COMP1752
Object-Oriented Programming Coursework Report

Author: Nguyen Quoc Bao – ID: 001436167

Link Github: https://github.com/Brian-ng05/COMP1752

Table of Contents

I. Introduction

In this report, I will simulate the JukeBox application in Python with the help of the Tkinter library. Focusing on the core principles of Object-Oriented Programming (OOP) in the real project. This simple music player application enables users to view their music library, create playlists, update ratings, simulate music playback, and receive notifications through the user interface. Some additional development features include sorting, searching by song name, and filtering by artist name. The project also includes input validation and emphasizes usability.

II. Design

1. Overview

The system consists of multiple GUIs that interact with a common music library. In this library, data is stored in a dictionary that maps string-based song IDs to LibraryItem objects. The design ensures each module interacts independently while sharing functionality through imported utilities and data accessors.

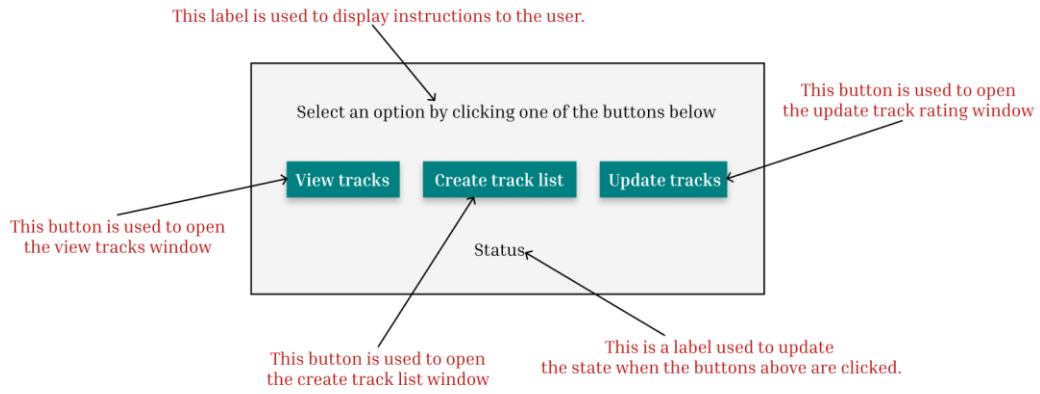
2. Project Structure

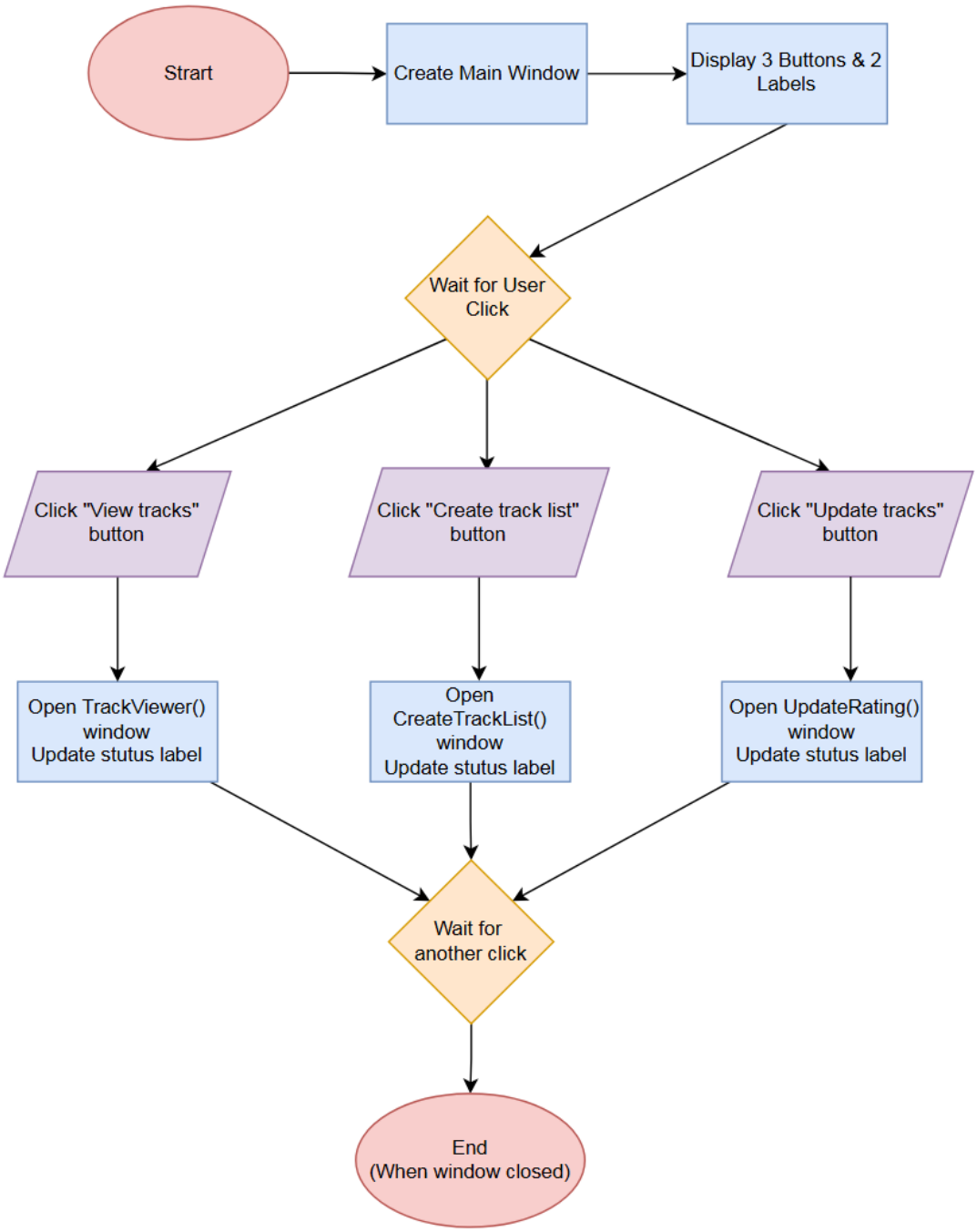
- *track_player.py*: Main UI to navigate sub-features.
- *view_tracks.py*: Display all track, allows filtering, sorting, and searching.
- *create_track_list.py*: Enables playlist creation, saving/loading, and simulated playback.
- *update_tracks.py*: Allows rating updates for tracks.
- *track_library.py*: Stores track data and provides methods.
- *library_item.py*: Defines the *LibraryItem* class.
- *playlist.json*: Stores saved playlists.
- *font_manager.py*: Provides font configure for consistency across the UI.

  Supplemented by a set of test files:
- *test_library_item.py*: Unit tests for individual track functions (e.g., info, starts, constructor).
- *test_track_library.py*: Unit tests for individual track functions (e.g., name, artist, rating, play count).

3. UI Design

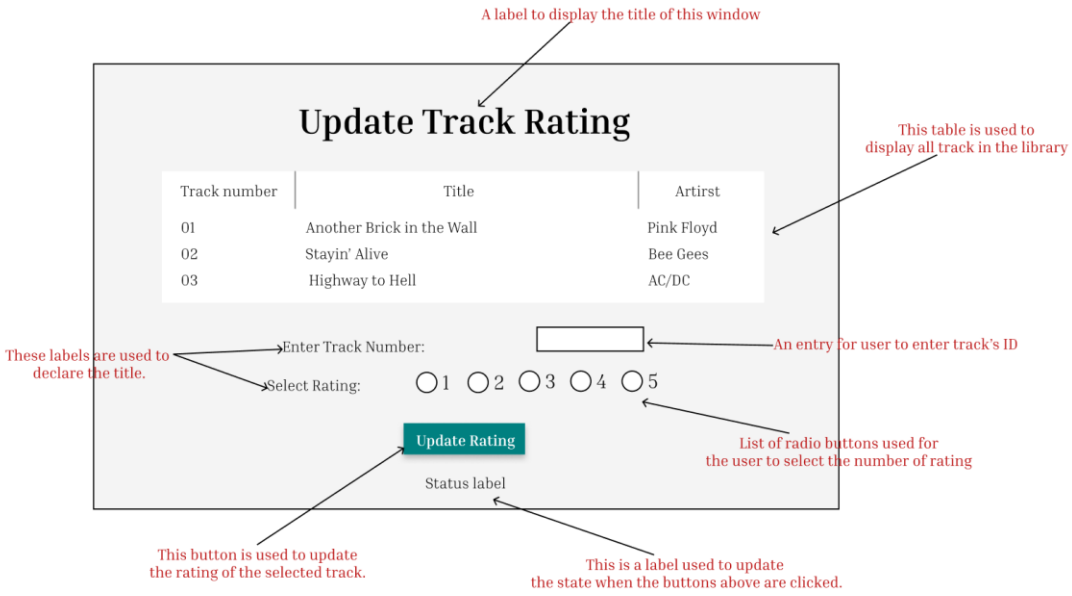
- **Main Window**

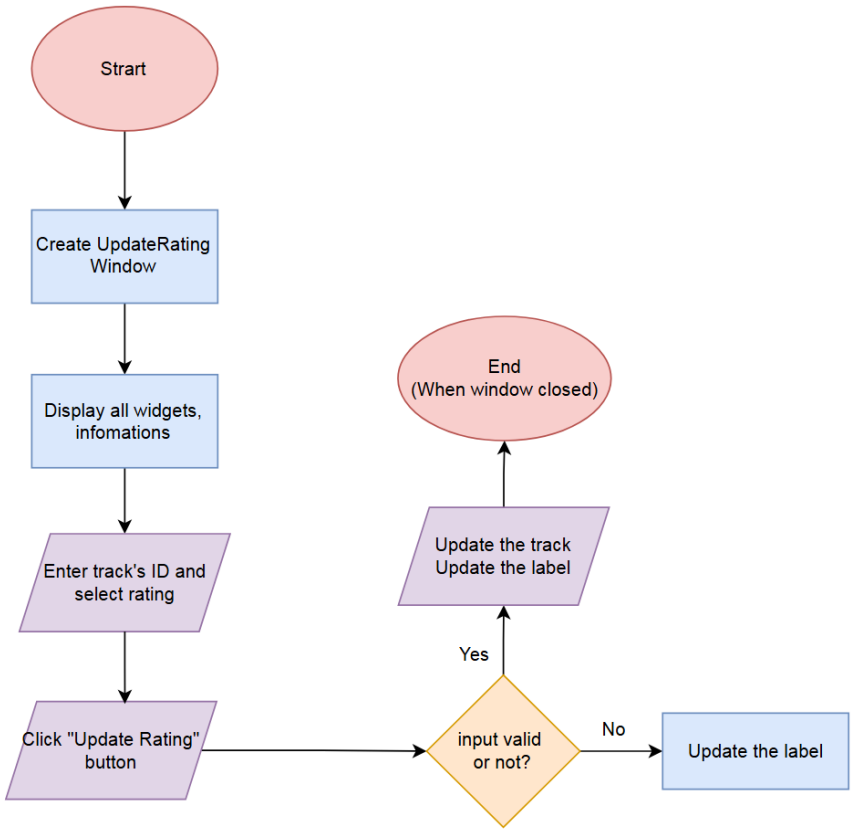




This is a Flowchart for the main window of the program

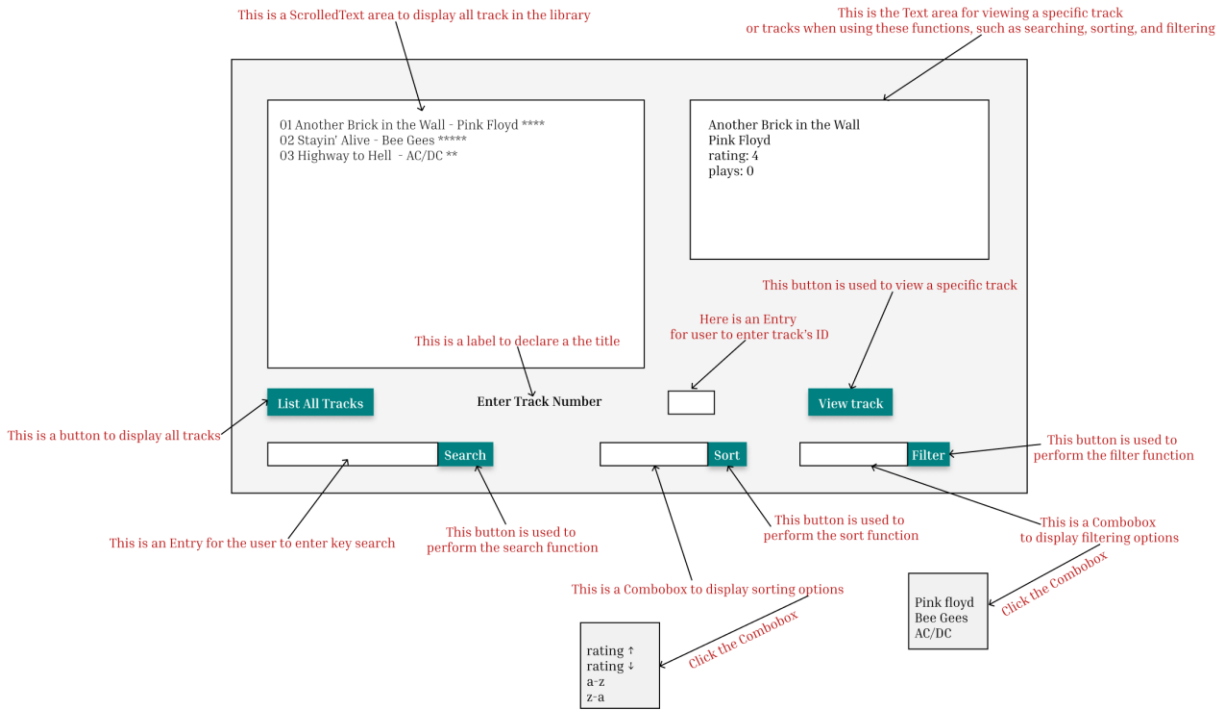
- **Update Track rating Window**

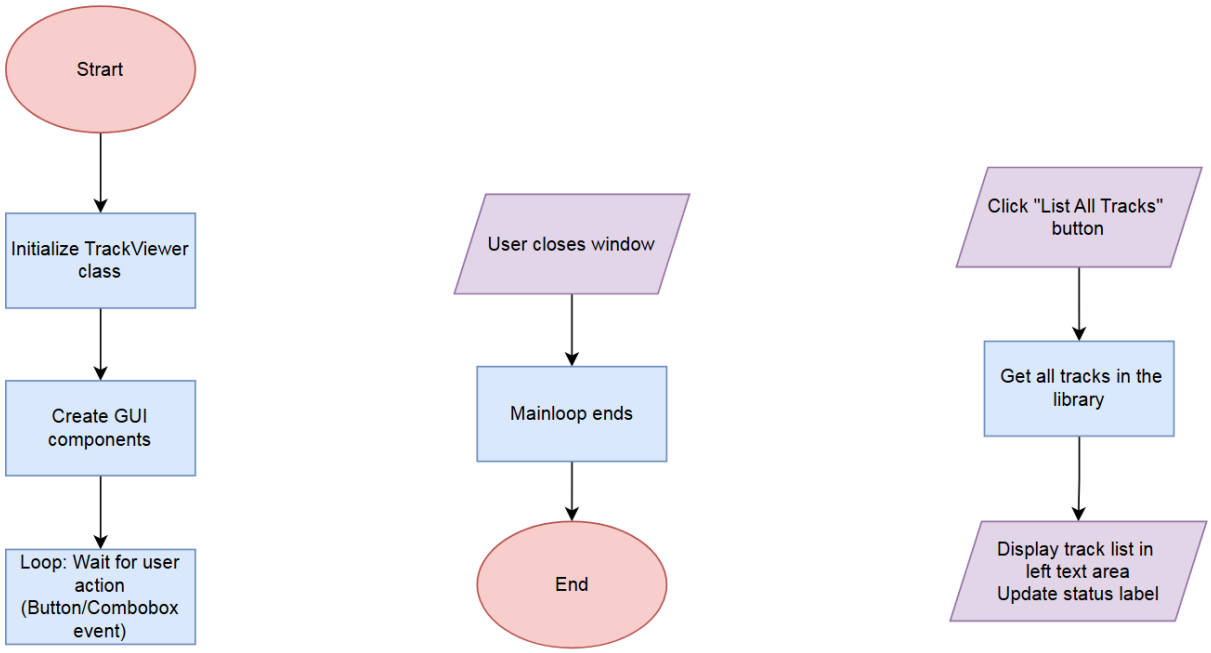




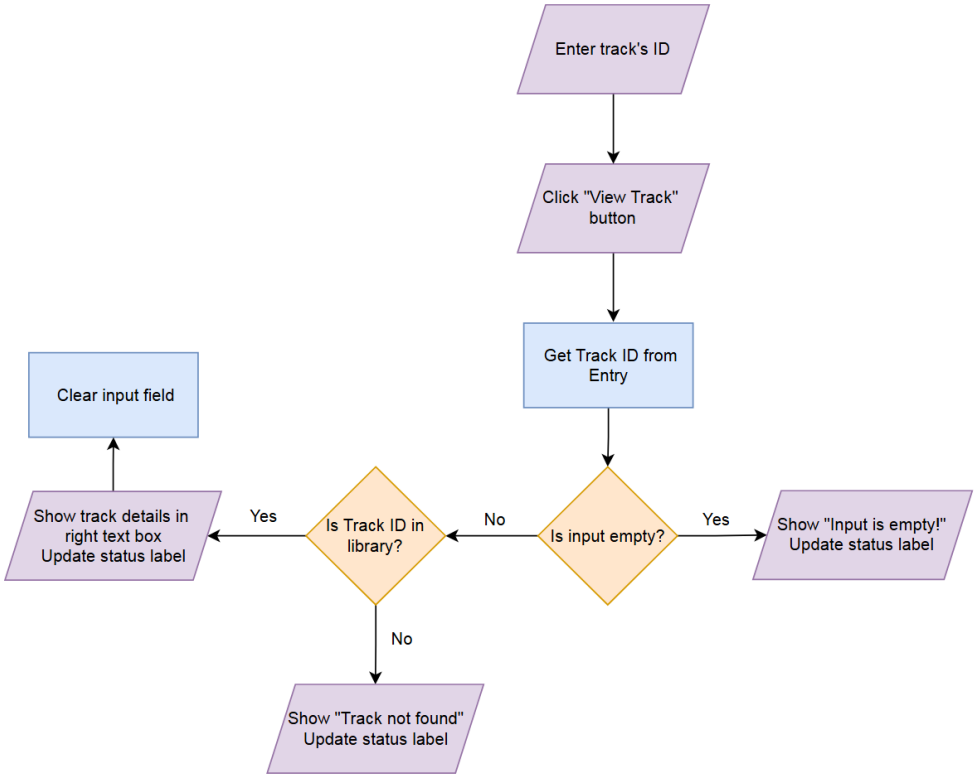
This is a Flowchart for Update Track Rating window

