixer.music.load(play_selection)
            # loads any music file you choose to open
            # requires the whole path name to load
            mixer.music.play()
            self.statusbar["text"] = "Playing music" + "-" + os.path.basename(play_selection)
            # edits the label for status bar to playing music + filename
            # os.path.basename removes the path of the file and displays only the name of the file
            Time().music_duration(play_selection)
```

Using the “if-else” statement, the window can check whether the pause button have been clicked by the user or not. In the case where the user clicked the pause button, the code will go to the “if” statement where mixer will unpause the music from the moment of pause through the command mixer.music.unpause(). It will then switch the value of paused back to False to allow the user to pause the song multiple times while running this code.

In the case where the pause function have not been used, the play function will automatically go to the “else” statement. Here, I used curselection() to get the value of the user’s chosen song on the playlist.. Since this function returns a value in tuple form, I typecasted it to become an integer value to match the type of value of the index. The selectod song is then stored in play_selection variable and the app proceeds to load and play that particular file with the help of pygame mixer module. The status of the statusbar is then updated to also display the name of the song by splitting the filepath and only displaying the basename of the file instead of the enitre path. T the music duration() function have also been called to get the total duration of only the song being selected.

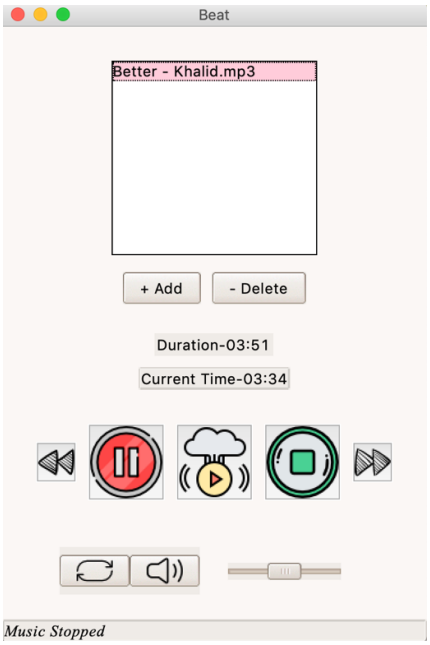
Pause




```
def pause(self):

    global paused

    paused = True  # not needed
    mixer.music.pause()
    self.statusbar["text"] = "le Music is Paused"
```

In this function, I changed the value of paused to True when paused button is clicked, as well as, switch the text that appears on the statusbar using the statusbar[] function. Using mixer from the pygame package, I then pause the music through the command mixer.music.pause().

Stop




```
def stop(self):

    mixer.music.stop()
    self.statusbar["text"] = "Don't Stop me Now! :("
    # edits the label for status bar to music stopped
```

For this feature, I simply make use of the music.stop() command in the pygame mixer module and altered the status of the statusbar.

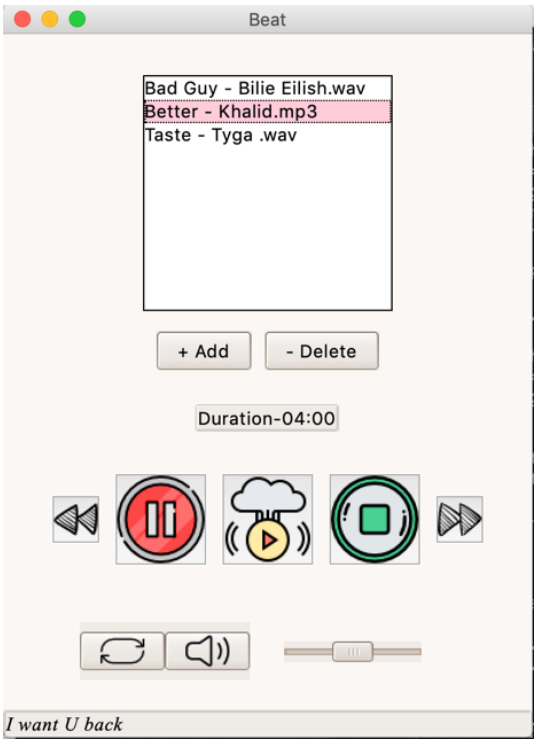
Mute button

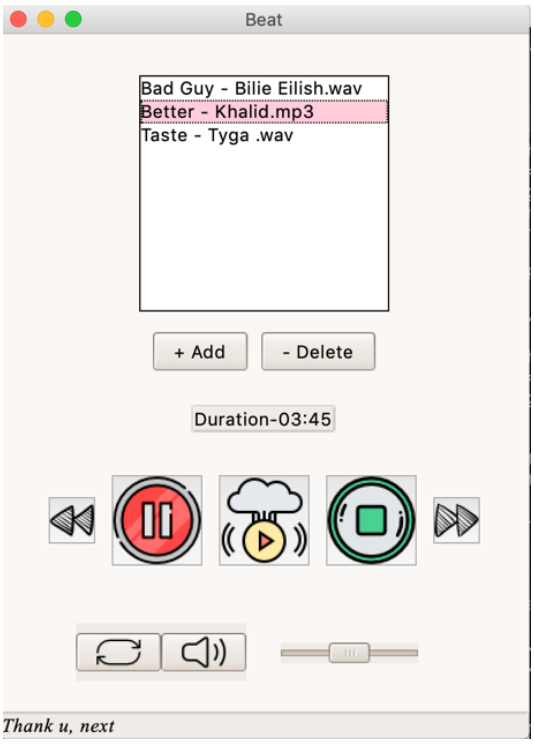
```
def muted(self):

