ICS3U Cumulative Project
GarvinTunes MMS

Overview

For your cumulative project, you will create a Music-Management System (MMS). Your program will build a library from the metadata in "music" files. It will be able to display various information about individual tracks (songs), albums or artists, and it should be able to "play" music files. While much of your program will be a simulation, it incorporates many of the same programming concepts as in a real-life MMS.

Tracks

Naturally, each track is saved in GAR file format. GAR files always end with the extension ".gar", such as "HeyJude.gar". All GAR files must be saved in the same directory (folder) as your program, so that they are properly detected.

Each GAR file is made up of 7 lines. The first 6 lines contain *metadata*: the artist's name, album name, release year, genre, track name, and track number, in that order. The seventh line contains the *song data* itself. Due to significant advances in file-compression technology (patents pending), tracks saved in GAR file format use a mere 10 characters per second of music. For example, 5 seconds of audio might be encoded in just 50 characters as:

♫♪♪♫♫♬♩ ♩ ♪♪♫♫♪♪♫♫♪♬♩ ♫♩ ♬ ♫♫♫♫♩ ♪♪♫♫♩ ♩ ♬ ♪♪♪♪♩ ♩ ♩ ♫♪♪♫♫♫♩ ♩ ♩ ♫♫♬ ♬ ♪♫♫♪♬

Several GAR files will be provided to test your program.

File-Handling Library

Since reading and writing files is not covered in this course (it is not difficult, but is outside of the scope of the Grade 11 Curriculum), I have provided you with a library, **gtunes.py**, that handles all of the file-handling requirements. In development terms, this is known as an *Application Programming Interface* (API). To use these functions, import the library at the start of your program. In it are the following functions:

- load_library(): scans the current directory, attempts to load all valid tracks into the music library.
- is_valid_data(data): determines whether the metadata and song data are in the proper format or not.
- load_track(file_name): loads the metadata for a single valid track into the music library.
- play_track(file_name): "plays" a previously-validated and loaded track.

Additional functions are also provided for Extensions (see page 3). You should leave these alone until your program can perform basic tasks, like loading and playing individual tracks.

- save_metadata(file_name, metadata): saves new metadata for a given track to its file.
- is_valid_playlist(playlist): determines if a playlist is in the correct file format or not.
- load_playlist(file_name): loads the content of a playlist into the program.
- save_playlist(file_name, playlist): saves a playlist to a file.

Being able to use a third-party API is an important part of programming larger projects. You should spend some time reading the comments in the library to understand how to use each function in the context of your program. **DO NOT** add or change any code in the file – doing so may break your program, and be difficult to fix! If you discover a serious error in the library, let me know and I will update it. While the library has been extensively tested, it is possible that something has been overlooked or omitted.

Stage 1: Testing Functionality

To become familiar with the file-handling library, you should create a program that uses the test tracks available in **test.zip**. This archive contains four GAR files. The files named testN.gar contain valid data, while the file named badfile.gar has an invalid release year. Extract these files into the same folder as the gtunes.py module. You program should use the techniques covered in class (sequential processing, decision-making, code repetition, string and list processing, and creating functions) to do the following:

- Create a music library by loading the metadata of all test tracks in the current directory. Verify that only three of the four tracks are loaded.
- Play the three loaded tracks.
- Display the metadata (artist, album, etc.) for each of the three tracks.

Stage 2: Core Program

Once you have basic functions up-and-running, modify your program so that it adds a main menu from which the user can select various options. These should include the following:

- displaying a list of general statistics for the library (in a nicely formatted form), including the number of albums, the number of *unique* artists (some artists may have multiple albums), the number of *unique* genres (some genres may apply to multiple albums) and the total number of tracks in the library
- displaying a list of all albums, given an album name, along with their associated artists (e.g. *Abbey Road* – The Beatles, *A Night At the Opera* – Queen, etc.)
- displaying a list of all artists in the library, along with their albums (e.g. The Beatles: *The White Album*, *Abbey Road*, *Let It Be*, ...)
- displaying a list of all genres in the library and the number of albums in each genre (e.g. pop (13), rock (21), country (2), ...)
- displaying information about a specific artist, including a list of all genres, albums, and total number of tracks in the library
- displaying information about a specific album, including album name, artist, release year, genre, number of tracks on the album, and a list of all tracks on the album
- displaying information about a specific track, such as album name, artist, release year, and genre
- quitting the program
- Locate a specific album, artist or track by searching for some sequence of characters. Note that searching for an album or a track may result in more than one match, so the user will need to select from the available options.
- “Play” a specific track, given an album name and artist
- “Play” an entire album, given an album name and artist

Delete the test tracks from the folder, and use the tracks in the two archives **rabbit_hole.zip** and **trust_issues.zip** to test your program. These archives contain GAR files for two albums (*Down the Rabbit Hole* by Jack Rabbit and *Trust Issues* by Suspicious Minds). All album names and track names should be unique, so you do not need to worry about duplicates yet.

All user input should be handled gracefully. You have learned how to parse user input to determine whether it can be typecast as an integer, or how to ensure that input is a fixed number of characters. Your program should not crash due to improper user input. Instead, it should display a message and redirect the user to enter correct input instead.

You should break up your main program into smaller functions, each of which does a specific task, rather than create a monolithic one. This will make your program easier to manage, and more importantly, easier to test and debug.

In addition to the program itself, your program must include a program header with your name, student number, date, and description of the program. In-line comments should be liberally used throughout the program to explain blocks of code. Each function you write must also have a brief function header, describing its purpose, its arguments, and any returned values.

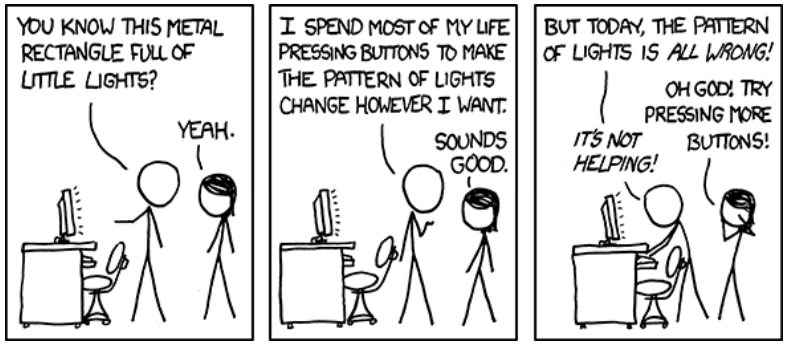
Stage 3: Program Extensions

Once you have a functional program, you may wish to try to implement some additional features. You are strongly encouraged to add as many extra features to your program as you can, provided they do not interfere with or degrade the performance of your basic program. The more features you successfully integrate into your program, the better. This is your chance to show off how much you have picked up during this course. Possible extensions include:

- When displaying album info, display all tracks from a particular album in ascending order (1, 2, 3, ...).
- Provide a way for the user to add additional tracks to the library.
- Handle cases involving albums or tracks with the same names. Use the additional archives of GAR files to help test your program. There are some duplicate track names in there.
- Provide a mechanism where the user can update the metadata of a particular track, or for all tracks on a specified album. For example, the user may change the genre of a track from Rock to Pop, and update the file accordingly.
- Allow the user to create, load, save and play playlists. Tracks may come from the same album, or from many.
- Add information about time, such as:
  - Display the length of each track in MM:SS format when playing or displaying info about a track.
  - Determine the length of the longest and shortest tracks on the library (include in library stats).
  - Determine the range of years over which the entire library spans (e.g. 1985 to 2010 = 25 years).
  - Determine the length of a playlist (if you have already implemented this feature).
- Use a module like rich to display information using colours, or in tabular form.
- Suggest something relevant to me. Be creative!