- **Track Viewer Window**

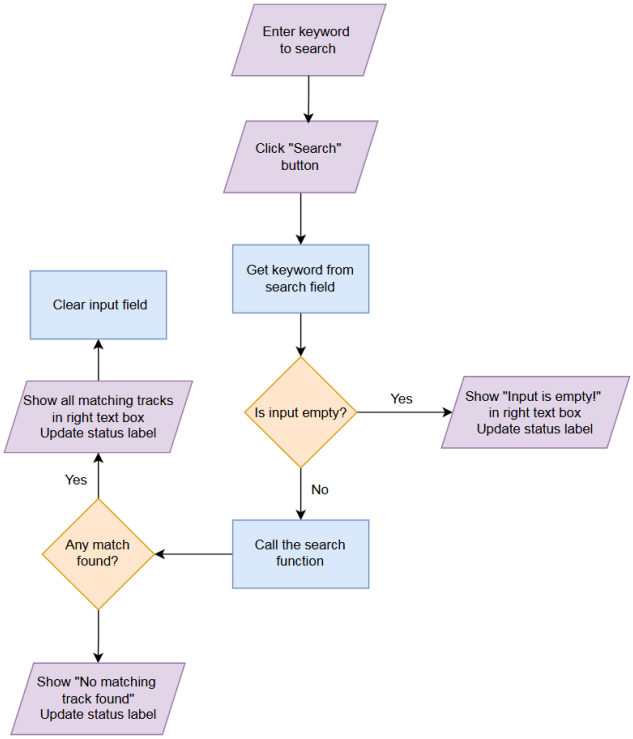




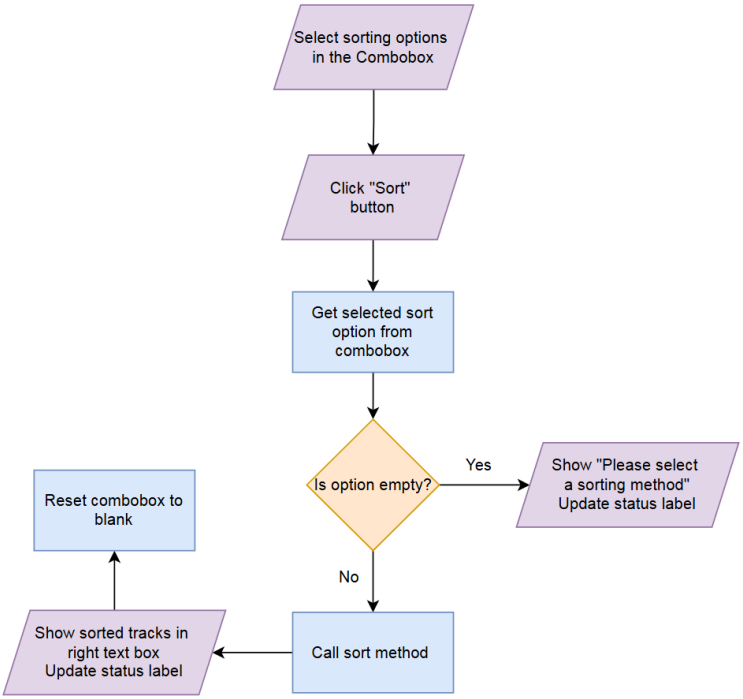
This Flowchart shows how the program starts, ends, and how List All Tracks function runs



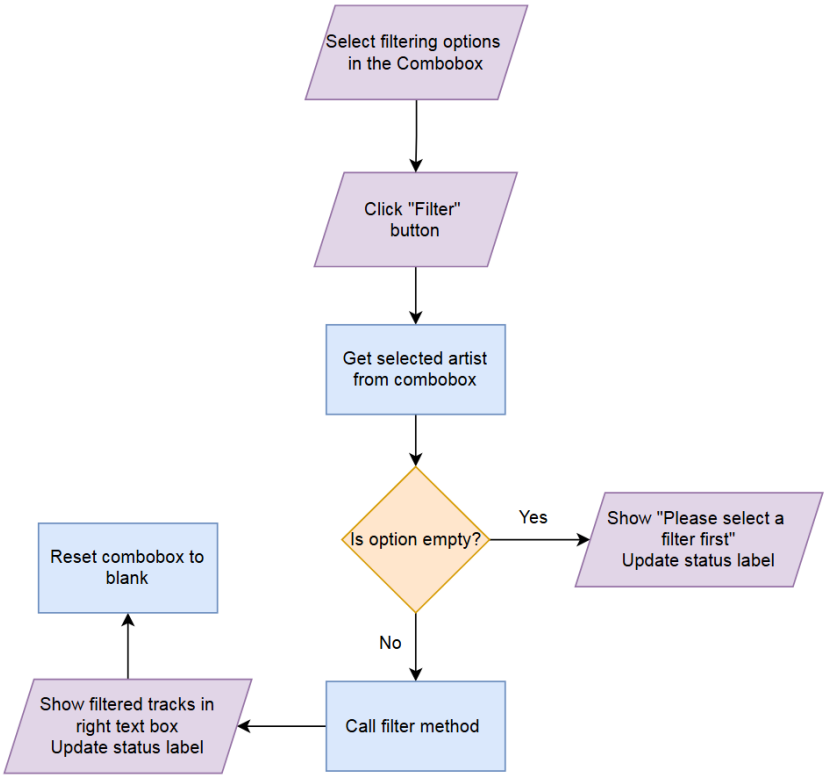
This Flowchart shows how the View Track function works



This Flowchart shows how the Search function works

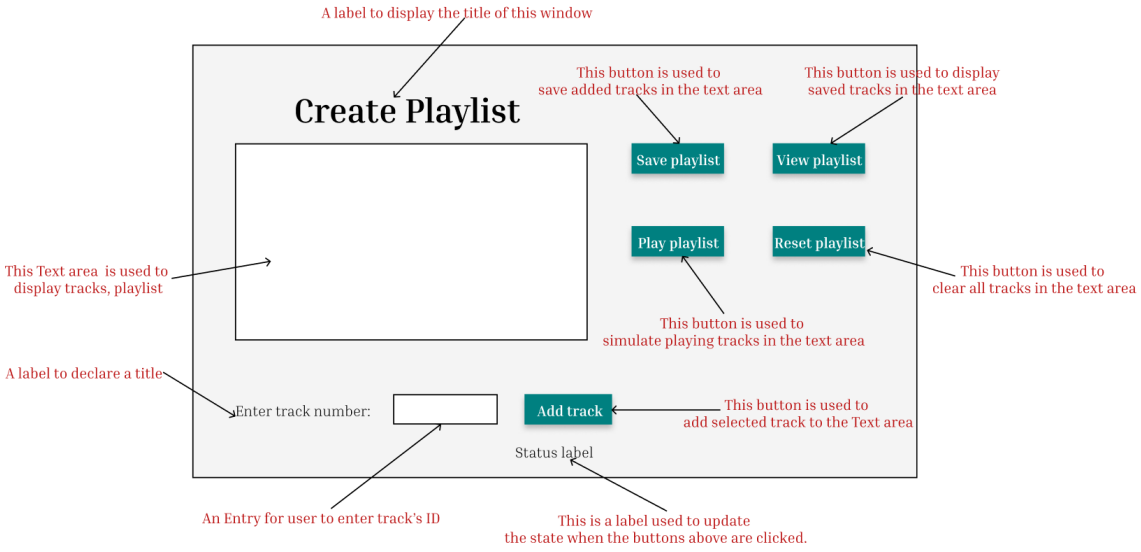


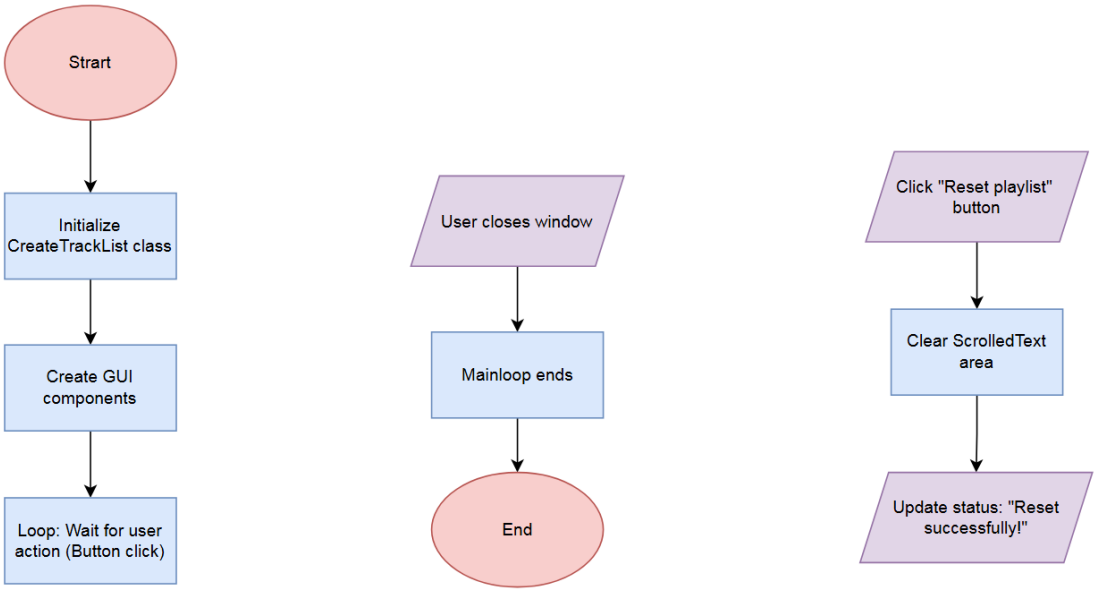
This Flowchart shows how the Sort function works



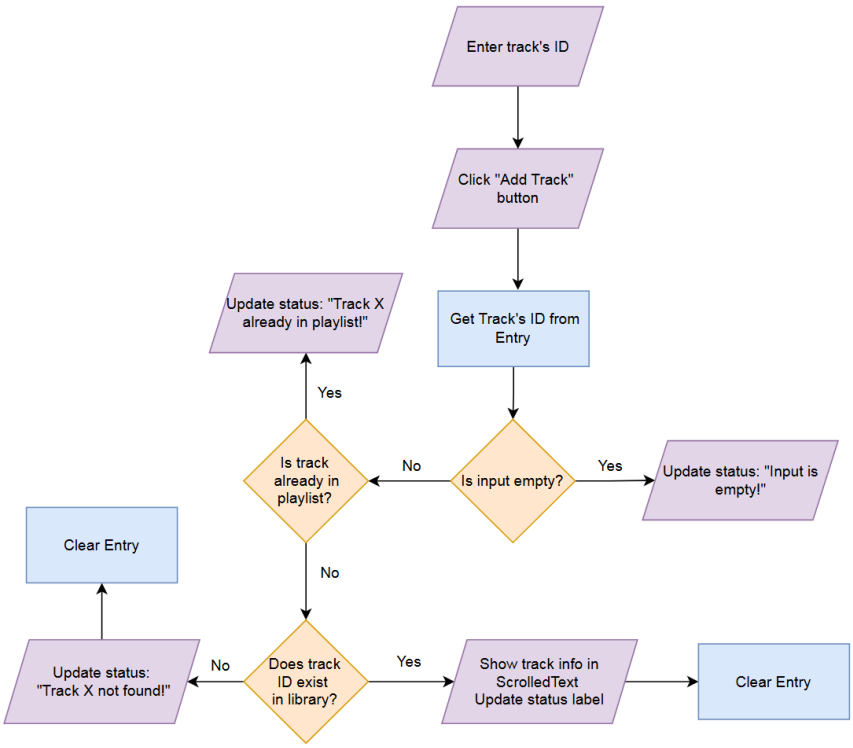
This Flowchart shows how the Filter function works

- **Create Playlist Window**

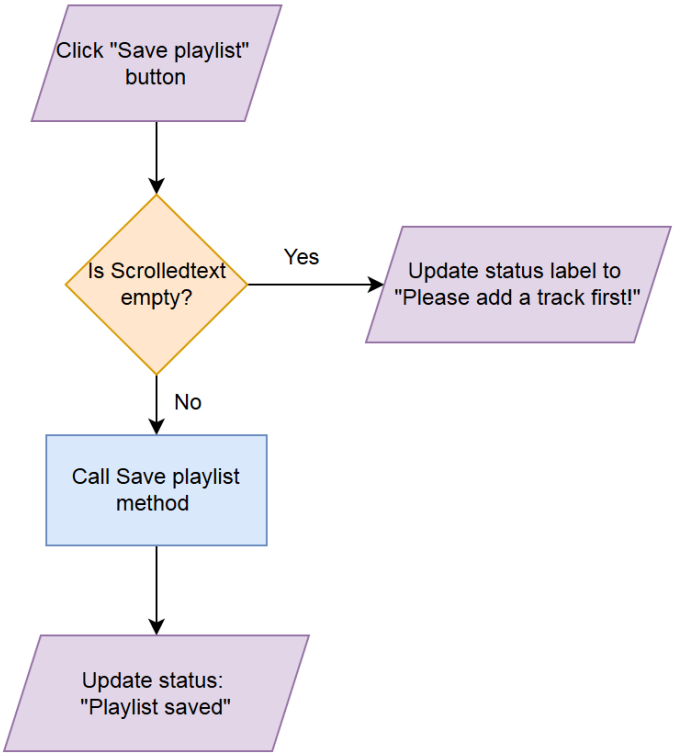




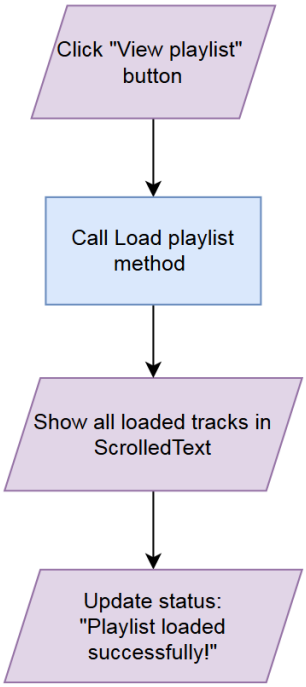
This Flowchart shows how the program starts, ends, and how Reset playlist function runs



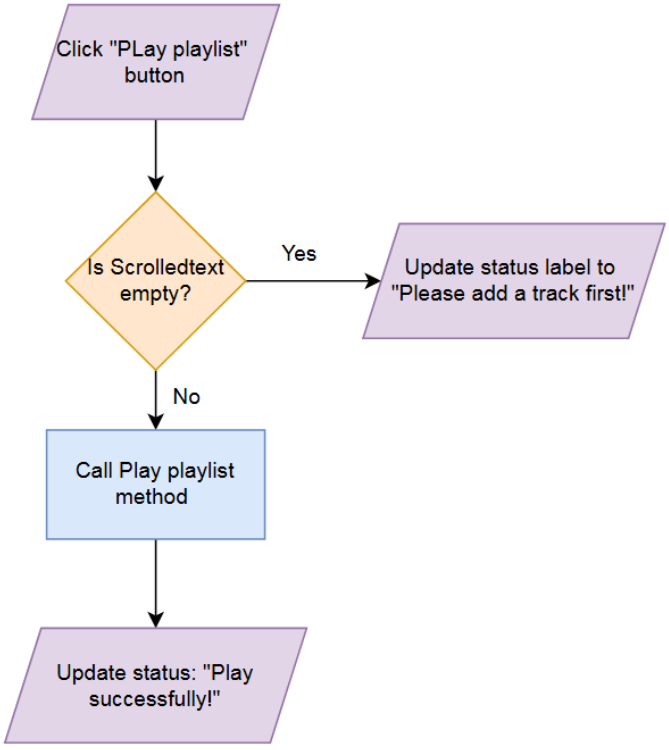
This Flowchart shows how the Add Track function works



This Flowchart shows how the Save playlist function works



This Flowchart shows how the View playlist function works



This Flowchart shows how the Play playlist function works

III. Development

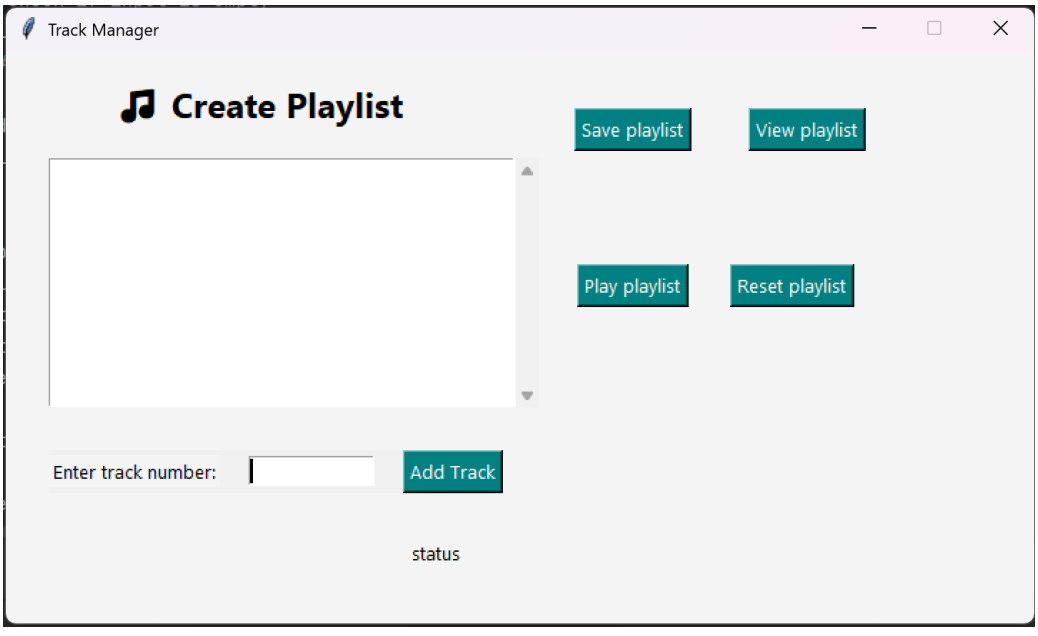
Based on the code provided below, the development part according to the assignment requirements includes 2 main parts: updating tracks and creating a track list with additional features such as searching, sorting, filtering, and using a JSON file.

First is the create_track_list.py file, which provides the GUI and functions that allow the user to create, store, and play music.

