System Requirements Specification Index

For

Music Playlist Management System

Version 1.0

IIHT Pvt. Ltd.

fullstack@iiht.com

TABLE OF CONTENTS

Music Playlist Management System

System Requirements Specification

1 PROJECT ABSTRACT

Melodia Music Streaming requires a playlist management system to organize their expanding music collection. The system will catalog songs, track playlist details, and generate reports using Python's tuple data structures and related operations. This console application demonstrates tuple immutability principles and effective use cases while allowing users to efficiently manage music collections. The system performs critical functions including organizing songs by attributes, creating immutable playlist records, manipulating music data through tuple operations, and generating listening statistics. By implementing these operations with Python tuples, the system provides an efficient way for users to organize and analyze their music collection while ensuring data integrity.

2 BUSINESS REQUIREMENTS:

Screen Name	Console input screen
Problem Statement	1. System needs to store and categorize different song attributes (artist, title, genre, duration, release year, album) 2. System must support filtering songs by genre, artist, duration range, or release decade 3. Console should handle different tuple operations like Creating immutable song records, Tuple packing and unpacking, Tuple as dictionary keys, Named tuples for improved readability, Tuple comparison and sorting

3 CONSTRAINTS

3.1 INPUT REQUIREMENTS

1. Song Records):
   - Must be stored as tuples with fields for id, title, artist, genre, duration, release_year, album
   - Must be stored in list variable `songs`
   - Example: `("S001", "Bohemian Rhapsody", "Queen", "rock", 354, 1975, "A Night at the Opera")`
2. Song Genres:
   - Must be one of: "rock", "pop", "jazz", "classical", "electronic", "hip-hop"
   - Must be stored as string in song's genre field (index 3)
   - Example: "rock"
3. Duration:
   - Must be stored as integer in seconds at index 4
   - Example: 354 (for 5:54)
4. Release Year:
   - Must be stored as integer at index 5
   - Example: 1975
5. Predefined Songs:
   - Must use these exact predefined songs in the initial song list:
   - `("S001", "Bohemian Rhapsody", "Queen", "rock", 354, 1975, "A Night at the Opera")`
   - `("S002", "Billie Jean", "Michael Jackson", "pop", 294, 1982, "Thriller")`
   - `("S003", "Take Five", "Dave Brubeck", "jazz", 324, 1959, "Time Out")`
   - `("S004", "Moonlight Sonata", "Ludwig van Beethoven", "classical", 363, 1801, "Piano Sonatas")`
   - `("S005", "Strobe", "Deadmau5", "electronic", 601, 2009, "For Lack of a Better Name")`
6. New Releases:

- Must use these exact predefined items in the new releases list:
- `("N001", "Blinding Lights", "The Weeknd", "pop", 200, 2020, "After Hours")`
- `("N002", "Bamboo", "J Balvin", "hip-hop", 190, 2023, "Colores")`

3.2 Operations Constraints

1. Song Record Creation:
   - Must use tuple constructor or literal syntax
   - Example: `song = ("S001", "Bohemian Rhapsody", "Queen", "rock", 354, 1975, "A Night at the Opera")`
   - Must use variable name `song` for individual songs
2. Song Filtering by Genre:
   - Must use tuple unpacking in list comprehension or filter
   - Example: `[song for song in songs if song[3] == "rock"]`
   - Must use variable name `filtered_songs` for the result
3. Duration Formatting:
   - Must use tuple indexing to access duration
   - Example: `f"{song[4] // 60}:{song[4] % 60:02d}"`
   - Must use variable name `formatted_duration` for the result
4. Tuple Packing and unpacking:
   - Must demonstrate both packing and unpacking operations
   - Example: `song_id, title, artist, *rest = song`
   - Must use variable names `song_id`, `title`, `artist`, and `rest`
5. Creating named Tuples:
   - Must use `collections.namedtuple` to create a Song type
   - Example: `Song = namedtuple('Song', ['id', 'title', 'artist', 'genre', 'duration', 'release_year', 'album'])`