Tips for Success

- **Start right now**. You cannot write a program of this size the night before it is due. Your program *will* contain errors – give yourself plenty of time to fix them.
- Pre-plan your program. Your code will come together faster if you have clear goals in mind.
- Start with the basic elements you need, like loading a few songs and playing them. Test everything before attempting to implement any more advanced features.
- Write your functions early. Not only will they help organize your code, but they may also make it easier to reuse code and reduce the amount of work you need to do in the end.
- Add one function/rule at a time. If your program misbehaves when you introduce a new function or rule, you can be relatively certain that any errors are resulting from that particular section of code.
- Document your program as you code it, not at the end. Comments should explain *what* sections of code do, or *how* they do these things. Use comments to remind yourself what you're doing, and why.
- Make backups of your program. If you make some changes that completely mess up your program, you can always revert to an earlier version instead of spending hours undoing your work.
- Test extensively. Try all edge cases, and verify that bad input is successfully handled. Hard-code values to force specific situations to occur. Get a friend to test your program: if s/he can break it, it needs fixing.
- Choose simplicity over cleverness. Get things working, *then* try to optimize.
- Keep a positive attitude, and have some fun! Everything in this project has been covered during the course. While some pieces may be harder to implement than others, you have all of the necessary skills to write this program.



Comic courtesy of XKCD.

Project Rubric

Category	Level R *	Level 1	Level 2	Level 3	Level 4
Knowledge	Program shows no knowledge of structure and coding conventions.	Program shows a limited knowledge of structure and coding conventions.	Program shows some understanding of structure and coding conventions.	Program shows a considerable understanding of structure and coding conventions.	Program is free from syntax and run-time errors, and uses functions for all major tasks, including any extensions.
	Program contains many major syntax or run-time errors.	Program contains many major syntax or run-time errors.	Program contains some minor syntax or run-time errors.	Program is free from syntax errors, contain few run-time errors.	
Application	Choice of data types, functions, and commands is not task-appropriate.	Few data types, functions and commands are task-appropriate.	Some data types, functions and commands are task-appropriate.	Most/all data types, functions and commands are task-appropriate.	Some program extensions have been incorporated, appropriate data types and commands have been used to implement these extensions.
	No, or very few, tasks are solved using appropriate tools, functions and commands.	Few tasks are solved using appropriate tools, functions and commands.	Some tasks are solved using appropriate tools, functions and commands.	Most/all tasks are solved using appropriate tools, functions and commands.	
	Program uses highly inefficient methods throughout.	Program uses few efficient methods to solve tasks.	Program uses some efficient methods to solve tasks.	Program uses many efficient methods to solve tasks.	
Thinking	Program has no logical organization.	Program has little logical organization.	Program has some logical organization.	Program has good logical organization.	Program is logically organized, uses efficient methods for all tasks, and produces correct output for all tasks, including any extensions.
	Program fails, or gives incorrect results, all of the time when invalid input is given.	Program fails, or gives incorrect results, in most cases when invalid input is given.	Program fails, or gives incorrect results, in some cases when invalid input is given.	Program handles most/all invalid input gracefully.	
	Program/functions produce incorrect results for all or most test cases.	Program/functions produce correct results for few test cases.	Program/functions produce correct results for some test cases.	Program/functions produce correct results for most/all test cases.	
Communication	No program or function headers provided.	Minimal program header, no function headers provided.	Minimal program and function headers provided.	Meaningful program and function headers provided.	Program contains meaningful program and function headers, all variable and function names are appropriate, sufficient inline documentation is provided.
	Variable/function names are poorly chosen.	Some variable/ function names are meaningful.	Many variable/ function names are meaningful.	Most/all variable/ function names are meaningful.	
	No inline comments provided, or most descriptions of tasks are unclear.	Some inline comments are provided, some descriptions of tasks are clear.	Some inline comments are provided, many descriptions of tasks are clear.	Many inline comments are provided, most/all descriptions of tasks are clear.	

* Level R indicates an insufficient achievement of curriculum expectations for a given descriptor.