The interface for the Create Track List section includes a title label, a ScrolledText box to display the added playlist, buttons for functions such as adding tracks, saving the playlist to JSON, resetting the playlist, viewing the playlist from JSON, and simulating play. Finally, a status label to provide feedback to the user. All widgets use the pack() and grid() methods to arrange the widgets.

```
def __init__(self, window):
    self.window = window                        #Assign a window object
    self.window.geometry("720x400")             # Set window size
    self.window.configure(bg="#f4f4f4")         #Set background color
    self.window.title("Track Manager")          #Set window title
    self.window.resizable(False, False)         #Disable resizing
    self.id_lib = []                            #Store track IDs added to
playlist
    self.file_path = "playlist.json"            #File to save/load playlist from

    self.create_widgets()   #call function to create GUI

def create_widgets(self):
    title_lbl = tk.Label(self.window, text="🎵 Create Playlist",
font=get_h1_font(), bg="#f4f4f4")       #Create and locate main label
    title_lbl.grid(row=0, column=0, pady=(10, 5), padx=(40,0))

    self.list_txt = tkst.ScrolledText(self.window, width=46, height=10,
wrap="none", font=("Segoe UI", 10))     #Create and locate text area to
display playlist tracks
    self.list_txt.grid(row=1, column=0, columnspan=3, sticky="w", padx=30,
pady=(5, 10))


    add_track_frame = ttk.Frame(master=self.window)     #Create and locate a
frame for track input and Add button
    add_track_frame.grid(row=2, column=0, columnspan=2, sticky="w", pady=20,
padx=30)

    input_lbl = tk.Label(add_track_frame, text="Enter track number:",
font=("Segoe UI", 10), bg="#f4f4f4")      #Create and locate label and entry
for track number input
    input_lbl.pack(side="left", padx=(0,20))

    self.track_txt = tk.Entry(add_track_frame, width=12, font=("Segoe UI",
10))     #Create Entry to get user input
    self.track_txt.pack(side="left", padx=(0,20))
    self.track_txt.focus()      #Automatically focus when window opens

    self.add_track_btn = tk.Button(add_track_frame, text="Add Track",
```

```
command=self.add_track_clicked, font=("Segoe UI", 10), bg="#008080",
fg="white")        #Create and locate a button to add track to playlist (binds
to add_track_clicked)
    self.add_track_btn.pack(side="left")

    self.save_btn = tk.Button(self.window, text="Save playlist",
command=self.save_clicked, font=("Segoe UI", 10), bg="#008080", fg="white")
#Create and locate a button to save current playlist to JSON file (binds to
save_clicked)
    self.save_btn.grid(row=0, column=2, padx=(20,0), pady=(40,0))

    self.view_playlist_btn = tk.Button(self.window, text="View playlist",
command=self.load_clicked, font=("Segoe UI", 10), bg="#008080", fg="white")
#Create and locate a button to load playlist from file (binds to
load_clicked)
    self.view_playlist_btn.grid(row=0, column=3, padx=(40,0), pady=(40,0))

    self.play_btn = tk.Button(self.window, text="Play playlist",
command=self.play_click, font=("Segoe UI", 10), bg="#008080", fg="white")
#Create and locate a button to simulate "playing" the playlist (binds to
play_click)
    self.play_btn.grid(row=1, column=2, padx=(20,0))

    self.reset_btn = tk.Button(self.window, text="Reset playlist",
command=self.reset_clicked, font=("Segoe UI", 10), bg="#008080", fg="white")
#Create and locate a button to reset the playlist (binds to reset_clicked)
    self.reset_btn.grid(row=1, column=3, padx=(20,0))

    self.status_lbl = tk.Label(self.window, text="status", font=("Segoe UI",
10), bg="#f4f4f4")  #Create and locate a label to follow status
    self.status_lbl.grid(row=3, column=0, columnspan=4, sticky="we", padx=10,
pady=10)
```



- Function to add tracks to playlist

```
def add_track_clicked(self):    #A function to add a track
    key = self.track_txt.get()      #Get track number from input field
    if not key:     #Check if input is empty
        self.status_lbl.configure(text="Input is empty!", fg="red")
#inform the status
        return      #Exit the loop if needed

    if key in self.id_lib:      #Check if input is already in id_lid
        self.status_lbl.configure(text=f"Track {key} already in playlist!",
fg="orange")
        return
    else:
        if key in library:       #Check if the entered track ID exists in the
music library
            self.id_lib.append(key)  #Add track ID to id_lib
            track_details = f"{get_name(key)} by {get_artist(key)} - rating:
{get_rating(key)}"     #Get track's detail
            self.status_lbl.configure(text=f"Track {key} added", fg="green")
#inform status
            insert_text(self.list_txt, track_details)  #Display the content
in list_txt
        else:
            self.status_lbl.configure(text=f"Track {key} not found!",
fg="red")     #Inform if track not found

    self.track_txt.delete(0, tk.END)    # Clear input field after attempt
```

This function allows the user to add a song to a playlist by entering the song ID. When the "Add Track" button is clicked and the add_track_clicked function is called at the same time, it will first validate if the input is empty. If there is data from the user input, it will further check if the input exists in the library. If yes, it will return the song details in scrolledText by using insert_text() method and update the status; otherwise, it will notify that the ID was not found in red. Finally, it will clear the input in the entry.

- Function to save playlist to JSON file

```
def save_clicked(self):     #Function to save the current playlist to a JSON
file.
    if self.id_lib:     #Check if id_lib is not empty
        with open(self.file_path, "w") as file:     #Open with file_path and
write to JSON file
            json.dump(self.id_lib, file, indent=4)      #Save track IDs as
JSON list
            self.status_lbl.configure(text="Playlist saved", fg="green")
#Inform the status if saved
    else:
        self.status_lbl.configure(text="Please add a track fist!", fg="red")
#Inform the status if list_txt is empty
```

This function allows the user to save a playlist in a JSON file (playlist.json) after clicking the "Save Playlist" button. First, the function will check if there is any key in id_lib. If there is, the program will write the track ID to the array id_lib in the JSON file using the dump() method and update the new status in green. If not, it will report an error message in red.

- Function to import data from JSON file

```
def load_clicked(self):     #Function to display playlist from JSON file to
the list_txt
    self.id_lib.clear()     #Clear current playlist
    with open(self.file_path, 'r') as file:
        data = json.load(file)      #Read track ID list from file
        load_track = []         #Create a list to store track in JSON file
        for trackId in data:
            load_track.append(f"{get_name(trackId)} by {get_artist(trackId)}
- rating: {get_rating(trackId)}")      #Add track information to load_track
            self.id_lib.append(trackId)     #Add track ID to id_lib
        set_text(self.list_txt, '\n'.join(load_track) + '\n')       #Display
the content in list_txt
        self.status_lbl.configure(text="Playlist loaded successfully!",
fg="green")     #Inform the status
```

When the user clicks the "Load Playlist" button, the load_clicked function will clear the old data in id_llib to avoid duplication when the user spams the button. Next, use the load() method to get the data saved in the JSON file and loop through each track ID to get all the information and print it into ScrollText using the set_text method and update the new status in green.

- Function to simulate playing playlist

```
def play_click(self):
    if self.id_lib:     #Check if it is empty
        for key in self.id_lib:
            increment_play_count(key)       #Call function to increase play
count
        self.status_lbl.configure(text="Play successfully!")    #Inform
status

    else:
        self.status_lbl.configure(text="Please add a track first!", fg="red")
#Inform status
```

The "Play Playlist" button simulates playing the songs contained in the ScrollText. First, it checks if there is a song; if there is, it loops through all songs and increases the play count by 1. Then it updates the new status in green. Otherwise, it updates a red error message.

- Function to reset playlist

```
def reset_clicked(self):    #Function to clear all data in list_txt
    self.list_txt.delete(1.0, tk.END)       #Delete data
    self.id_lib.clear()  # Delete track IDs in id_lib
    self.status_lbl.configure(text="Reset successfully!", fg="green")
#Inform status
```

When the user clicks "Reset Playlist" the function will completely delete the playlist in ScrollText as well as id_lib. Then update the new status in red.

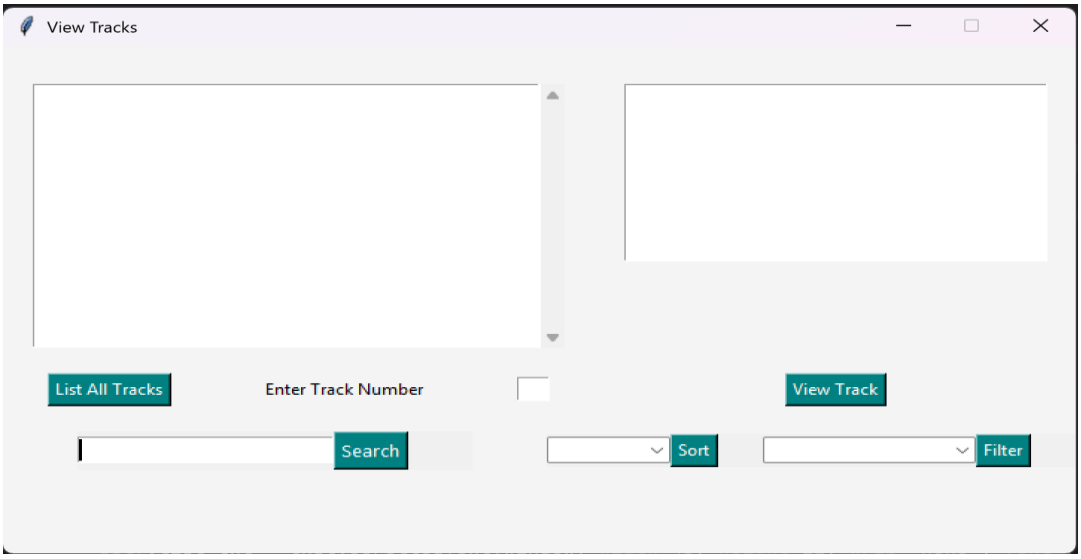
- View all source code here

The second is the view_tracks file, in addition to the GUI and available functions like list all tracks and view track, I have added searching, sorting, and filtering functions.

The interface for the Track View section consists of 2 ScrolledText boxes to display details of the current playlist and all songs, buttons for functions such as listing all tracks, viewing tracks, searching, sorting, and filtering. Finally, status labels provide feedback to the user. All widgets use the pack() and grid() methods to arrange the widgets.

```
def __init__(self, window):      #Constructor for the TrackViewer GUI class
    self.window = window         #Store reference to the parent window
    self.window.geometry("720x400")          #Set size for the window
    self.window.configure(bg="#f4f4f4")      #Set background color
    self.window.title("View Tracks")         #Set window title
    self.window.resizable(False, False)      #Disable resizing


    self.list_txt = tkst.ScrolledText(window, width=48, height=12,
wrap="none", font=("Segoe UI", 10))       #Create and locate text box with
scrollbar to show all tracks
    self.list_txt.grid(row=0, column=0, columnspan=3, sticky="W", padx=20,
pady=(30, 10))

    self.track_txt = tk.Text(window, width=40, height=8, wrap="none",
font=("Segoe UI", 10))         #Another text box to show details of selected
or searched track
    self.track_txt.grid(row=0, column=3, sticky="NW", padx=20, pady=(30, 10))


    self.list_tracks_btn = tk.Button(window, text="List All Tracks",
command=self.list_tracks_clicked, bg="#008080", fg="white")         #Button to
list all available tracks
    self.list_tracks_btn.grid(row=1, column=0, padx=10, pady=10)

    enter_lbl = tk.Label(window, text="Enter Track Number", bg="#f4f4f4")
#Label for the track number entry
    enter_lbl.grid(row=1, column=1, padx=10, pady=10)

    self.input_txt = tk.Entry(window, width=3)       #Create and locate entry
field to input track number
    self.input_txt.grid(row=1, column=2, padx=10, pady=10)

    self.check_track_btn = tk.Button(window, text="View Track",
command=self.view_tracks_clicked, bg="#008080", fg="white")      #Button to
show details of the track with entered ID
    self.check_track_btn.grid(row=1, column=3, padx=10, pady=10)


    search_frame = ttk.Frame(master=self.window)         #Create a sub-frame
to hold the search entry and button
    search_frame.grid(row=2, column=0, columnspan=2, sticky="we",
```

```
padx=(50,0), pady=10)

    self.search_txt = tk.Entry(search_frame, width=24, font=("Segoe UI", 10))
#Create and locate entry field to get user input
    self.search_txt.pack(side="left")
    self.search_txt.focus()      #Set focus to the search input

    self.search_btn = tk.Button(search_frame, text="Search",
command=self.search_track ,font=("Segoe UI", 10), bg="#008080", fg="white")
#Button to trigger search
    self.search_btn.pack(side="left")

    groups_sorting = ["", "rating ↑", "rating ↓", "a-z", "z-a"]     #List of
sort options

    sort_frame = ttk.Frame(window)      #Create and locate frame for both
sorting and filtering controls
    sort_frame.grid(row=2, column=2, columnspan=2, sticky="we", padx=(50,0),
pady=10)

    self.combo_sort = ttk.Combobox(sort_frame, width=10)        #Create a
combobox
    self.combo_sort["values"] = groups_sorting                  #Set the
option in groups_sorting
    self.combo_sort.current(0)              #Set default selection (empty)
    self.combo_sort.pack(side="left")       #Place the sorting combobox

    self.check_combobox = tk.Button(sort_frame, text="Sort",
command=self.sort_combobox_clicked, bg="#008080", fg="white")      #Button to
apply selected sort option
    self.check_combobox.pack(side="left", padx=(0,30))

    groups_filter = [""]        #Initialize artist filter list with an empty
option
    for key in lib.library:
        if get_artist(key) not in groups_filter:    #Avoid duplicates
            groups_filter.append(get_artist(key))   #Add artist

    self.combo_filter = ttk.Combobox(sort_frame)        #Create a combobox
    self.combo_filter["values"] = groups_filter         #Set the option in
groups_filter
    self.combo_filter.current(0)                        #Set default
selection (empty)
    self.combo_filter.pack(side="left")                 #Place the sorting
combobox

    self.check_combo_filter = tk.Button(sort_frame, text="Filter",
command=self.filtered_combobox_clicked, bg="#008080", fg="white")
#Create and locate a button binds to filtered_combobox_clicked
    self.check_combo_filter.pack(side="left")


    self.status_lbl = tk.Label(window, text="", font=("Segoe UI", 10),
bg="#f4f4f4")        #Create a label to follow status
    self.status_lbl.grid(row=3, column=0, columnspan=4, sticky="we", padx=10,
pady=10)      #Locate the label
```



Next is the main functional analysis section.

- Function to search track in library

```
def search_track(self):     #Called when "Search" button is clicked
    search_input = self.search_txt.get().strip().lower()        #Get input
and normalize it

    if search_input == "":      #If input is empty
        set_text(self.track_txt, "No matching track found")     #Display
content in right text box
        self.status_lbl.configure(text="Input is empty!", fg="red")
#Update status
    else:
        filtered_tracks = [track for track in lib.library.values()
                           if search_input in track.name.lower() or
search_input in track.artist.lower()]       #Filter tracks that match input
(in name or artist)

        if filtered_tracks:     # Found matches
            lines = [f"{track.name} by {track.artist} (Rating:
{track.rating})" for track in filtered_tracks]
            set_text(self.track_txt, '\n'.join(lines))      #Show results
            self.status_lbl.configure(text=f"Found {len(filtered_tracks)}
track(s)", fg="green")
            self.search_txt.delete(0, tk.END)       #Clear search entry
        else:
            set_text(self.track_txt, "No matching track found")
            self.status_lbl.configure(text="No matching track found",
fg="red")
```

This function allows users to search by keywords, including song title and artist name. When the user enters an input, the program will first validate the input, then use a loop to compare

the input with all song titles and artist names. Then display all matching songs in the ScrollText box using the set_text() function. After completion, it will clear the input that the user entered.

- Function to sort the playlist

```
def ascending_sort(self):
    sorted_tracks = sorted(lib.library.values(), key=lambda x: x.rating)
#Use lambda to extract 'rating' attribute as sorting key
    self.display_sorted(sorted_tracks)      #Display the sorted list
def descending_sort(self):
    sorted_tracks = sorted(lib.library.values(), key=lambda x: x.rating,
reverse=True)       #'reverse=True' reverses the order of the sorted list
    self.display_sorted(sorted_tracks)
def alphabet_sort(self):
    sorted_tracks = sorted(lib.library.values(), key=lambda x:
x.name.lower())      #Convert to lowercase to ensure case-insensitive sorting
    self.display_sorted(sorted_tracks)
def alphabet_reserve(self):
    sorted_tracks = sorted(lib.library.values(), key=lambda x:
x.name.lower(), reverse=True)
    self.display_sorted(sorted_tracks)
```

These functions allow the user to sort by star or alphabetical order. Use an anonymous lambda function to get the values to be sorted, and use the sorted() method to sort the values contained in the list.

- Function to filter the playlist

```
def artist_filter(self, filter_get_txt):    #Called to filter tracks by
artist name
    filtered_list = [track for track in lib.library.values() if
filter_get_txt == track.artist]     #Get track information
    self.display_sorted(filtered_list)      #Show matching tracks
    self.status_lbl.configure(text=f"Found {len(filtered_list)} track(s) by
{filter_get_txt}", fg="green")
```

This feature allows the user to view all tracks of a specific artist. When the user selects an artist contained in the combobox, the artist_filter() function will be called to find the songs in the library of that artist. Then display it into the ScrollText Box using the display_sort() function. Finally, update the new status to green.

- Function to display filtering and sorting the playlist

```
def display_sorted(self, sorted_tracks):    #Display tracks in right-side
text box
    lines = []      #Initialize an empty list to store formatted track
strings
    for track in sorted_tracks:
        lines.append(f"{track.name} by {track.artist} (Rating:
{track.rating})")    #Format each track as a string and add it to the list
    set_text(self.track_txt, '\n'.join(lines))      # Show all results
```

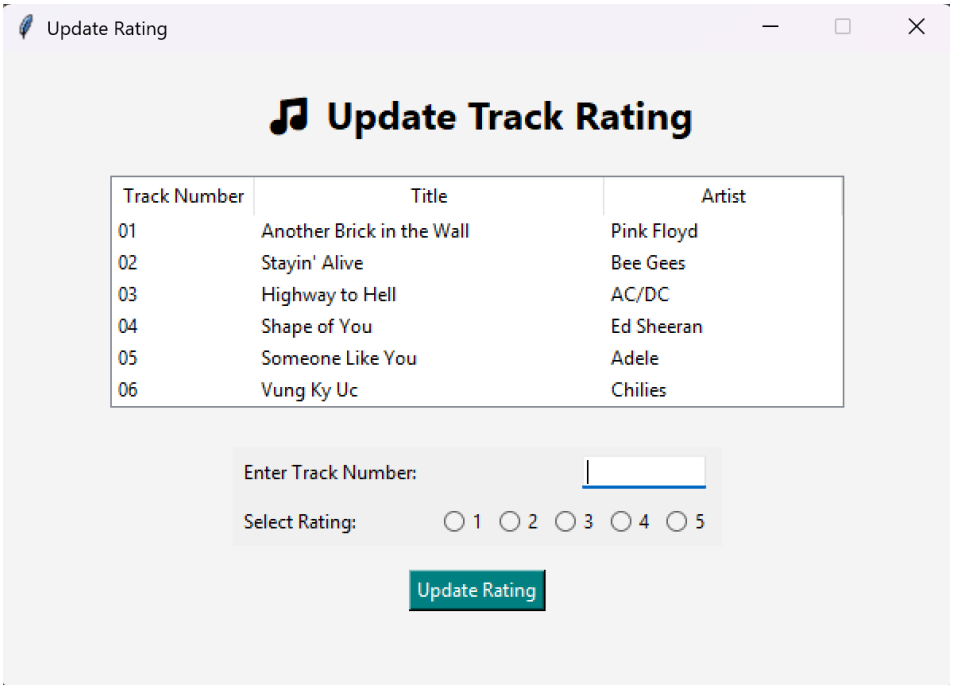
This function creates a list to contain all the values then displays in ScrollText Box using set_text() function.

- View all source code here

Finally, is the update_tracks file, in this part, I design a GUI using a treeview to create a table containing song information, along with widgets like Label, Entry, RadioButton, and Button. All widgets use the pack() and grid() methods to arrange the widgets.

```
def __init__(self, window):
    self.window = window    #Store the main window reference as an instance
variable for later use
    self.window.title("Update Rating")  #Set name title
    self.window.geometry("600x400")     #Set size for window
    self.window.resizable(False, False)     #Don't allow to resize
    self.window.configure(bg="#f4f4f4")     #Set background color

    self.library = library  #get all data from library

    self.create_widgets()   #call function to create GUI


def create_widgets(self):
    title_lbl = tk.Label(self.window, text="🎵 Update Track Rating",
font=get_h1_font(), bg="#f4f4f4")
    title_lbl.pack(pady=(20, 5))

    self.tree = ttk.Treeview(self.window, columns=("order", "title",
"artist"), show="headings", height=6, selectmode="none")  # Create a Treeview
widget to display a list of tracks with 3 columns: Track Number, Title, and
Artist
    self.tree.heading("title", text="Title")            #Define column header
for track title
    self.tree.heading("artist", text="Artist")          #Define column header
for artist name
    self.tree.heading("order", text="Track Number")     #Define column header
for track number (key)
    #Set the width for each column
    self.tree.column("title", width=220)
    self.tree.column("artist", width=150)
    self.tree.column("order", width=90)

    self.tree.pack(pady=15)     #Add vertical padding around the widget

    self.populate_tree()    #Call method to insert data into the table

    input_frame = ttk.Frame(master= self.window)    #create a frame to put
all the input buttons in
    input_frame.pack(pady=10)   #Add vertical padding around the input_frame

    ttk.Label(input_frame, text="Enter Track Number:").grid(row=0, column=0,
padx=5, pady=5, sticky="we")   #Create and locate a label
    self.track_num_var = tk.StringVar()     #create a string variable to
store user input
    self.track_num_input = ttk.Entry(input_frame,
textvariable=self.track_num_var, width=12)    #Create an input field for
track number (binds to self.track_num_var)
```

```
    self.track_num_input.grid(row=0, column=1, padx=10, pady=5, sticky="e")
#locate the track_num_input
    self.track_num_input.focus()     # Place the cursor in the input box

    ttk.Label(input_frame, text="Select Rating:").grid(row=1, column=0,
padx=5, pady=5, sticky="we")     #Create and locate a label in input_frame

    rating_frame = ttk.Frame(input_frame)    #create a frame to put all the
radio buttons in
    rating_frame.grid(row=1, column=1, padx=5, pady=5, sticky="e")
#locate the rating_frame

    self.rating_var = tk.IntVar()    #create an integer variable to store user
selection
    for i in range(1, 6):
        ttk.Radiobutton(rating_frame, text=str(i), variable=self.rating_var,
value=i).pack(side="left", padx=3)      #use a loop to create 5 radio buttons
in a row

    self.update_btn = tk.Button(self.window, text="Update Rating",
bg="#008080", fg="white", command=self.update_rating)     #create Update
Rating button
    self.update_btn.pack(pady=5)

    self.status_lbl = tk.Label(self.window, text="", fg="green",
bg="#f4f4f4")       #create label to inform status
    self.status_lbl.pack(pady=5)

def populate_tree(self):     #Insert each track into the Treeview with its ID,
name, and artist
    for key in self.library:
        self.tree.insert('', 'end', values=(key, get_name(key),
get_artist(key)))
```



Next is the main functional analysis section.