    global previousVol
    global volPic
    global volumeButton
    global mutePic

    if self.mute:  # un-mute Music
        mixer.music.set_volume(previousVol*0.01)  # to make it easier to interpret value
        volumeButton.configure(image=volPic)
        volume.set(previousVol)  # sets the volume back to previous setting
        self.mute = False
    else:  # mute music
        mixer.music.set_volume(0)  # mute music
        volumeButton.configure(image=mutePic)  # changes the image when clicked
        previousVol = volume.get()  # gets the value of volume before muted
        volume.set(0)
        self.mute = True
```

For the muted function, I make use of the “if-else” statement. “If” section checks whether the music is mute and proceeds to un-mute the music when clicked. This is done by setting the

The “else” section on the other hand, will proceed if the mute button have not been clicked and will mute the music in the app. This is done by setting the value in the scalebar to zero and changing the logo of the mute button. It then sets the value of mute back to True so that users will be able to mute and un-mute as they please.

Next and Previous button



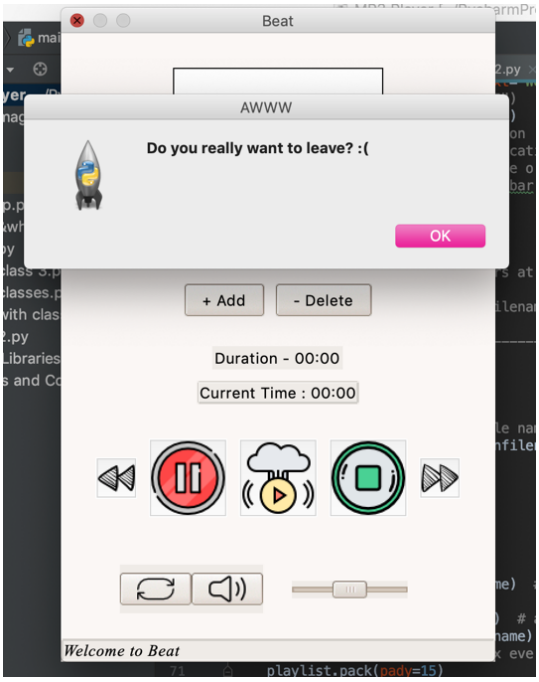



```
def next_music(self):

    global play_List

    self.stop()
    time.sleep(1)
    self.index += 1
    nextt = play_List[self.index]
    mixer.music.load(nextt)
    mixer.music.play()
    self.statusbar["text"] = "Thank u, next"
    Time().music_duration(nextt)
```

```
def previous_music(self):

    global play_List

    self.stop()
    time.sleep(1)
    self.index -= 1
    previouss = play_List[self.index]
    mixer.music.load(previouss)
    mixer.music.play()
    self.statusbar["text"] = "I want U back"
    Time().music_duration(previouss)
```

The next button is associated to the function called next_music where it receives the parameter of self. Here, global play_List allows us to use this variable throughout our code as it is located outside the class and is not in another function. For this feature, I first call on the stop() functino in the Action class to stop the music and initiate time.sleep(1) to let the app buffer for one second just like how I did for the play/resume button. I then increase the index by 1 to get the index of the next song on the list and instead of playing the song selected by the user, Beat will now load and play the next song on the list and is stored in the variable called nextt. I then changed the text of the statusbar once again and last but not least, I made sure that the time in the music duration will change accordingly. The previous button works in the same exact way as the next button does and implement all the same code and functionality except, instead of increasing the number of index, it decreases the index value by 1 therefore getting the data of the previous song and storing it In the variable called previouss.

The x button




```
# overwriting the x button in the window
def quitting():
    tkinter.messagebox.showinfo("AWWW", "Do you really want to leave? :(")
    stop()  # stops music if playing
    window.destroy()  # quits the window without giving error


window.protocol("WM_DELETE_WINDOW", quitting)
```

To overwrite the x button, I used protocol handlers, where protocol mostly refers to the interaction between the window itself and the window manager, to tell the window manager how the x button will function. In this case,
the protocol handler called WM_DELETE_WINDOW is being ued to define what happens when the user decides to close the window using the window manager (the x button).

References:

- https://docs.python.org/3/library/tkinter.ttk.html
- https://ttkthemes.readthedocs.io/en/latest/theming.html#choosing-a-theme
- https://www.daniweb.com/programming/software-development/threads/205335/my-first-program-mp3-player
- https://stackoverflow.com/questions/28551948/tkinter-look-theme-in-linux
- https://www.javatpoint.com/python-tkinter-spinbox