- Function to update rating

```
def update_rating(self):    #update rating function
    selected = self.track_num_var.get()     #get user input value

    if not selected:    #check if it is empty
        self.status_lbl.config(text="Please enter a track first.", fg="red")
        return

    if selected not in self.library:    #check if it not exist in library
        self.status_lbl.config(text="Invalid track number.", fg="red")
        return

    new_rating = self.rating_var.get()      #get data
    if new_rating == 0:     #check if user don't select
        self.status_lbl.config(text="Please select a rating.", fg="red")
        return

    set_rating(selected, new_rating)    #set new rating value
    self.status_lbl.config(text=f"Updated rating for '{get_name(selected)}'
to {new_rating} ★! Play count: {get_play_count(selected)}", fg="green")
#print new rating status and play count

    self.track_num_var.set("")  # clear track number entry
    self.rating_var.set(0)      # clear rating selection
```

When clicking the "Update Rating" button, the update_rating() function will be called. First, this function will validate the input to see if the user has entered enough data and conditions. If it is enough, it will call the set_rating() function to update the new rating and update the new status in green. If not, it will display an error in red. Finally, clear all the values entered by the user.

- View all source code here

IV. Testing and Validation

No.	Description	Expected Output	Input	Output	Evidence	Test Result
1	Test main GUI	Display the main application window	Run the program	Main GUI window appears	Evidence 1	PASSED
2	Open Create Playlist window	Create Playlist window appears and update the status	Click “Create Track List” button	New window opens, and update status	Evidence 2	PASSED
3	Open Update window	Update window appears and	Click “Update	New window	Evidence 3	PASSED

		update the status	Tracks" button	opens, and update status		
4	Open View Tracks window	View window appears and update the status	Click "View Tracks" button	New window opens, and update status	Evidence 4	PASSED
5	Add valid track to playlist	Track added and displayed	Enter "01", click "Add Track"	Track added to playlist	Evidence 5	PASSED
6	Add duplicate track	Error message shown	Add "01" again	"Already in playlist" message shown	Evidence 6	PASSED
7	Add invalid track	Error message shown	Enter "12", click "Add Track"	"Track not found" message shown	Evidence 7	PASSED
8	Reset playlist	Playlist cleared	Click "Reset Playlist"	Playlist area becomes empty	Evidence 8	PASSED
9	Save playlist	File saved	Click "Save Playlist" to save track IDs [01, 02, 03]	playlist.json created	Evidence 9	PASSED
10	Load playlist	Playlist reloaded	Click "View Playlist"	Tracks appear in text area	Evidence 10	PASSED
11	Simulate playback	Play count increased	Click "Play Playlist"	Status: play successfully	Evidence 11	PASSED
12	Update rating success	Rating updated	Enter "03", select rating "5", click "Update Rating"	Status: updated, play count shown	Evidence 12	PASSED

13	Update without rating	Error message	Enter “03”, no rating selected, click “Update Rating”	“Please select a rating” shown	Evidence 13	PASSED
14	Update with invalid ID	Error message	Enter “12a”, click “Update Rating”	“Invalid track number” shown	Evidence 14	PASSED
15	Search track by keyword	Matching tracks shown, Status: track found	Enter “Ex’s”, click “Search”	Tracks with “Ex’s” in name/artist shown, status: found 02 tracks	Evidence 15	PASSED
16	Sort by name A–Z	Tracks sorted alphabetically	Select “A–Z”	List sorted A to Z	Evidence 16	PASSED
17	Sort by rating descending	Tracks sorted 5 → 1	Select “Rating ↓”	List sorted from highest to lowest rating	Evidence 17	PASSED
18	Filter by artist	Only that artist’s tracks shown	Select “Adele” from filter	Tracks by Adele shown	Evidence 18	PASSED
19	View track by ID	Track details shown	Enter “03”, click “View Track”	Shows name, artist, stars, play count	Evidence 19	PASSED
20	View track by invalid ID	Track not found	Enter “12”, click “View Track”	Track 12 not found	Evidence 20	PASSED
21	Search with no entry	Status: Input is empty, and No matching track found	Don’t enter any character	Status: input is empty, and Track not found	Evidence 21	PASSED

Unit testing by using PyTest:

- Test info(), starts() functions and default constructor

```
from library_item import LibraryItem

def test_info():
    item = LibraryItem("Ex's Hate Me", "B Ray", 4)
    expected = "Ex's Hate Me - B Ray ****"
    assert item.info() == expected

def test_starts():
    item = LibraryItem("Stayin' Alive", "Bee Gees", 5)
    assert item.stars() == "*****"

def test_default_constructor():
    item = LibraryItem("Stayin' Alive", "Bee Gees")
    assert item.play_count == 0
    assert item.rating == 0
```

Evidence 22

- Test get_name(), get_artist(), get_rating(), increment_play_count() functions

```
from track_library import get_play_count, get_artist, get_rating, get_name,
set_rating, increment_play_count

def test_get_name():
    assert get_name("01") == "Another Brick in the Wall"
def test_get_name_invalid():
    assert get_name("12") is None

def test_get_artist():
    assert get_artist("01") == "Pink Floyd"
def test_get_artist_invalid():
    assert get_artist("12") is None

def test_get_rating():
    assert get_rating("03") == 2
def test_get_rating_invalid():
    assert get_rating("12") == -1

def test_set_rating_valid():
    set_rating("04", 5)
    assert get_rating("04") == 5
def test_set_rating_invalid():
    assert set_rating("12", 4) is None

def test_increment_play_count():
    increment_play_count("01")
    assert get_play_count("01") == 1
```

```python
def test_get_play_count_invalid():
    assert get_play_count("12") == -1
```

Evidence 23

V. Conclusion and Further Development

This project helped me understand the principles of OOP in a practical way. I built more features based on the provided template to create a more user-friendly interface. The system allows users to view tracks, create playlists, simulate playback, and add functions like sorting, searching, and filtering. Additionally, I used a JSON file to save the playlist for future access.

If I had 3 more months to develop, I would focus on developing advanced features to increase realism such as playing real sounds using pygame or playsound libraries. Besides I will add music duration bar that allows to follow the music or fast forward the audio right on the duration bar. Next is allowing users to create multiple albums. Moreover, I also want to restructure the code into an MVC structure to make it easier to extend and test in the future.

Thanks to this course, I have a deeper understanding of how object-oriented programming supports modularity and reusability, maintainability, and upgradeability. Creating independent GUI components that share logic helped me understand how well-designed classes and functions reduce repetition and increase flexibility.

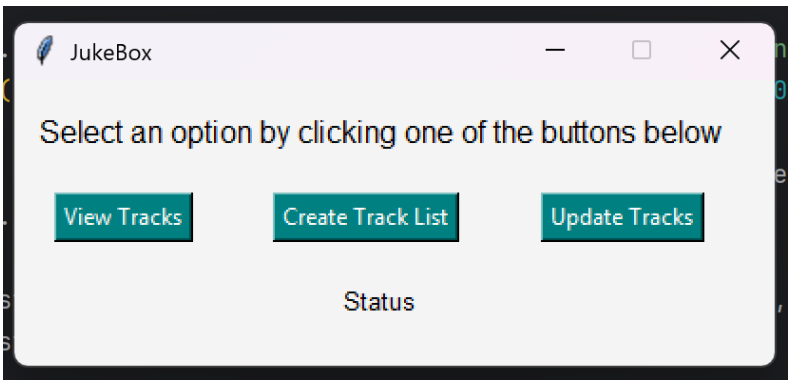
The most challenging part for me in this project was interacting with JSON files. I had to learn on my own to ensure data integrity. Besides, handling user input validation required ensuring data consistency. These tasks pushed me to improve my programming skills and self-study as well as testing skills.

On the other hand, designing the user interface was more straightforward. I had a lot of practice with GUIs and was familiar with layout management using grids and packs in Tkinter. Therefore, I pretty like to design widgets that have a user-friendly experience.

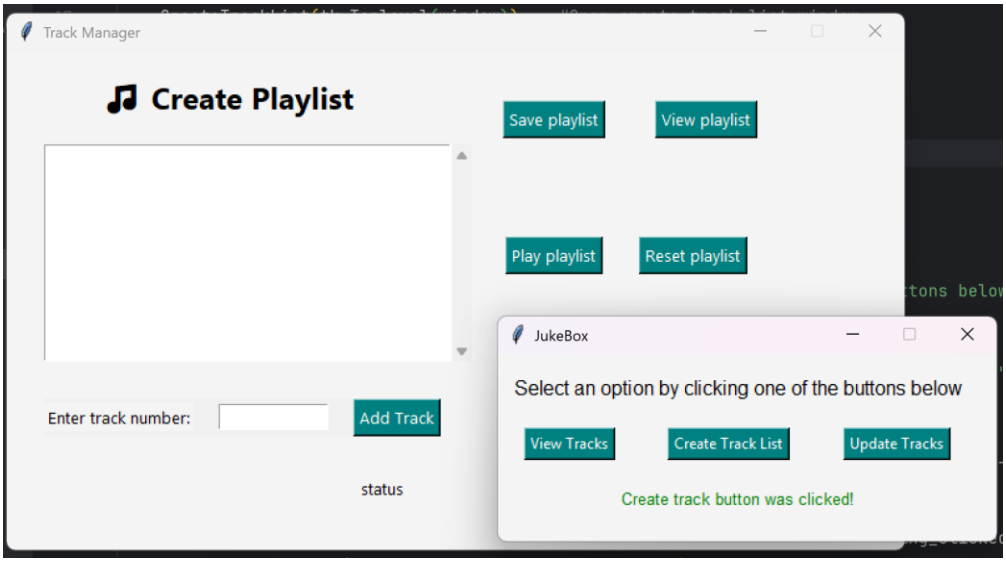
Overall, this project helped me improve my Python language skills significantly, especially in GUI programming, module design, and automated testing. It taught me how to structure and manage data optimally and reasonably. Most importantly, it allowed me to better understand the properties of OOP, which is an indispensable part of my future career.

VI. Appendix

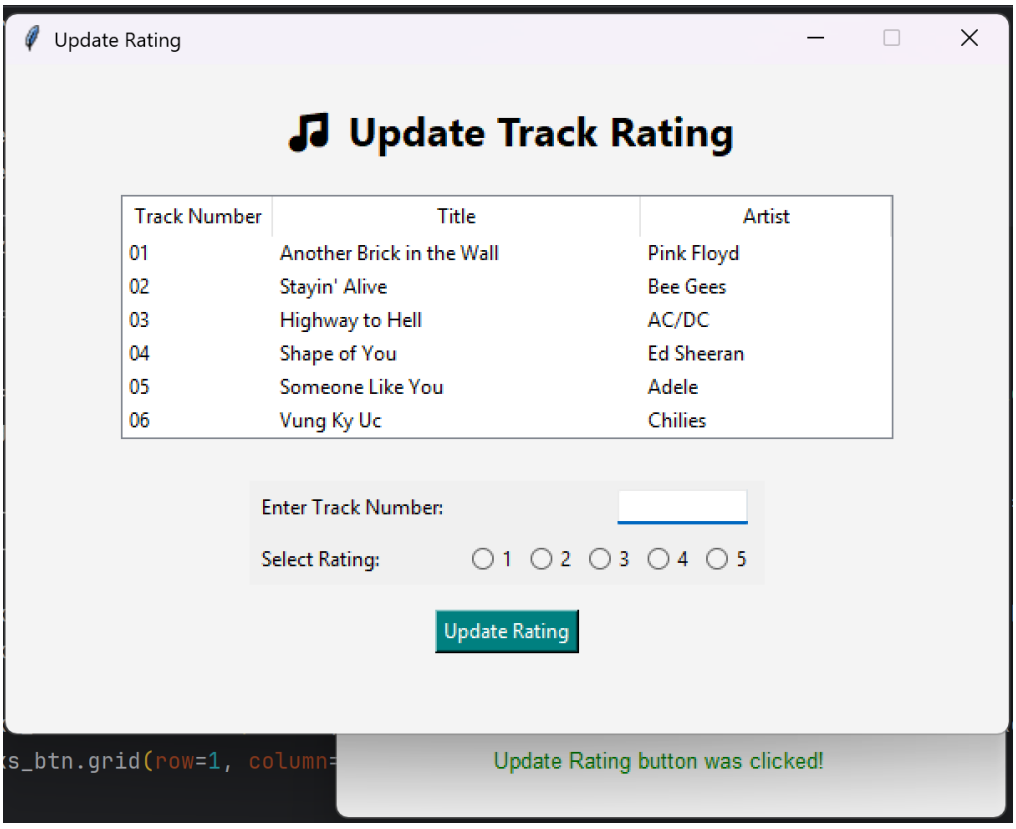
- Evidence 1: Test main GUI



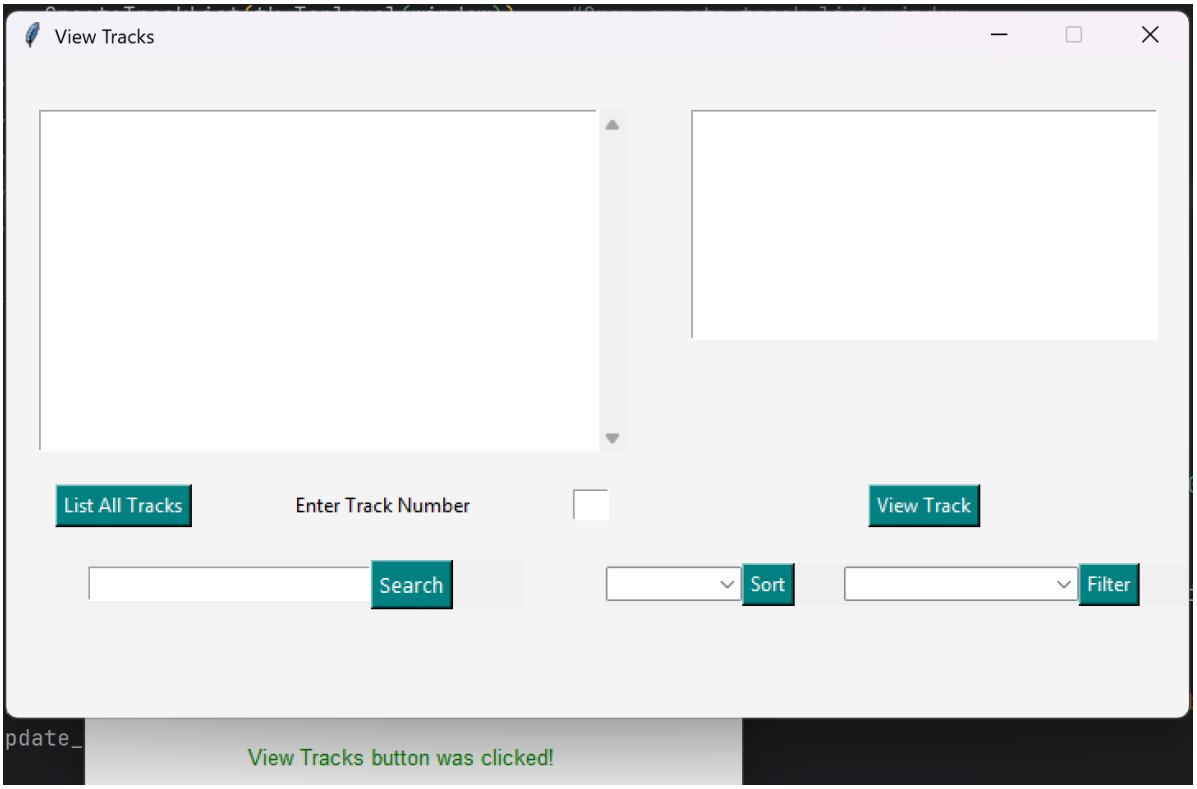
- Evidence 2: Open Create Playlist window



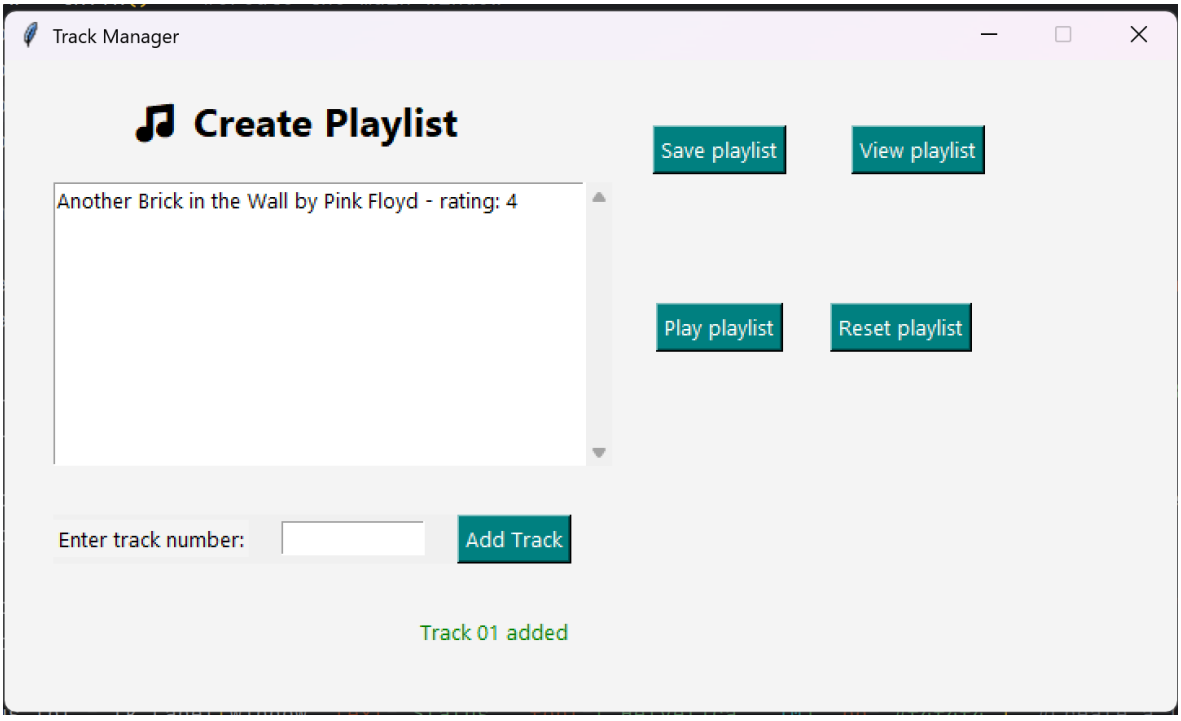
- Evidence 3: Open Update window



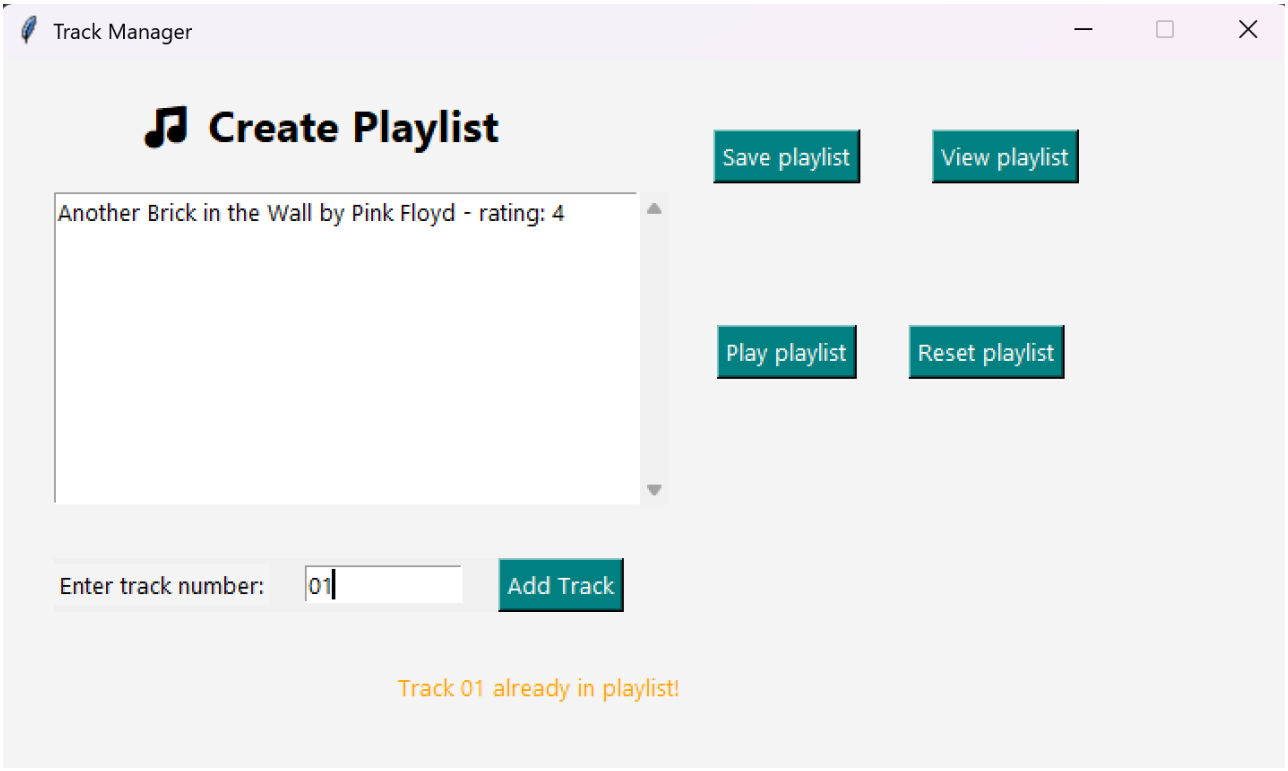
- Evidence 4: Open View Tracks window



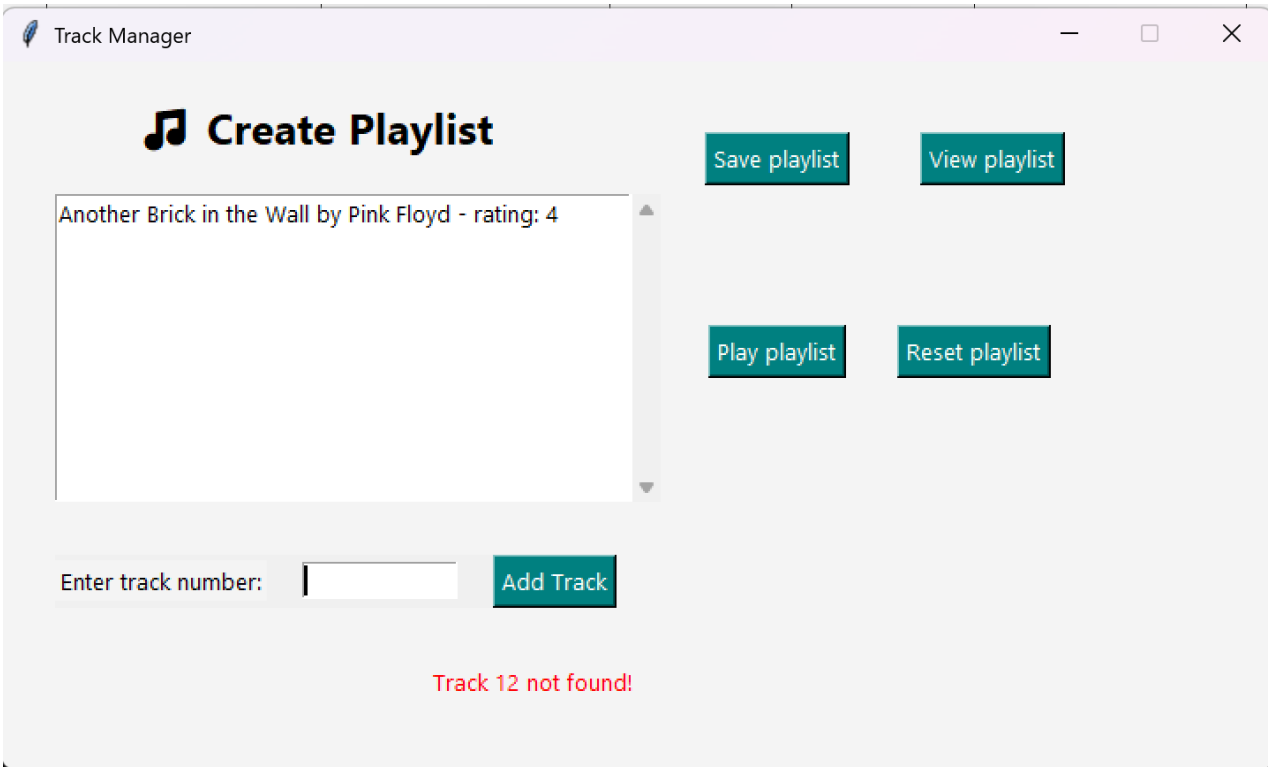
- Evidence 5: Add valid track to playlist



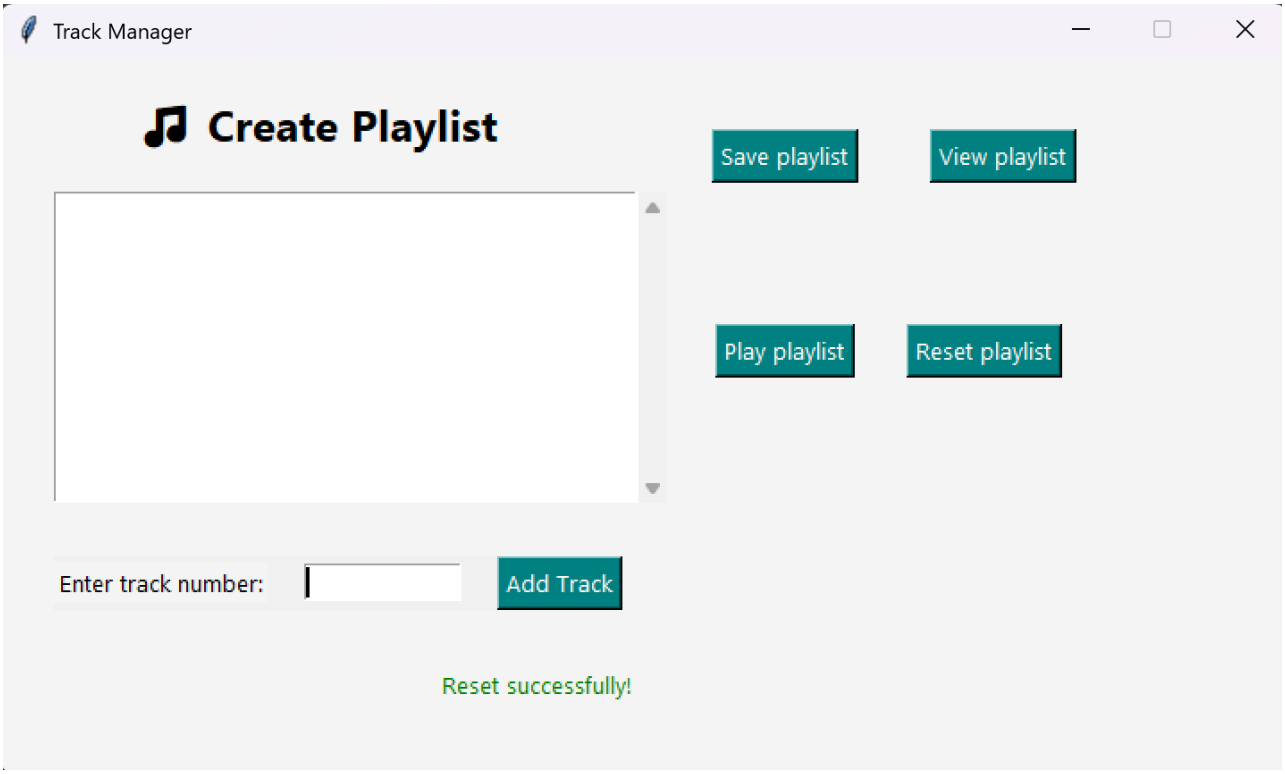
- Evidence 6: Add duplicate track



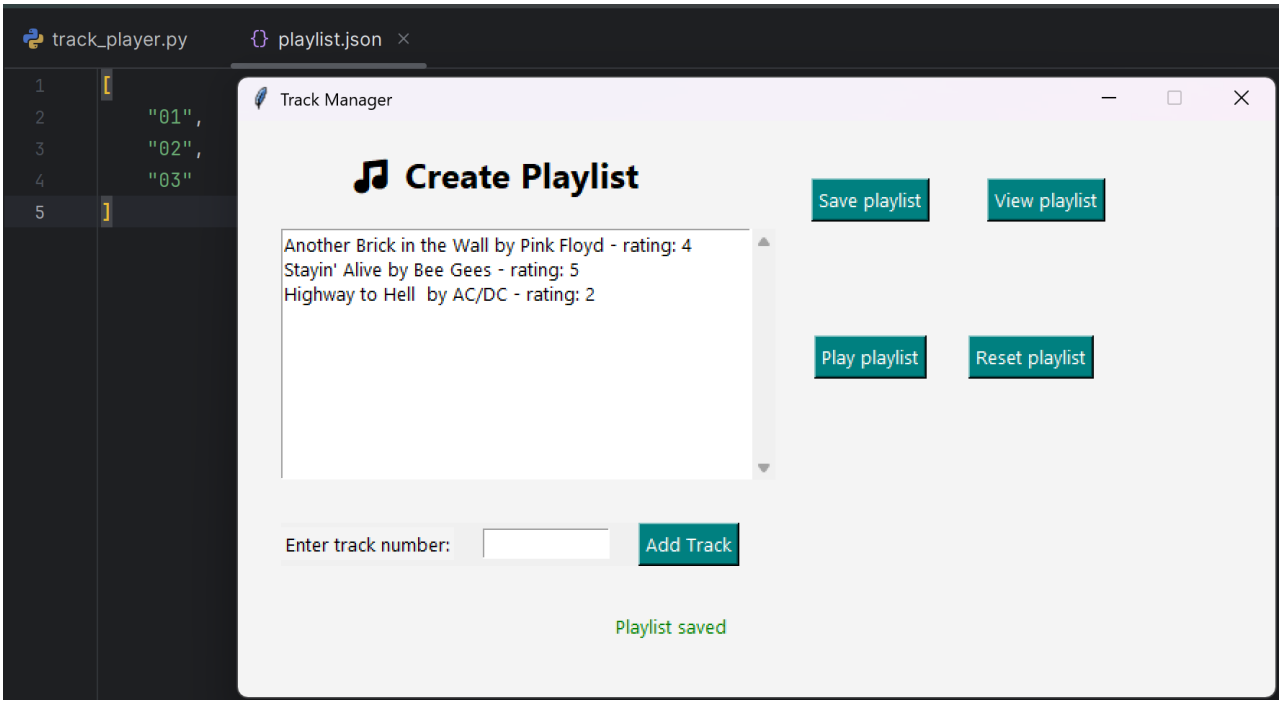
- Evidence 7: Add invalid track



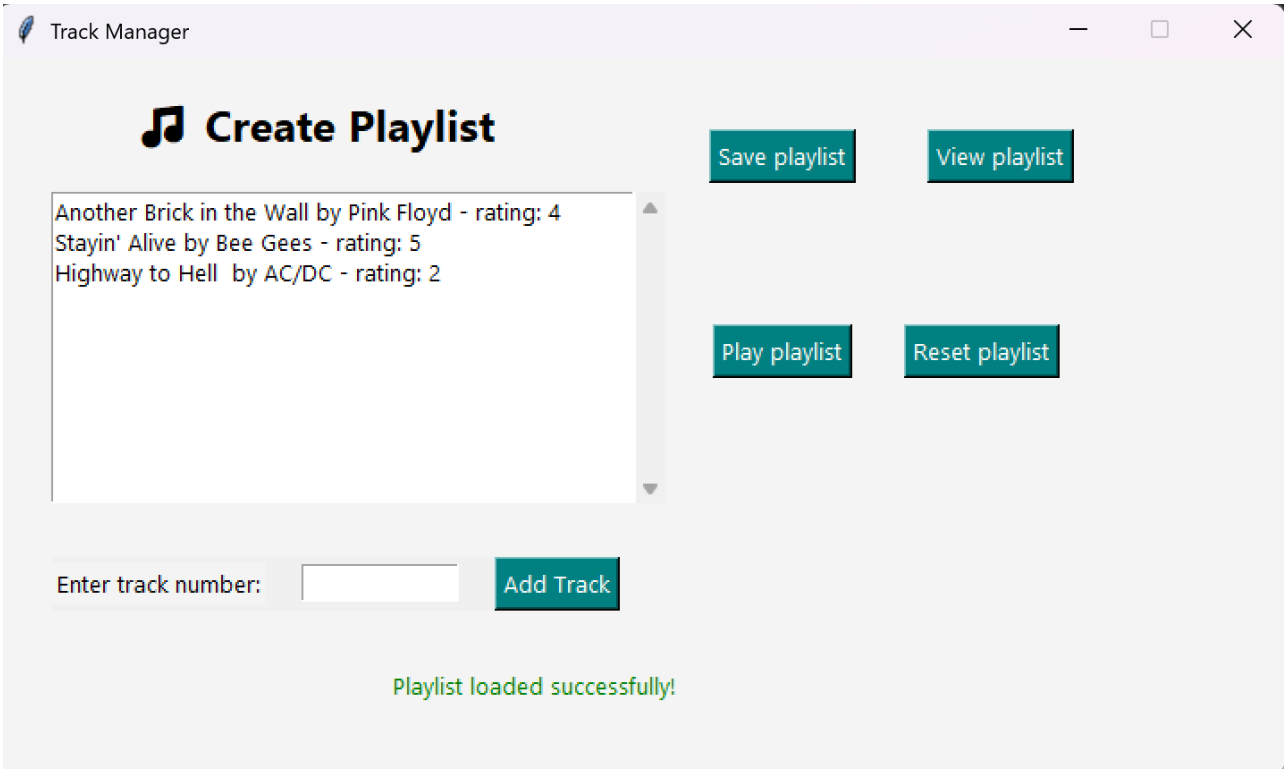
- Evidence 8: Reset playlist



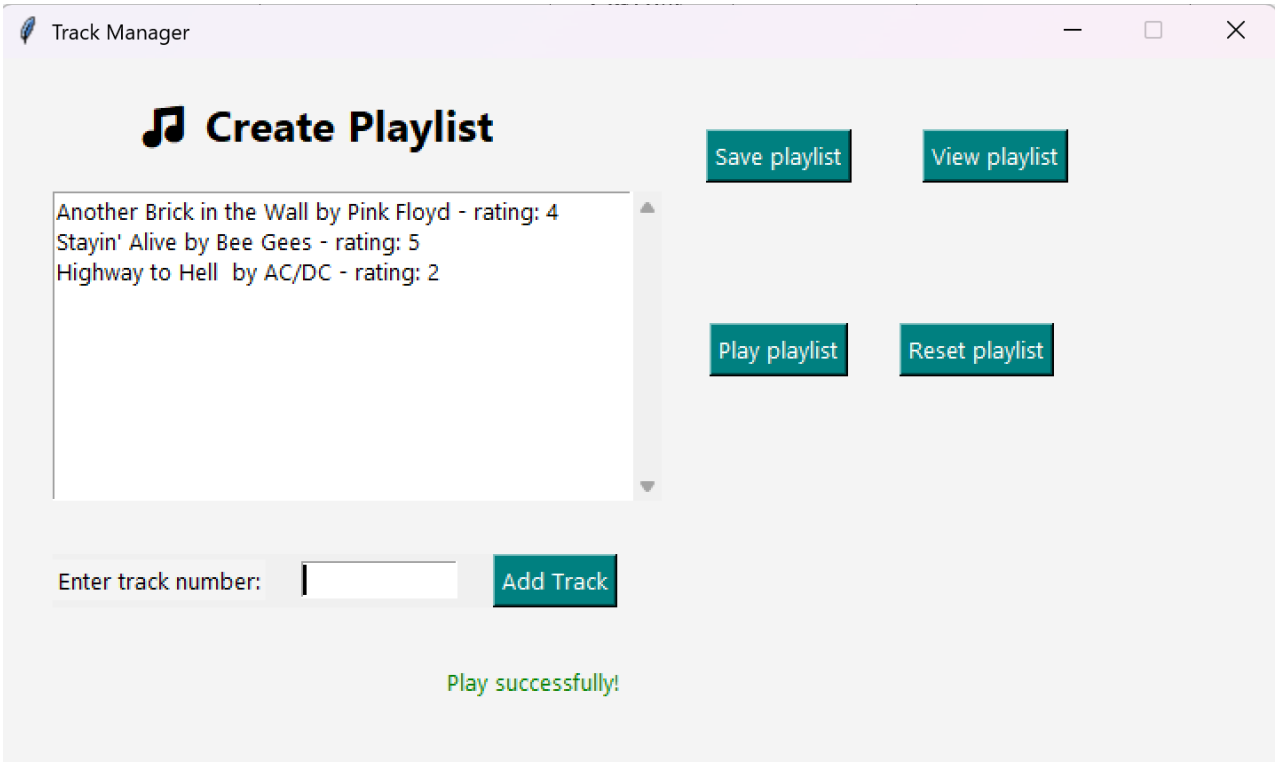
- Evidence 9: Save playlist



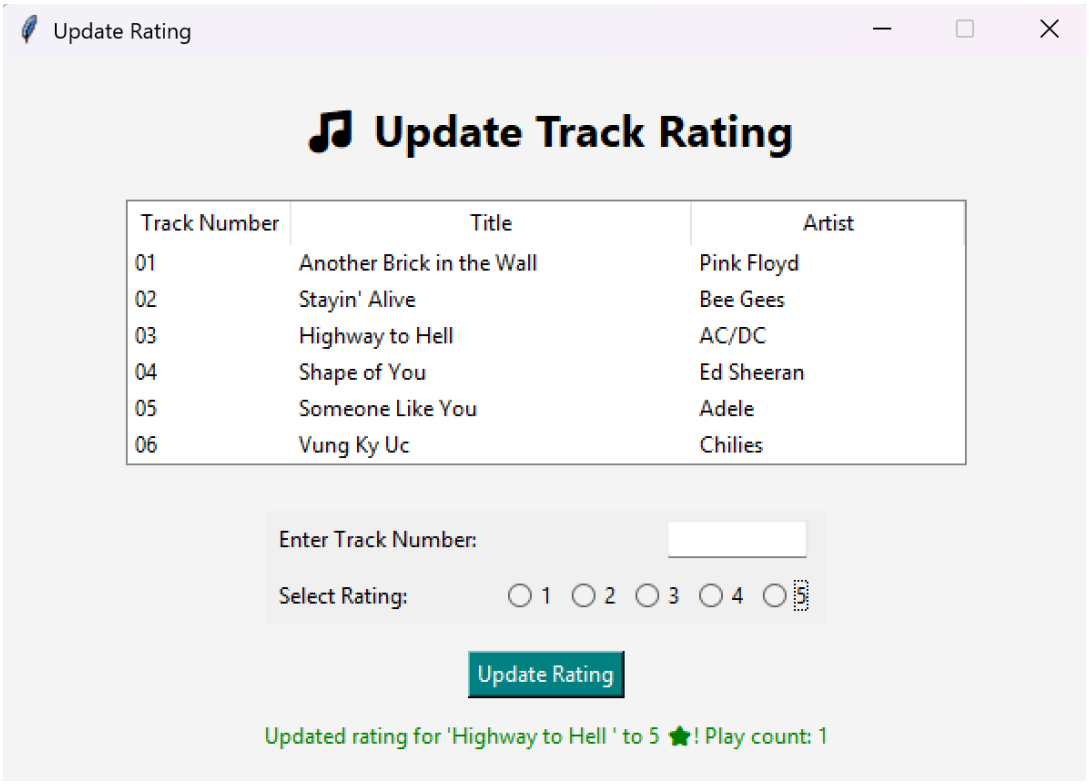
- Evidence 10: Load playlist



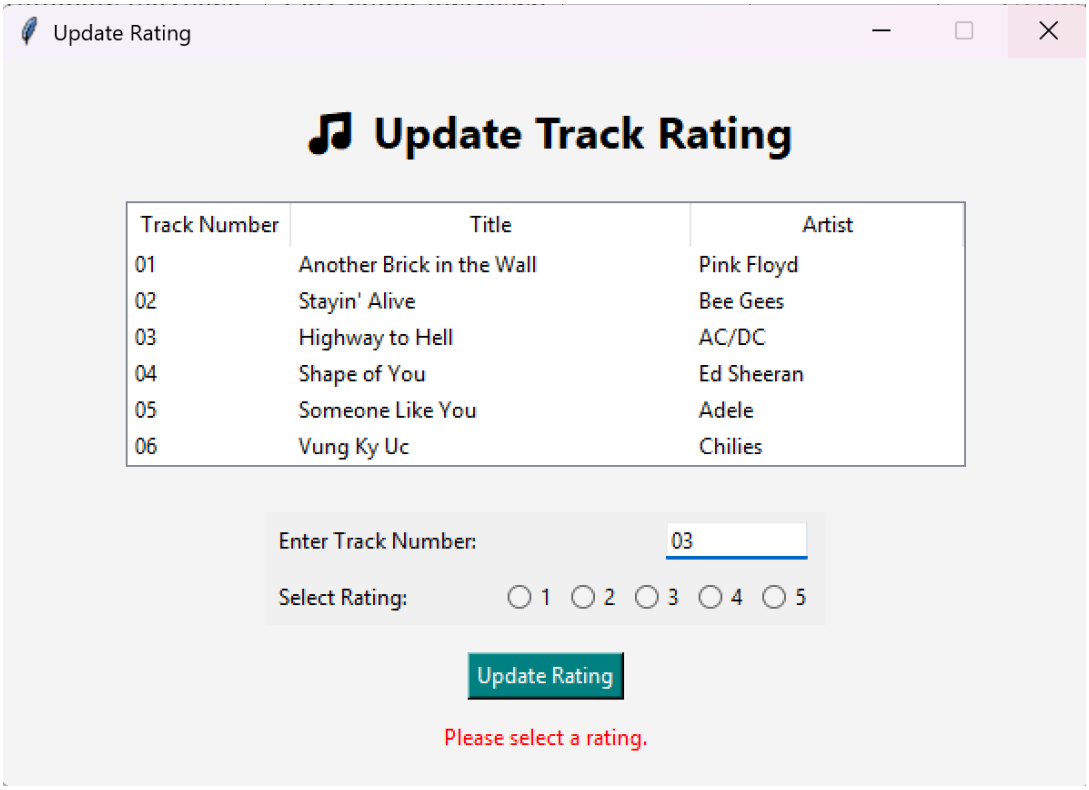
- Evidence 11: Simulate playback



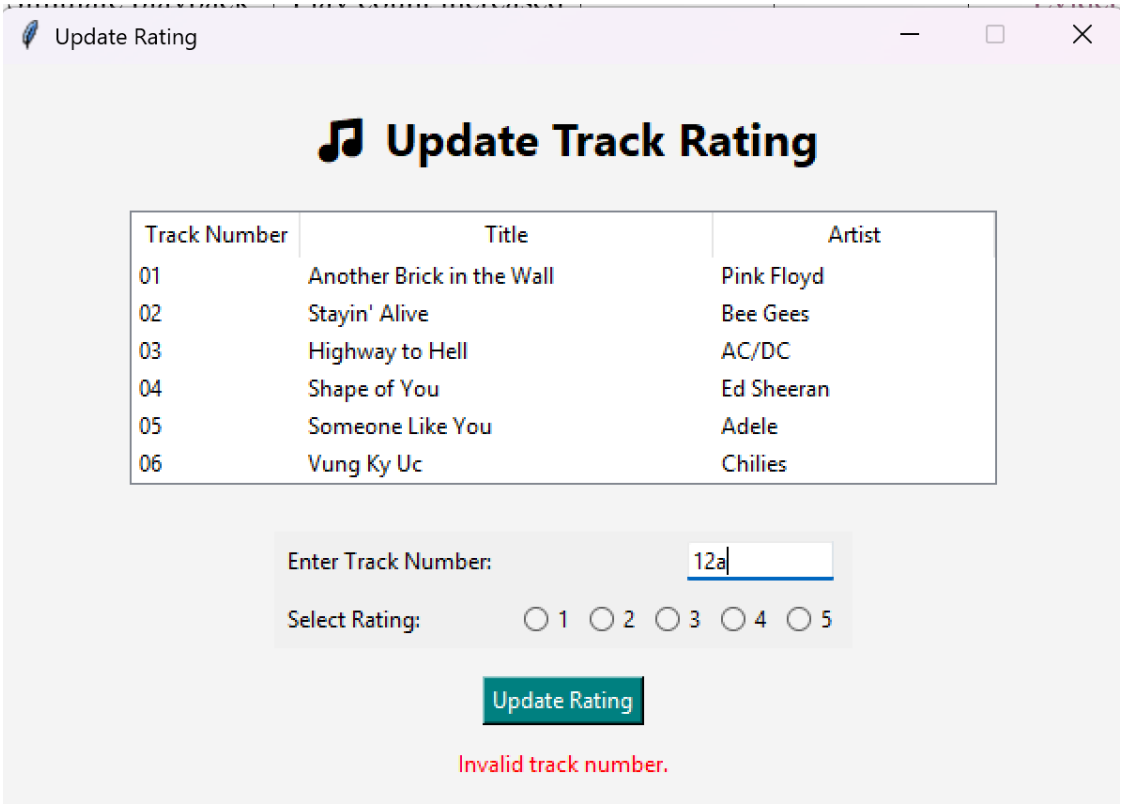
- Evidence 12: Update rating success



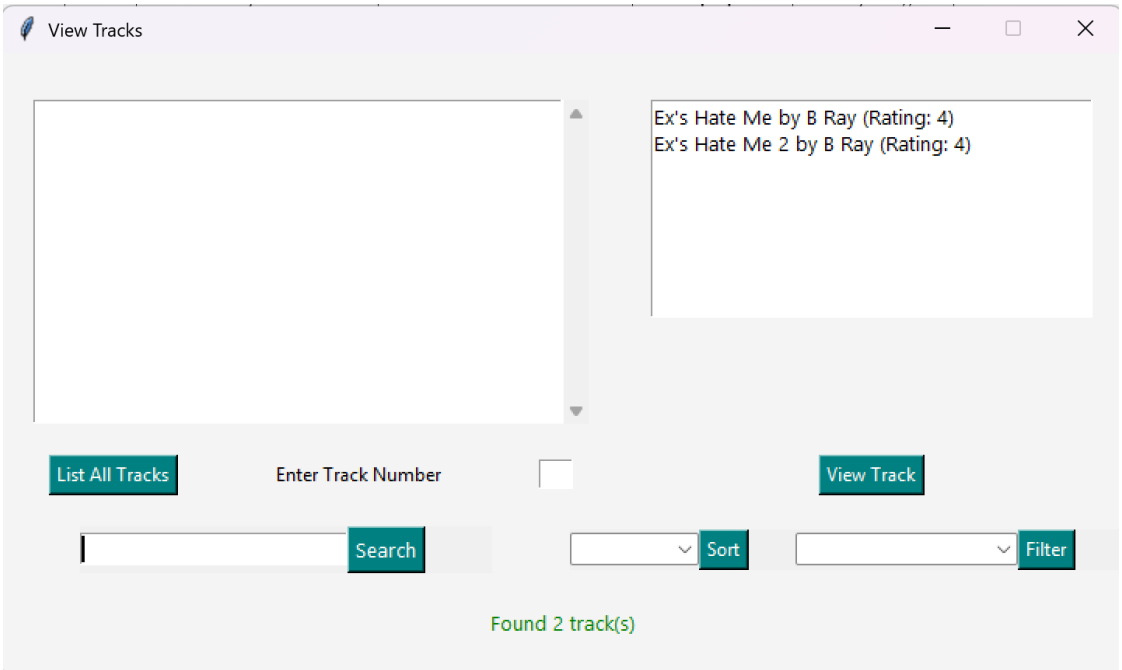
- Evidence 13: Update without rating



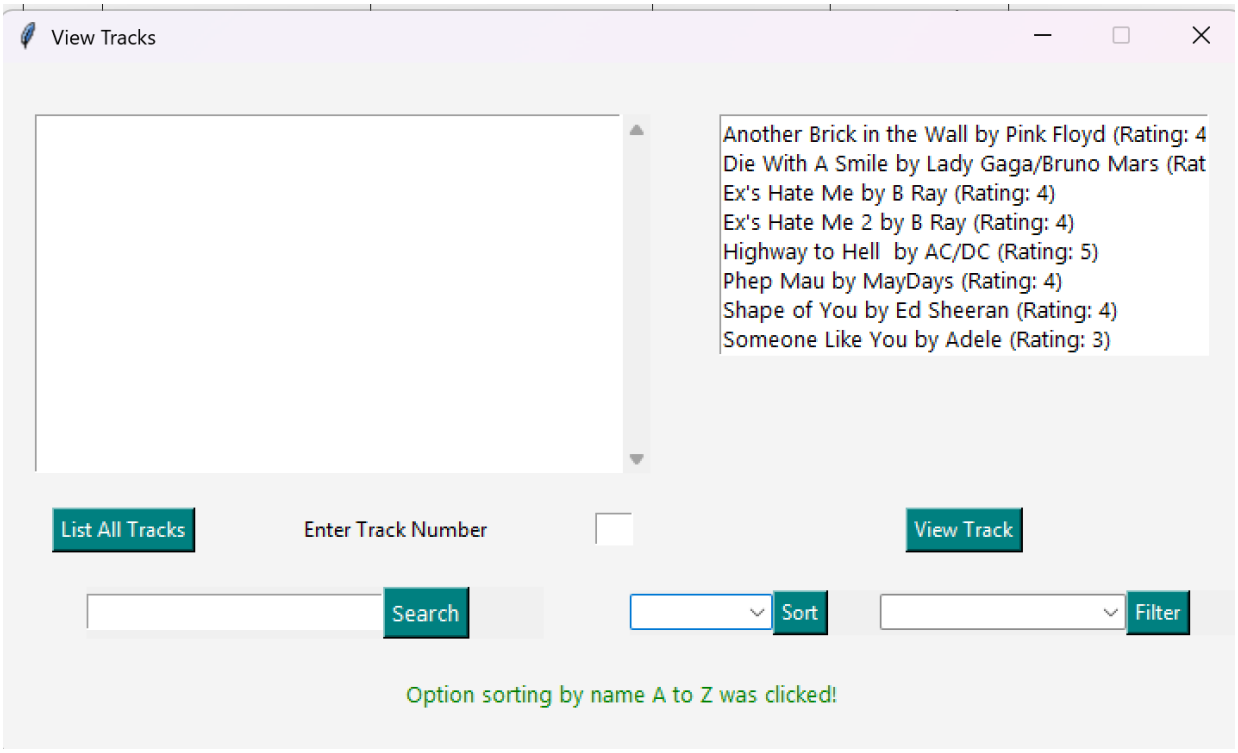
- Evidence 14: Update with invalid ID



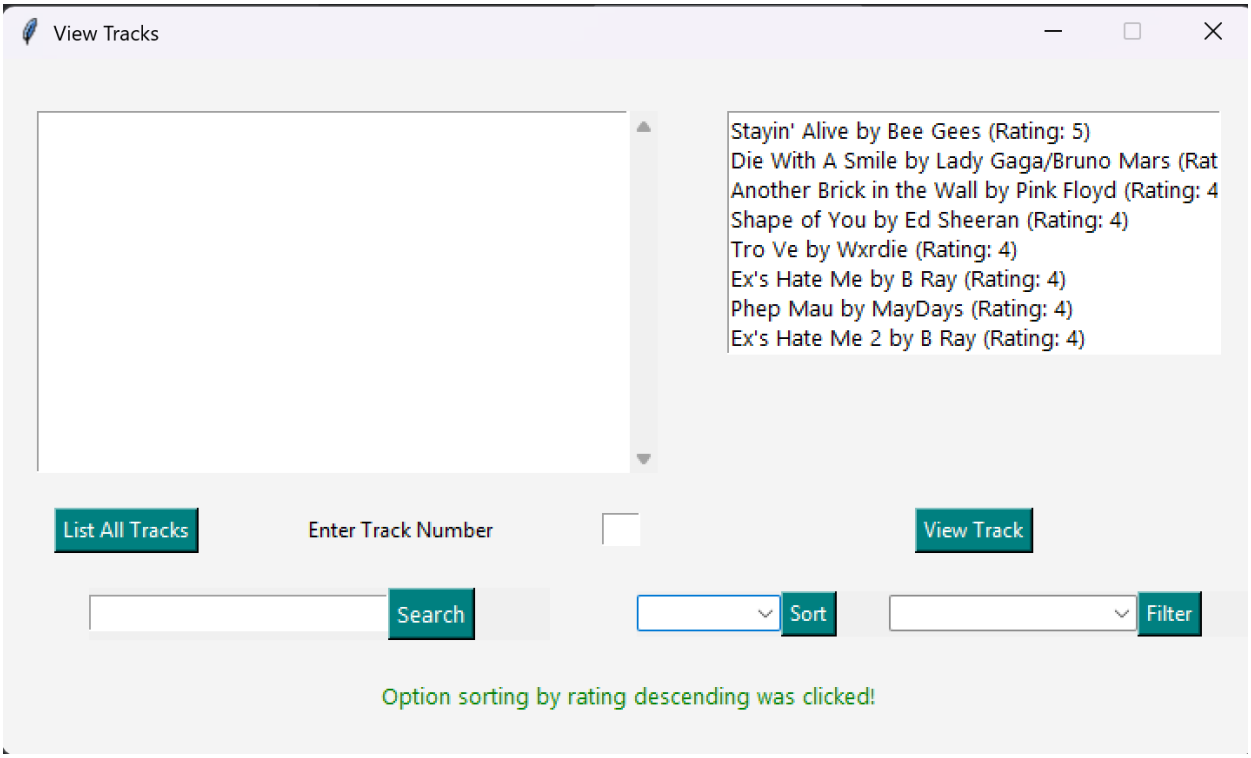
- Evidence 15: Search track by keyword



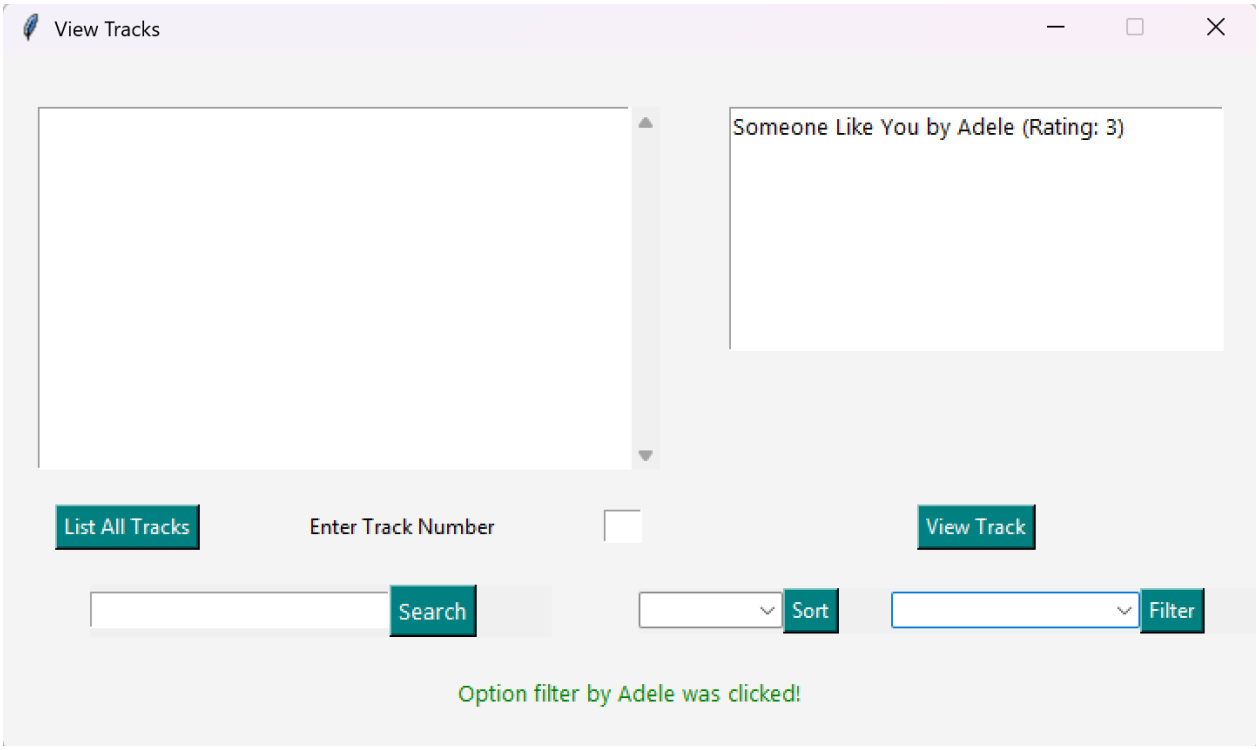
- Evidence 16: Sort by name A–Z



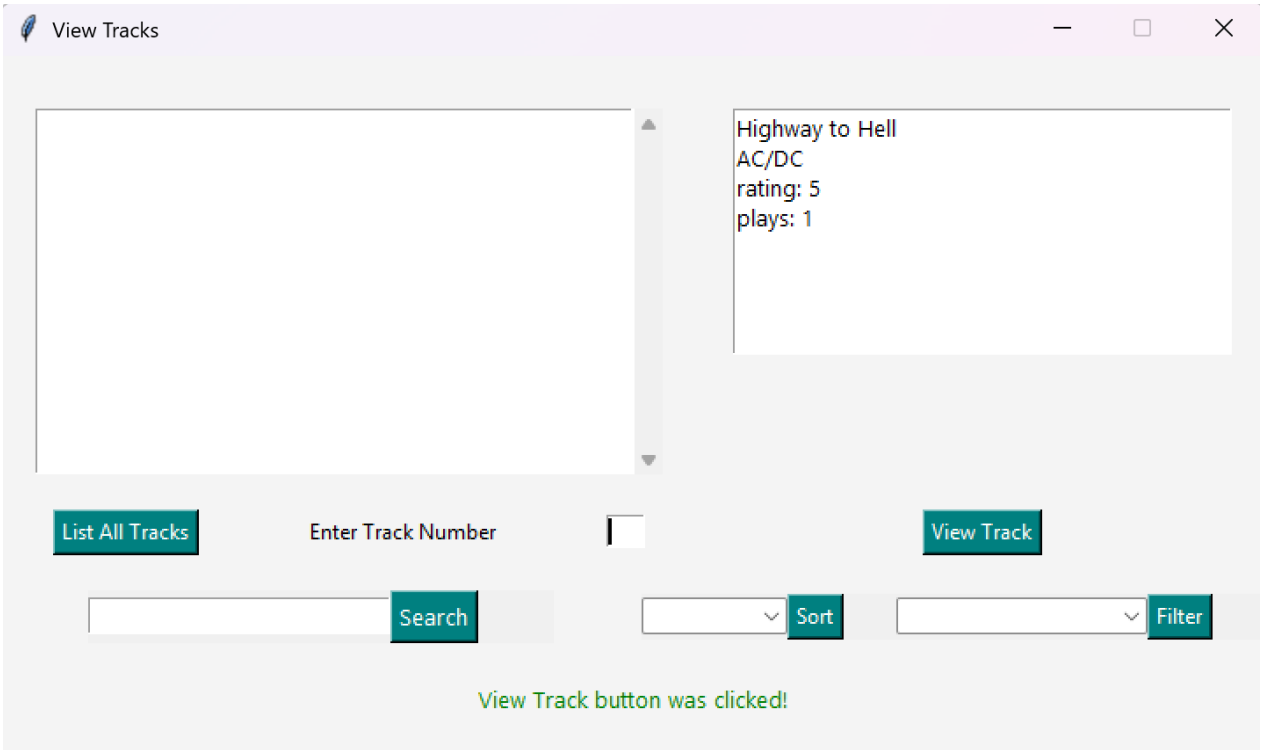
- Evidence 17: Sort by rating descending



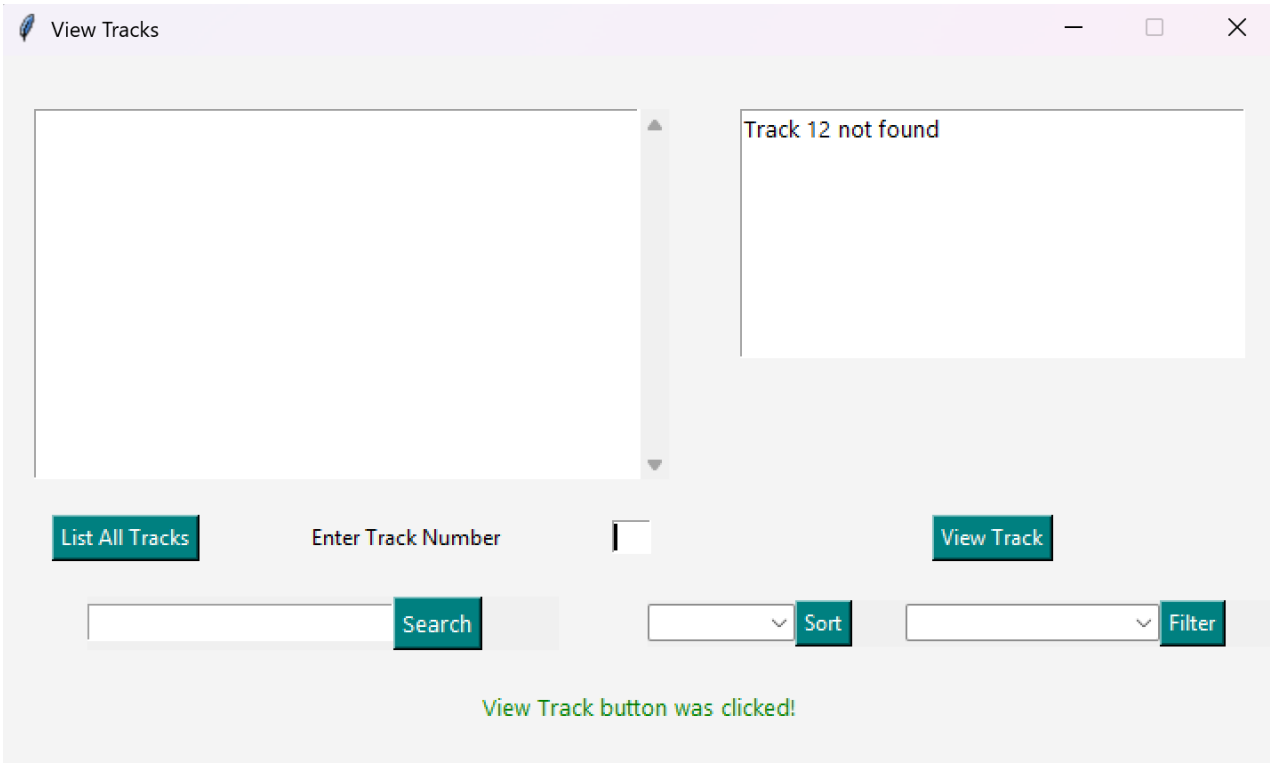
- Evidence 18: Filter by artist



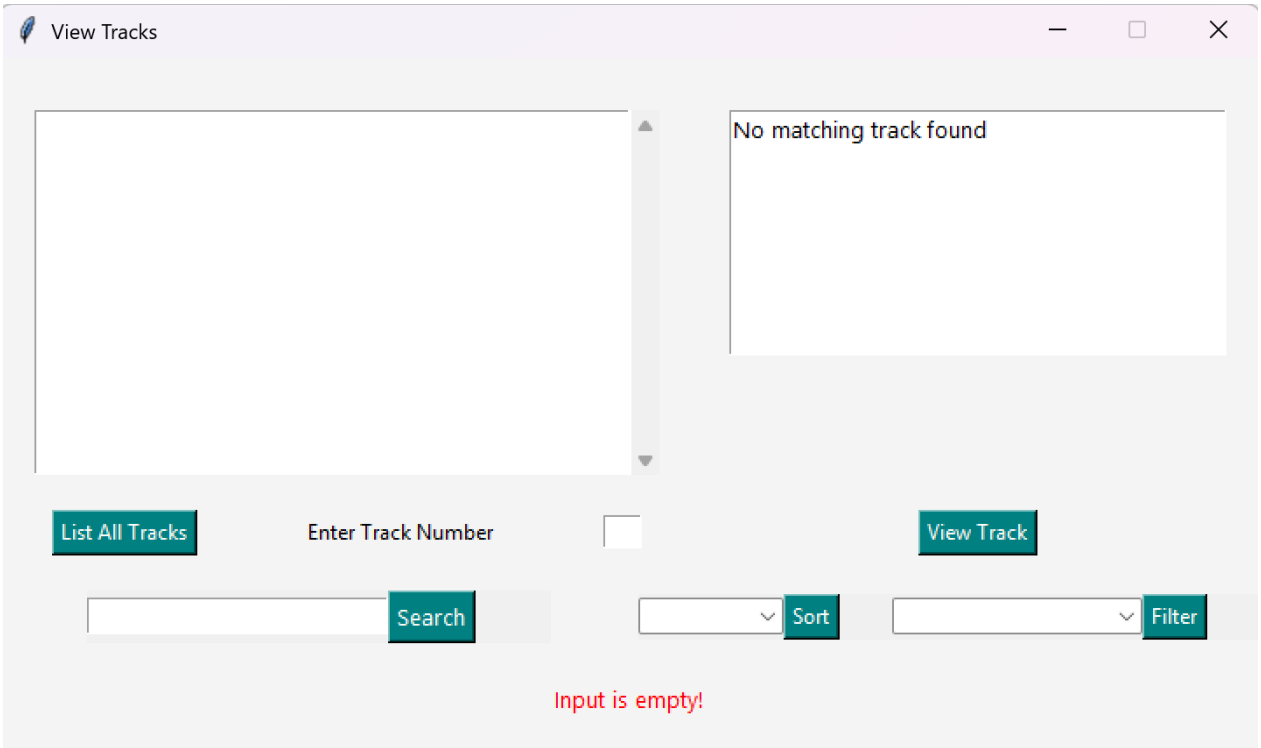
- Evidence 19: View track by ID



- Evidence 20: View track by invalid ID



- Evidence 21: Search with no entry



- Evidence 22: Test info(), starts() functions and default constructor

```
============================= test session starts ==============================
collecting ... collected 3 items

test_library_item.py::test_info PASSED                                   [ 33%]
test_library_item.py::test_starts PASSED                                 [ 66%]
test_library_item.py::test_default_constructor PASSED                    [100%]

============================== 3 passed in 0.01s ===============================

Process finished with exit code 0
```

- Evidence 23: Test get_name(), get_artist(), get_rating(), increment_play_count() functions

```
============================= test session starts ==============================
collecting ... collected 10 items

test_track_library.py::test_get_name PASSED                              [ 10%]
test_track_library.py::test_get_name_invalid PASSED                      [ 20%]
test_track_library.py::test_get_artist PASSED                            [ 30%]
test_track_library.py::test_get_artist_invalid PASSED                    [ 40%]
test_track_library.py::test_get_rating PASSED                            [ 50%]
test_track_library.py::test_get_rating_invalid PASSED                    [ 60%]
test_track_library.py::test_set_rating_valid PASSED                      [ 70%]
test_track_library.py::test_set_rating_invalid PASSED                    [ 80%]
test_track_library.py::test_increment_play_count PASSED                  [ 90%]
test_track_library.py::test_get_play_count_invalid PASSED                [100%]

============================== 10 passed in 0.02s ==============================

Process finished with exit code 0
```

- Evidence 24: Source code of create track list

```
from tkinter import ttk
import tkinter as tk
import track_library as lib
from font_manager import get_h1_font
import tkinter.scrolledtext as tkst
from track_library import increment_play_count, get_name, get_artist,
get_rating, library
import json


def insert_text(text_area, content):    #Append content to the end of a text
widget, followed by a newline
    text_area.insert(tk.END, content)    #Insert new content at the end of the
```

```
text
    text_area.insert(tk.END, '\n')      #Insert a newline character at the
end

def set_text(text_area, content):       #Clear and set content for a text
widget
    text_area.delete("1.0", tk.END)     #Clear all content in the text area
    text_area.insert(1.0, content)      #Insert new content at the beginning
(line 1, char 0)



class CreateTrackList:
    def __init__(self, window):
        self.window = window                    #Assign a window object
        self.window.geometry("720x400")         # Set window size
        self.window.configure(bg="#f4f4f4")     #Set background color
        self.window.title("Track Manager")      #Set window title
        self.window.resizable(False, False)     #Disable resizing
        self.id_lib = []                        #Store track IDs added to
playlist
        self.file_path = "playlist.json"        #File to save/load playlist
from

        self.create_widgets()   #call function to create GUI

    def create_widgets(self):
        title_lbl = tk.Label(self.window, text="🎵 Create Playlist",
font=get_h1_font(), bg="#f4f4f4")       #Create and locate main label
        title_lbl.grid(row=0, column=0, pady=(10, 5), padx=(40,0))

        self.list_txt = tkst.ScrolledText(self.window, width=46, height=10,
wrap="none", font=("Segoe UI", 10))     #Create and locate text area to
display playlist tracks
        self.list_txt.grid(row=1, column=0, columnspan=3, sticky="w",
padx=30, pady=(5, 10))


        add_track_frame = ttk.Frame(master=self.window)     #Create and
locate a frame for track input and Add button
        add_track_frame.grid(row=2, column=0, columnspan=2, sticky="w",
pady=20, padx=30)

        input_lbl = tk.Label(add_track_frame, text="Enter track number:",
font=("Segoe UI", 10), bg="#f4f4f4")      #Create and locate label and entry
for track number input
        input_lbl.pack(side="left", padx=(0,20))

        self.track_txt = tk.Entry(add_track_frame, width=12, font=("Segoe
UI", 10))     #Create Entry to get user input
        self.track_txt.pack(side="left", padx=(0,20))
        self.track_txt.focus()      #Automatically focus when window opens

        self.add_track_btn = tk.Button(add_track_frame, text="Add Track",
command=self.add_track_clicked, font=("Segoe UI", 10), bg="#008080",
fg="white")       #Create and locate a button to add track to playlist (binds
```

```
to add_track_clicked)
        self.add_track_btn.pack(side="left")

        self.save_btn = tk.Button(self.window, text="Save playlist",
command=self.save_clicked, font=("Segoe UI", 10), bg="#008080", fg="white")
#Create and locate a button to save current playlist to JSON file (binds to
save_clicked)
        self.save_btn.grid(row=0, column=2, padx=(20,0), pady=(40,0))

        self.view_playlist_btn = tk.Button(self.window, text="View playlist",
command=self.load_clicked, font=("Segoe UI", 10), bg="#008080", fg="white")
#Create and locate a button to load playlist from file (binds to
load_clicked)
        self.view_playlist_btn.grid(row=0, column=3, padx=(40,0),
pady=(40,0))

        self.play_btn = tk.Button(self.window, text="Play playlist",
command=self.play_click, font=("Segoe UI", 10), bg="#008080", fg="white")
#Create and locate a button to simulate "playing" the playlist (binds to
play_click)
        self.play_btn.grid(row=1, column=2, padx=(20,0))

        self.reset_btn = tk.Button(self.window, text="Reset playlist",
command=self.reset_clicked, font=("Segoe UI", 10), bg="#008080", fg="white")
#Create and locate a button to reset the playlist (binds to reset_clicked)
        self.reset_btn.grid(row=1, column=3, padx=(20,0))

        self.status_lbl = tk.Label(self.window, text="status", font=("Segoe
UI", 10), bg="#f4f4f4")  #Create and locate a label to follow status
        self.status_lbl.grid(row=3, column=0, columnspan=4, sticky="we",
padx=10, pady=10)



    def add_track_clicked(self):    #A function to add a track
        key = self.track_txt.get()      #Get track number from input field
        if not key:     #Check if input is empty
            self.status_lbl.configure(text="Input is empty!", fg="red")
#inform the status
            return      #Exit the loop if needed

        if key in self.id_lib:      #Check if input is already in id_lid
            self.status_lbl.configure(text=f"Track {key} already in
playlist!", fg="orange")
            return
        else:
            if key in library:       #Check if the entered track ID exists in
the music library
                self.id_lib.append(key)  #Add track ID to id_lib
                track_details = f"{get_name(key)} by {get_artist(key)} -
rating: {get_rating(key)}"     #Get track's detail
                self.status_lbl.configure(text=f"Track {key} added",
fg="green")    #inform status
                insert_text(self.list_txt, track_details)  #Display the
content in list_txt
            else:
                self.status_lbl.configure(text=f"Track {key} not found!",
```

```
fg="red")     #Inform if track not found

        self.track_txt.delete(0, tk.END)    # Clear input field after attempt

    def save_clicked(self):     #Function to save the current playlist to a
JSON file.
        if self.id_lib:     #Check if id_lib is not empty
            with open(self.file_path, "w") as file:     #Open with file_path
and write to JSON file
                json.dump(self.id_lib, file, indent=4)      #Save track IDs
as JSON list
                self.status_lbl.configure(text="Playlist saved", fg="green")
#Inform the status if saved
        else:
            self.status_lbl.configure(text="Please add a track fist!",
fg="red")    #Inform the status if list_txt is empty


    def load_clicked(self):     #Function to display playlist from JSON file
to the list_txt
        self.id_lib.clear()     #Clear current playlist
        with open(self.file_path, 'r') as file:
            data = json.load(file)      #Read track ID list from file
            load_track = []         #Create a list to store track in JSON
file
            for trackId in data:
                load_track.append(f"{get_name(trackId)} by
{get_artist(trackId)} - rating: {get_rating(trackId)}")      #Add track
information to load_track
                self.id_lib.append(trackId)     #Add track ID to id_lib
            set_text(self.list_txt, '\n'.join(load_track) + '\n')
#Display the content in list_txt
            self.status_lbl.configure(text="Playlist loaded successfully!",
fg="green")     #Inform the status


    def play_click(self):
        if self.id_lib:     #Check if it is empty
            for key in self.id_lib:
                increment_play_count(key)       #Call function to increase
play count
            self.status_lbl.configure(text="Play successfully!")    #Inform
status

        else:
            self.status_lbl.configure(text="Please add a track first!",
fg="red")       #Inform status

    def reset_clicked(self):    #Function to clear all data in list_txt
        self.list_txt.delete(1.0, tk.END)       #Delete data
        self.id_lib.clear()  # Delete track IDs in id_lib
        self.status_lbl.configure(text="Reset successfully!", fg="green")
#Inform status


if __name__ == "__main__":      # only runs when this file is run as a
standalone
```

```
    window = tk.Tk()            # create a TK object
    CreateTrackList(window)     # open the CreateTrackList GUI
    window.mainloop()           # run the window main loop, reacting to
button presses, etc
```

- Evidence 25: Source code of View tracks

```
import tkinter as tk
from tkinter import ttk
import tkinter.scrolledtext as tkst
import track_library as lib
from track_library import get_artist, get_rating, get_name, library,
get_play_count


def set_text(text_area, content):
    text_area.delete("1.0", tk.END)     # Clear all content in the text area
    text_area.insert(1.0, content)      # Insert new content at the beginning
(line 1, char 0)


class TrackViewer():
    def __init__(self, window):     #Constructor for the TrackViewer GUI
class
        self.window = window        #Store reference to the parent window
        self.window.geometry("720x400")         #Set size for the window
        self.window.configure(bg="#f4f4f4")     #Set background color
        self.window.title("View Tracks")        #Set window title
        self.window.resizable(False, False)     #Disable resizing


        self.list_txt = tkst.ScrolledText(window, width=48, height=12,
wrap="none", font=("Segoe UI", 10))      #Create and locate text box with
scrollbar to show all tracks
        self.list_txt.grid(row=0, column=0, columnspan=3, sticky="W",
padx=20, pady=(30, 10))

        self.track_txt = tk.Text(window, width=40, height=8, wrap="none",
font=("Segoe UI", 10))        #Another text box to show details of selected
or searched track
        self.track_txt.grid(row=0, column=3, sticky="NW", padx=20, pady=(30,
10))


        self.list_tracks_btn = tk.Button(window, text="List All Tracks",
command=self.list_tracks_clicked, bg="#008080", fg="white")        #Button to
list all available tracks
        self.list_tracks_btn.grid(row=1, column=0, padx=10, pady=10)

        enter_lbl = tk.Label(window, text="Enter Track Number", bg="#f4f4f4")
#Label for the track number entry
        enter_lbl.grid(row=1, column=1, padx=10, pady=10)

        self.input_txt = tk.Entry(window, width=3)      #Create and locate
entry field to input track number
        self.input_txt.grid(row=1, column=2, padx=10, pady=10)
```

```
        self.check_track_btn = tk.Button(window, text="View Track",
command=self.view_tracks_clicked, bg="#008080", fg="white")     #Button to
show details of the track with entered ID
        self.check_track_btn.grid(row=1, column=3, padx=10, pady=10)


        search_frame = ttk.Frame(master=self.window)        #Create a sub-
frame to hold the search entry and button
        search_frame.grid(row=2, column=0, columnspan=2, sticky="we",
padx=(50,0), pady=10)

        self.search_txt = tk.Entry(search_frame, width=24, font=("Segoe UI",
10))        #Create and locate entry field to get user input
        self.search_txt.pack(side="left")
        self.search_txt.focus()      #Set focus to the search input

        self.search_btn = tk.Button(search_frame, text="Search",
command=self.search_track ,font=("Segoe UI", 10), bg="#008080", fg="white")
#Button to trigger search
        self.search_btn.pack(side="left")

        groups_sorting = ["", "rating ↑", "rating ↓", "a-z", "z-a"]     #List
of sort options

        sort_frame = ttk.Frame(window)      #Create and locate frame for both
sorting and filtering controls
        sort_frame.grid(row=2, column=2, columnspan=2, sticky="we",
padx=(50,0), pady=10)

        self.combo_sort = ttk.Combobox(sort_frame, width=10)        #Create a
combobox
        self.combo_sort["values"] = groups_sorting                  #Set the
option in groups_sorting
        self.combo_sort.current(0)              #Set default selection
(empty)
        self.combo_sort.pack(side="left")       #Place the sorting combobox

        self.check_combobox = tk.Button(sort_frame, text="Sort",
command=self.sort_combobox_clicked, bg="#008080", fg="white")      #Button to
apply selected sort option
        self.check_combobox.pack(side="left", padx=(0,30))

        groups_filter = [""]        #Initialize artist filter list with an
empty option
        for key in lib.library:
            if get_artist(key) not in groups_filter:    #Avoid duplicates
                groups_filter.append(get_artist(key))   #Add artist

        self.combo_filter = ttk.Combobox(sort_frame)        #Create a
combobox
        self.combo_filter["values"] = groups_filter         #Set the option
in groups_filter
        self.combo_filter.current(0)                        #Set default
selection (empty)
        self.combo_filter.pack(side="left")                 #Place the
sorting combobox
```

```
        self.check_combo_filter = tk.Button(sort_frame, text="Filter",
command=self.filtered_combobox_clicked, bg="#008080", fg="white")
#Create and locate a button binds to filtered_combobox_clicked
        self.check_combo_filter.pack(side="left")


        self.status_lbl = tk.Label(window, text="", font=("Segoe UI", 10),
bg="#f4f4f4")        #Create a label to follow status
        self.status_lbl.grid(row=3, column=0, columnspan=4, sticky="we",
padx=10, pady=10)      #Locate the label


    def view_tracks_clicked(self):          #Called when "View Track" button
is clicked
        key = self.input_txt.get()          #Get track ID from input field
        if not key:     #Check if it is empty
            self.status_lbl.configure(text="Input is empty!", fg="red")
            return      #Exit loop
        else:
            if key in library:      #Check if it is empty
                track_details = f"{get_name(key)}\n{get_artist(key)}\nrating:
{get_rating(key)}\nplays: {get_play_count(key)}"      #Get track's detail
                set_text(self.track_txt, track_details)     #Display track
information in track_txt
            else:
                set_text(self.track_txt, f"Track {key} not found")
#inform if key is not found

        self.input_txt.delete(0, tk.END)        #Clear input_txt (from first
letter to end)
        self.status_lbl.configure(text="View Track button was clicked!",
fg="green")    #Display status

    def list_tracks_clicked(self):              #Called when "List All
Tracks" button is clicked
        track_list = lib.list_all()             #Get string containing all
track info
        set_text(self.list_txt, track_list)     #Display in left ScrolledText
        self.status_lbl.configure(text="List Tracks button was clicked!",
fg="green")       #Update status

    def search_track(self):     #Called when "Search" button is clicked
        search_input = self.search_txt.get().strip().lower()        #Get
input and normalize it

        if search_input == "":      #If input is empty
            set_text(self.track_txt, "No matching track found")     #Display
content in right text box
            self.status_lbl.configure(text="Input is empty!", fg="red")
#Update status
        else:
            filtered_tracks = [track for track in lib.library.values()
                               if search_input in track.name.lower() or
search_input in track.artist.lower()]       #Filter tracks that match input
(in name or artist)

            if filtered_tracks:     # Found matches
```

```
                lines = [f"{track.name} by {track.artist} (Rating:
{track.rating})" for track in filtered_tracks]
                set_text(self.track_txt, '\n'.join(lines))      #Show results
                self.status_lbl.configure(text=f"Found {len(filtered_tracks)}
track(s)", fg="green")
                self.search_txt.delete(0, tk.END)       #Clear search entry
            else:
                set_text(self.track_txt, "No matching track found")
                self.status_lbl.configure(text="No matching track found",
fg="red")


    def sort_combobox_clicked(self):        #Called when "Sort" button is
clicked
        group = self.combo_sort.get()       #Get selected sort option
        if group == "":
            self.status_lbl.configure(text="Please select a sorting method
first!", fg="red")
        elif group == "rating ↑":
            self.ascending_sort()      #Sort by rating (highest to lowest)
            self.status_lbl.configure(text="Option sorting by rating
ascending was clicked!", fg="green")
            self.combo_sort.current(0)  #Reset combobox selection
        elif group == "rating ↓":
            self.descending_sort()       #Sort by rating (lowest to highest)
            self.status_lbl.configure(text="Option sorting by rating
descending was clicked!", fg="green")
            self.combo_sort.current(0)
        elif group == "a-z":
            self.alphabet_sort()        #Sort by name A-Z
            self.status_lbl.configure(text="Option sorting by name A to Z was
clicked!", fg="green")
            self.combo_sort.current(0)
        elif group == "z-a":
            self.alphabet_reserve()     #Sort by name Z-A
            self.status_lbl.configure(text="Option sorting by name Z to A was
clicked!", fg="green")
            self.combo_sort.current(0)

    def ascending_sort(self):
        sorted_tracks = sorted(lib.library.values(), key=lambda x: x.rating)
#Use lambda to extract 'rating' attribute as sorting key
        self.display_sorted(sorted_tracks)      #Display the sorted list
    def descending_sort(self):
        sorted_tracks = sorted(lib.library.values(), key=lambda x: x.rating,
reverse=True)       #'reverse=True' reverses the order of the sorted list
        self.display_sorted(sorted_tracks)
    def alphabet_sort(self):
        sorted_tracks = sorted(lib.library.values(), key=lambda x:
x.name.lower())      #Convert to lowercase to ensure case-insensitive sorting
        self.display_sorted(sorted_tracks)
    def alphabet_reserve(self):
        sorted_tracks = sorted(lib.library.values(), key=lambda x:
x.name.lower(), reverse=True)
        self.display_sorted(sorted_tracks)

    def filtered_combobox_clicked(self):        #Called when "Filter" button
```

```
is clicked
        group = self.combo_filter.get()         #Get selected artist name
        if group == "":
            self.status_lbl.configure(text="Please select a filter first!",
fg="red")
        else:
            self.artist_filter(group)       # Filter by selected artist
            self.status_lbl.configure(text=f"Option filter by {group} was
clicked!", fg="green")
            self.combo_filter.current(0)     #Reset combobox selection

    def artist_filter(self, filter_get_txt):    #Called to filter tracks by
artist name
        filtered_list = [track for track in lib.library.values() if
filter_get_txt == track.artist]     #Get track information
        self.display_sorted(filtered_list)      #Show matching tracks
        self.status_lbl.configure(text=f"Found {len(filtered_list)} track(s)
by {filter_get_txt}", fg="green")


    def display_sorted(self, sorted_tracks):    #Display tracks in right-side
text box
        lines = []      #Initialize an empty list to store formatted track
strings
        for track in sorted_tracks:
            lines.append(f"{track.name} by {track.artist} (Rating:
{track.rating})")    #Format each track as a string and add it to the list
        set_text(self.track_txt, '\n'.join(lines))      # Show all results


if __name__ == "__main__":  # only runs when this file is run as a standalone
    window = tk.Tk()        # create a TK object
    TrackViewer(window)     # open the TrackViewer GUI
    window.mainloop()       # run the window main loop, reacting to button
presses, etc
```

- Evidence 26: Source code of update track rating

```
import tkinter as tk
from tkinter import ttk
from track_library import library
from track_library import set_rating, get_name, get_play_count, get_artist
from font_manager import get_h1_font

class UpdateRating:
    def __init__(self, window):
        self.window = window    #Store the main window reference as an
instance variable for later use
        self.window.title("Update Rating")  #Set name title
        self.window.geometry("600x400")     #Set size for window
        self.window.resizable(False, False)     #Don't allow to resize
        self.window.configure(bg="#f4f4f4")     #Set background color

        self.library = library  #get all data from library

        self.create_widgets()   #call function to create GUI
```

```
    def create_widgets(self):
        title_lbl = tk.Label(self.window, text="🎵 Update Track Rating",
font=get_h1_font(), bg="#f4f4f4")
        title_lbl.pack(pady=(20, 5))

        self.tree = ttk.Treeview(self.window, columns=("order", "title",
"artist"), show="headings", height=6, selectmode="none")  # Create a Treeview
widget to display a list of tracks with 3 columns: Track Number, Title, and
Artist
        self.tree.heading("title", text="Title")            #Define column
header for track title
        self.tree.heading("artist", text="Artist")          #Define column
header for artist name
        self.tree.heading("order", text="Track Number")     #Define column
header for track number (key)
        #Set the width for each column
        self.tree.column("title", width=220)
        self.tree.column("artist", width=150)
        self.tree.column("order", width=90)

        self.tree.pack(pady=15)     #Add vertical padding around the widget

        self.populate_tree()    #Call method to insert data into the table

        input_frame = ttk.Frame(master= self.window)    #create a frame to
put all the input buttons in
        input_frame.pack(pady=10)   #Add vertical padding around the
input_frame

        ttk.Label(input_frame, text="Enter Track Number:").grid(row=0,
column=0, padx=5, pady=5, sticky="we")   #Create and locate a label
        self.track_num_var = tk.StringVar()     #create a string variable to
store user input
        self.track_num_input = ttk.Entry(input_frame,
textvariable=self.track_num_var, width=12)    #Create an input field for
track number (binds to self.track_num_var)
        self.track_num_input.grid(row=0, column=1, padx=10, pady=5,
sticky="e")     #locate the track_num_input
        self.track_num_input.focus()    # Place the cursor in the input box

        ttk.Label(input_frame, text="Select Rating:").grid(row=1, column=0,
padx=5, pady=5, sticky="we")    #Create and locate a label in input_frame

        rating_frame = ttk.Frame(input_frame)   #create a frame to put all
the radio buttons in
        rating_frame.grid(row=1, column=1, padx=5, pady=5, sticky="e")
#locate the rating_frame

        self.rating_var = tk.IntVar()   #create an integer variable to store
user selection
        for i in range(1, 6):
            ttk.Radiobutton(rating_frame, text=str(i),
variable=self.rating_var, value=i).pack(side="left", padx=3)     #use a loop
to create 5 radio buttons in a row

        self.update_btn = tk.Button(self.window, text="Update Rating",
```

```
bg="#008080", fg="white", command=self.update_rating)    #create Update
Rating button
        self.update_btn.pack(pady=5)

        self.status_lbl = tk.Label(self.window, text="", fg="green",
bg="#f4f4f4")      #create label to inform status
        self.status_lbl.pack(pady=5)

    def populate_tree(self):    #Insert each track into the Treeview with its
ID, name, and artist
        for key in self.library:
            self.tree.insert('', 'end', values=(key, get_name(key),
get_artist(key)))

    def update_rating(self):    #update rating function
        selected = self.track_num_var.get()     #get user input value

        if not selected:    #check if it is empty
            self.status_lbl.config(text="Please enter a track first.",
fg="red")
            return

        if selected not in self.library:    #check if it not exist in library
            self.status_lbl.config(text="Invalid track number.", fg="red")
            return

        new_rating = self.rating_var.get()      #get data
        if new_rating == 0:     #check if user don't select
            self.status_lbl.config(text="Please select a rating.", fg="red")
            return

        set_rating(selected, new_rating)    #set new rating value
        self.status_lbl.config(text=f"Updated rating for
'{get_name(selected)}' to {new_rating} ★! Play count:
{get_play_count(selected)}", fg="green")     #print new rating status and
play count

        self.track_num_var.set("")  # clear track number entry
        self.rating_var.set(0)      # clear rating selection


if __name__ == "__main__":  # only runs when this file is run as a standalone
    window = tk.Tk()        # create a TK object
    UpdateRating(window)    # open the UpdateRating GUI
    window.mainloop()       # run the window main loop, reacting to button
presses, etc
```

- Evidence 27: Source code of font manager

```
import tkinter.font as tkfont


def configure():    #This function customizes the default font styles
    # family = "Segoe UI"
    family = "Helvetica"    #The current default font being applied
    default_font = tkfont.nametofont("TkDefaultFont")   #Set the default font
for widgets like Button, Label, etc.
```

```
    default_font.configure(size=15, family=family)
    text_font = tkfont.nametofont("TkTextFont")      #Set the default font for
widgets like Text and ScrolledText.
    text_font.configure(size=12, family=family)
    fixed_font = tkfont.nametofont("TkFixedFont")    #Get the font for Fixed-
Width text (like code or monospaced text)
    fixed_font.configure(size=12, family=family)

def get_h1_font(root=None):      #This function returns a bold, large-sized
font suitable for main headers
    return tkfont.Font(root=root, family="Segoe UI", size=18, weight="bold")
```

- Evidence 28: Source code of library item

```
class LibraryItem:
    def __init__(self, name, artist, rating=0):      #create constructor
        self.name = name
        self.artist = artist
        self.rating = rating
        self.play_count = 0

    def info(self):      #Return a string combining track name, artist, and
visual star rating
        return f"{self.name} - {self.artist} {self.stars()}"

    def stars(self):      #Generates a string of asterisks (*) representing a
rating
        stars = ""
        for i in range(self.rating):
            stars += "*"
        return stars
```

- Evidence 29: Source code of track library

```
from library_item import LibraryItem

library = {}     #Create a dictionary to store data

#Initialize the music library
library["01"] = LibraryItem("Another Brick in the Wall", "Pink Floyd", 4)
library["02"] = LibraryItem("Stayin' Alive", "Bee Gees", 5)
library["03"] = LibraryItem("Highway to Hell ", "AC/DC", 2)
library["04"] = LibraryItem("Shape of You", "Ed Sheeran", 4)
library["05"] = LibraryItem("Someone Like You", "Adele", 3)
library["06"] = LibraryItem("Vung Ky Uc", "Chilies", 3)
library["07"] = LibraryItem("Die With A Smile", "Lady Gaga/Bruno Mars", 5)
library["08"] = LibraryItem("Tro Ve", "Wxrdie",4)
library["09"] = LibraryItem("Ex's Hate Me","B Ray", 4)
library["10"] = LibraryItem("Phep Mau", "MayDays", 4)
library["11"] = LibraryItem("Ex's Hate Me 2","B Ray", 4)



def list_all():      #Returns a string containing all the track information
    output = ""
    for key in library:
        item = library[key]
        output += f"{key} {item.info()}\n"
```

```
    return output


def get_name(key):      #Return name of song
    try:
        item = library[key]
        return item.name
    except KeyError:
        return None


def get_artist(key):    #Return artist of song
    try:
        item = library[key]
        return item.artist
    except KeyError:
        return None


def get_rating(key):    #Return rating of song
    try:
        item = library[key]
        return item.rating
    except KeyError:
        return -1


def set_rating(key, rating):    #Set rating of song
    try:
        item = library[key]
        item.rating = rating
    except KeyError:
        return


def get_play_count(key):    #Return play count
    try:
        item = library[key]
        return item.play_count
    except KeyError:
        return -1


def increment_play_count(key):      #Plus 1 for play
    try:
        item = library[key]
        item.play_count += 1
    except KeyError:
        return
```

- Evidence 30: Source code of track player

```
import tkinter as tk
from view_tracks import TrackViewer
from update_tracks import UpdateRating
from create_track_list import CreateTrackList
```

```
def view_tracks_clicked():      #Callback function triggered when the "View
Tracks" button is clicked
    status_lbl.configure(text="View Tracks button was clicked!", fg="green")
#Updates the status label
    TrackViewer(tk.Toplevel(window))    #Create a new child window for
viewing tracks

def update_rating_clicked():    #Callback function for the "Update Tracks"
button
    status_lbl.configure(text="Update Rating button was clicked!",
fg="green")
    UpdateRating(tk.Toplevel(window))   #Open update rating window

def create_track_clicked():     #Callback for the "Create Track List" button
    status_lbl.configure(text="Create track button was clicked!", fg="green")
    CreateTrackList(tk.Toplevel(window))    #Open create track list window

window = tk.Tk()    #Create the main window
window.geometry("400x150")      #Set window size
window.title("JukeBox")     #Set the title of the window
window.configure(bg="#f4f4f4")      #Set background color
window.resizable(False, False)      #Disable resizing

# fonts.configure()     #Apply global font configuration if available

header_lbl = tk.Label(window, text="Select an option by clicking one of the
buttons below", font=("Helvetica", 12), bg="#f4f4f4")   #Create and locate a
label
header_lbl.grid(row=0, column=0, columnspan=3, padx=10, pady=(15,10))

view_tracks_btn = tk.Button(window, text="View Tracks",
command=view_tracks_clicked, bg="#008080", fg="white")      #Create and
locate "View Tracks" button (binds to view_tracks_clicked)
view_tracks_btn.grid(row=1, column=0, padx=10, pady=10)

create_track_list_btn = tk.Button(window, text="Create Track List",
command=create_track_clicked, bg="#008080", fg="white")     #Create and
locate "Create Track List" button (binds to create_track_clicked)
create_track_list_btn.grid(row=1, column=1, padx=10, pady=10)

update_tracks_btn = tk.Button(window, text="Update Tracks",
command=update_rating_clicked, bg="#008080", fg="white")    #Create and
locate "Update Tracks" button (binds to update_rating_clicked)
update_tracks_btn.grid(row=1, column=2, padx=10, pady=10)

status_lbl = tk.Label(window, text="Status", font=("Helvetica", 10),
bg="#f4f4f4")  #Create a label to display the status
status_lbl.grid(row=2, column=0, columnspan=3, padx=10, pady=10)

window.mainloop()    #Keeps the window open and responsive
```

- Evidence 31: Source code of playlist.json

```
[
    "01",
    "02",
```

```
    "03"
]
```

- Evidence 32: Source code of test library item

```
from library_item import LibraryItem

def test_info():
    item = LibraryItem("Ex's Hate Me", "B Ray", 4)
    expected = "Ex's Hate Me - B Ray ****"
    assert item.info() == expected

def test_starts():
    item = LibraryItem("Stayin' Alive", "Bee Gees", 5)
    assert item.stars() == "*****"

def test_default_constructor():
    item = LibraryItem("Stayin' Alive", "Bee Gees")
    assert item.play_count == 0
    assert item.rating == 0
```

- Evidence 33: Source code of test track library

```
from track_library import get_play_count, get_artist, get_rating, get_name,
set_rating, increment_play_count

def test_get_name():
    assert get_name("01") == "Another Brick in the Wall"
def test_get_name_invalid():
    assert get_name("12") is None

def test_get_artist():
    assert get_artist("01") == "Pink Floyd"
def test_get_artist_invalid():
    assert get_artist("12") is None

def test_get_rating():
    assert get_rating("03") == 2
def test_get_rating_invalid():
    assert get_rating("12") == -1

def test_set_rating_valid():
    set_rating("04", 5)
    assert get_rating("04") == 5
def test_set_rating_invalid():
    assert set_rating("12", 4) is None

def test_increment_play_count():
    increment_play_count("01")
    assert get_play_count("01") == 1
def test_get_play_count_invalid():
    assert get_play_count("12") == -1
```

Declaration of AI Use

Please append this page to the end of your assignment when you have used AI during the process of undertaking the assignment to acknowledge the ways in which you have used it.

I have used AI while undertaking my assignment in the following ways:

- To develop research questions on the topic – YES/NO
- To create an outline of the topic – YES/NO
- To explain concepts – YES/NO
- To support my use of language – YES/NO
- To summarise the following articles/resources:

1. Yes
2. Yes
3. No
4. No
5. Yes

I have used AI while undertaking my assignment in the following ways: To develop research questions on the topic, to create an outline of the topic, and to summarize the conclusion and the following articles/resources.