- Must use variable name `Song` for the named tuple type
- Must use variable name `named_songs` for a list of named tuple instances

6. Tuple as Dictionary Keys:
    - Must use tuples as dictionary keys for counting or grouping
    - Example: `plays = {(song[2], song[1]): 0 for song in songs}`
    - Must use variable name `plays_dict` for the result
7. Multiple Playlist Creation:
    - Must create immutable playlist records that include name, created_date, and song_ids
    - Example: `playlist = ("Rock Classics", "2023-03-04", tuple(song[0] for song in rock_songs))`
    - Must use variable name `playlist` for individual playlists
    - Must use variable name `playlists` for a list of playlists
8. Song Sorting:
    - Must use tuple comparison for sorting songs
    - Example: `sorted(songs, key=lambda x: (x[2], x[5]))` (by artist then year)
    - Must use variable name `sorted_songs` for the result
9. Decade Filtering:
    - Must use tuple data for filtering by decade
    - Example: `[song for song in songs if song[5] // 10 == 197]` (for 1970s)
    - Must use variable name `decade_songs` for the result
10. Genre Distribution Calculation:
    - Must use tuples in dict creation to count genre distribution
    - Example: `{genre: len([s for s in songs if s[3] == genre]) for genre in genres}`

- Must use variable name `genre_counts` for the result

3.3 Output Constraints

1. Display Format:
   - Show song ID, title, artist, genre, duration, release year, album
   - Format duration as MM:SS
   - Each song must be displayed on a new line

2. Output:
   - Show "===== MUSIC PLAYLIST MANAGEMENT SYSTEM ====="
   - Show "Total Songs: {count}"
   - Show "Total Duration: {hours}h {minutes}m {seconds}s"
   - Show "Current Song Collection:"
   - Show songs with format: "{id} | {title} | {artist} | {genre} | {duration} | {year} | {album}"
   - Show "Playlist Information:" when displaying playlist data
   - Show "Song Distribution:" when showing genre counts

4. Template code Structure:

1. Data Management Functions:
   - `initialize_data()` - creates the initial song and new releases lists

2. Tuple Operation Functions:
   - `create_song_record(id, title, artist, genre, duration, release_year, album)` creates immutable song record

- `filter_by_genre(songs, genre)` - filters songs by genre using tuple data
- `filter_by_artist(songs, artist)` - filters songs by artist
- `filter_by_duration(songs, min_duration, max_duration)` - filters songs by duration range
- `filter_by_decade(songs, decade)` - filters songs by release decade
- `format_duration(seconds)` - formats duration from seconds to MM:SS
- `create_named_tuple_songs(songs)` - converts regular tuples to named tuples
- `create_playlist(name, song_ids, songs)` - creates immutable playlist record
- `sort_songs(songs, sort_key)` - sorts songs by specified attribute
- `calculate_genre_distribution(songs)` - calculates song count by genre
- `integrate_new_releases(songs, new_releases)` - combines song lists

3. Display Functions:
   - `get_formatted_song(song)` - formats a song for display
   - `display_data(data, data_type)` - displays songs or other data types

4. Program Control Functions:
   - `main()` - main program function

5. DETAILED FUNCTION IMPLEMENTATION GUIDE

5.1 Data Initialization Functions

1. Write a Python function to initialize the music data with predefined songs using tuples.

Define: `initialize_data()`
The function should:

- Create a list named `songs` containing exactly 5 predefined song tuples
- Each song tuple must have 7 elements: (id, title, artist, genre, duration, release_year, album)
- Use these exact songs:
    - `("S001", "Bohemian Rhapsody", "Queen", "rock", 354, 1975, "A Night at the Opera")`
    - `("S002", "Billie Jean", "Michael Jackson", "pop", 294, 1982, "Thriller")`
    - `("S003", "Take Five", "Dave Brubeck", "jazz", 324, 1959, "Time Out")`
    - `("S004", "Moonlight Sonata", "Ludwig van Beethoven", "classical", 363, 1801, "Piano Sonatas")`
    - `("S005", "Strobe", "Deadmau5", "electronic", 601, 2009, "For Lack of a Better Name")`
- Create a list named `new_releases` containing exactly 2 predefined release tuples:
    - `("N001", "Blinding Lights", "The Weeknd", "pop", 200, 2020, "After Hours")`
    - `("N002", "Bamboo", "J Balvin", "hip-hop", 190, 2023, "Colores")`
- Create a tuple named `genres` containing valid music genres: `("rock", "pop", "jazz", "classical", "electronic", "hip-hop")`
- Return a tuple containing `(songs, new_releases, genres)`
- All data must be immutable using tuple structures

2. Write a Python function to create an immutable song record as a tuple.

Define: `create_song_record(id, title, artist, genre, duration, release_year, album)`
The function should:

- Validate that `id` is a non-empty string, raise ValueError if invalid
- Validate that `title` is a non-empty string, raise ValueError if invalid
- Validate that `artist` is a non-empty string, raise ValueError if invalid
- Validate that `genre` is a non-empty string, raise ValueError if invalid
- Validate that `duration` is a positive integer (> 0), raise ValueError if invalid
- Validate that `release_year` is an integer, raise ValueError if invalid
- Validate that `album` is a non-empty string, raise ValueError if invalid
- Create and return a tuple with all song information: `(id, title, artist, genre, duration, release_year, album)`
- Use variable name `song` for the created tuple

- Ensure the returned tuple is immutable and contains exactly 7 elements
- Example return: `("S001", "Test Song", "Test Artist", "rock", 180, 2000, "Test Album")`

5.2 Filtering Functions

3. Write a Python function to filter songs by genre using tuple data.

Define: `filter_by_genre(songs, genre)`
The function should:

- Validate that `genre` is a non-empty string, raise ValueError if invalid
- Use list comprehension to filter songs where the genre field (index 3) matches the specified genre
- Access genre using tuple indexing: `song[3]`
- Return a list of song tuples that match the specified genre
- Use variable name `filtered_songs` for intermediate results
- Handle empty input lists by returning an empty list
- Preserve the original song tuple structure in results
- Example: `[song for song in songs if song[3] == genre]`

4. Write a Python function to filter songs by artist using tuple data.

Define: `filter_by_artist(songs, artist)`
The function should:

- Validate that `artist` is a non-empty string, raise ValueError if invalid
- Use list comprehension to filter songs where the artist field (index 2) matches the specified artist
- Access artist using tuple indexing: `song[2]`
- Return a list of song tuples by the specified artist
- Use variable name `filtered_songs` for results
- Perform case-sensitive matching
- Handle empty input lists by returning an empty list
- Preserve the original song tuple structure in results
- Example: `[song for song in songs if song[2] == artist]`

5. Write a Python function to filter songs by duration range using tuple data.

Define: `filter_by_duration(songs, min_duration, max_duration)`
The function should:

- Validate that `min_duration` is a non-negative integer, raise ValueError if invalid
- Validate that `max_duration` is a positive integer, raise ValueError if invalid
- Validate that `min_duration <= max_duration`, raise ValueError if invalid
- Use list comprehension to filter songs where duration (index 4) is within the specified range
- Access duration using tuple indexing: `song[4]`
- Include songs where `min_duration <= song[4] <= max_duration` (inclusive range)
- Return a list of song tuples within the duration range
- Use variable name `filtered_songs` for results
- Handle empty input lists by returning an empty list
- Example: `[song for song in songs if min_duration <= song[4] <= max_duration]`

6. Write a Python function to filter songs by release decade using tuple data.

Define: `filter_by_decade(songs, decade)`
The function should:

- Validate that `decade` is an integer, raise ValueError if invalid
- Calculate decade range: decade to decade + 9 (inclusive)
- Use list comprehension to filter songs where release year (index 5) is within the decade
- Access release year using tuple indexing: `song[5]`
- Include songs where `decade <= song[5] <= decade + 9`
- Return a list of song tuples from the specified decade
- Use variable name `decade_songs` for results
- Handle empty input lists by returning an empty list
- Example: for decade=1970, include songs from 1970-1979

5.3 Data Processing Functions

7. Write a Python function to format duration from seconds to MM:SS.

Define: `format_duration(seconds)`
The function should:

- Validate that `seconds` is a non-negative integer, raise ValueError if invalid
- Calculate minutes using integer division: `minutes = seconds // 60`
- Calculate remaining seconds using modulo: `remaining_seconds = seconds % 60`

- Format as string with zero-padding for seconds: `f"{minutes}:{remaining_seconds:02d}"`
- Use variable name `formatted_duration` for the result
- Return formatted string in MM:SS format
- Handle edge cases like 0 seconds (return "0:00")
- Example: 354 seconds becomes "5:54"

8. Write a Python function to convert regular tuple songs to named tuples.

Define: `create_named_tuple_songs(songs)`
The function should:

- Validate that `songs` is not empty, raise ValueError if empty list
- Import `namedtuple` from collections module
- Create a named tuple type: `Song = namedtuple('Song', ['id', 'title', 'artist', 'genre', 'duration', 'release_year', 'album'])`
- Use variable name `Song` for the named tuple type
- Convert each song tuple to a named tuple using: `Song(*song)`
- Use list comprehension: `[Song(*song) for song in songs]`
- Use variable name `named_songs` for the result list
- Return list of named tuple instances
- Ensure field access works: `named_song.title`, `named_song.artist`, etc.
- Maintain immutability of the resulting named tuples

9. Write a Python function to create an immutable playlist record.

Define: `create_playlist(name, song_ids, songs)`
The function should:

- Validate that `name` is a non-empty string, raise ValueError if invalid
- Validate that `song_ids` is not empty, raise ValueError if empty list
- Validate that `songs` is not empty, raise ValueError if empty list
- Extract all available song IDs from songs list: `[song[0] for song in songs]`
- Validate that all song_ids exist in the songs list, raise ValueError for invalid IDs
- Get current date using `datetime.now().strftime("%Y-%m-%d")`
- Create immutable playlist tuple: `(name, current_date, tuple(song_ids))`
- Use variable name `playlist` for the result
- Return tuple with exactly 3 elements: name, date, and song IDs tuple
- Ensure song_ids are stored as an immutable tuple within the playlist tuple
- Example return: `("Rock Playlist", "2023-12-10", ("S001", "S002"))`

10. Write a Python function to sort songs by specified attribute using tuple comparison.

Define: `sort_songs(songs, sort_key)`
The function should:

- Validate that `songs` is not empty, raise ValueError if empty list
- Validate that `sort_key` is a non-empty string, raise ValueError if invalid
- Define valid sort keys: "title" (index 1), "artist" (index 2), "year" (index 5), "duration" (index 4), "genre" (index 3), "artist_year" (combined)
- Validate that `sort_key` is in the valid keys list, raise ValueError if invalid
- Create lambda functions for sorting:
  - "title": `lambda x: x[1]`
  - "artist": `lambda x: x[2]`
  - "year": `lambda x: x[5]`
  - "duration": `lambda x: x[4]`
  - "genre": `lambda x: x[3]`
  - "artist_year": `lambda x: (x[2], x[5])`
- Use `sorted()` function with appropriate key function
- Use variable name `sorted_songs` for the result
- Return sorted list of song tuples
- Maintain original tuple structure in results

5.4 Statistical Analysis Functions

11. Write a Python function to calculate the distribution of songs by genre.

Define: `calculate_genre_distribution(songs, genres)`
The function should:

- Validate that `songs` is not empty, raise ValueError if empty list
- Validate that `genres` is not empty, raise ValueError if empty tuple
- Use dictionary comprehension to count songs per genre
- For each genre in genres tuple, count songs where `song[3] == genre`
- Create dictionary: `{genre: len([s for s in songs if s[3] == genre]) for genre in genres}`
- Use variable name `genre_counts` for the result
- Return dictionary with genre names as keys and counts as values
- Include all genres from the genres tuple, even if count is 0
- Access genre field using tuple indexing: `song[3]`

- Example return: `{"rock": 2, "pop": 1, "jazz": 1, "classical": 0, "electronic": 1, "hip-hop": 0}`

12. Write a Python function to calculate the total duration of all songs.

Define: `calculate_total_duration(songs)`
The function should:

- Validate that `songs` is not empty, raise ValueError if empty list
- Sum all durations using: `sum(song[4] for song in songs)`
- Access duration field using tuple indexing: `song[4]`
- Convert total seconds to hours, minutes, and seconds:
  - `hours = total_seconds // 3600`
  - `minutes = (total_seconds % 3600) // 60`
  - `seconds = total_seconds % 60`
- Return tuple containing `(hours, minutes, seconds)`
- Use variable name `total_seconds` for the sum
- Handle edge case of zero total duration
- Example: 7200 seconds returns `(2, 0, 0)` for 2 hours

13. Write a Python function to integrate new releases into the main song list.

Define: `integrate_new_releases(songs, new_releases)`
The function should:

- Validate that `songs` is a list, raise ValueError if not a list
- Validate that `new_releases` is a list, raise ValueError if not a list
- Combine lists using concatenation: `songs + new_releases`
- Use variable name `combined_songs` for the result
- Return the combined list of song tuples
- Preserve the original order: existing songs first, then new releases
- Do not modify the original input lists (create new list)
- Maintain tuple structure for all songs in the result
- Handle empty lists gracefully (return the non-empty list)

5.5 Display and Formatting Functions

14. Write a Python function to format a song tuple for display.

Define: `get_formatted_song(song)`
The function should:

- Validate that `song` is a tuple with at least 7 elements, raise ValueError if invalid
- Unpack the song tuple: `song_id, title, artist, genre, duration, year, album = song`
- Format duration using the `format_duration()` function
- Use variable name `formatted_duration` for the formatted time
- Return formatted string: `f"{song_id} | {title} | {artist} | {genre} | {formatted_duration} | {year} | {album}"`
- Use " | " (space-pipe-space) as separator between fields
- Access all tuple elements by index or unpacking
- Handle duration conversion from seconds to MM:SS format
- Example return: `"S001 | Bohemian Rhapsody | Queen | rock | 5:54 | 1975 | A Night at the Opera"`

15. Write a Python function to format a playlist tuple for display.

Define: `get_playlist_info(playlist, songs)`
The function should:

- Validate that `playlist` is a tuple with at least 3 elements, raise ValueError if invalid
- Validate that `songs` is not empty, raise ValueError if empty list
- Unpack playlist tuple: `name, created_date, song_ids = playlist`
- Create song lookup dictionary: `{song[0]: song for song in songs}`
- Calculate total duration by summing durations of playlist songs
- Count total songs in playlist: `len(song_ids)`
- Format playlist information with these lines:
  - `f"Name: {name}"`
  - `f"Created: {created_date}"`
  - `f"Songs: {len(song_ids)}"`
  - `f"Duration: {format_duration(total_duration)}"`
  - `"Tracks:"`
- Add numbered track list with format: `f"  {i}. {song[1]} - {song[2]} ({format_duration(song[4])})"`
- Join all lines with newline characters
- Use variable name `playlist_info` for the result list
- Return complete formatted string

16. Write a Python function to display formatted song data or statistics.

Define: `display_data(data, data_type="songs")`
The function should:

- Validate that `data` is not None, raise ValueError if None
- Handle different data_type values:
    - "songs" or "results": Display list of songs using `get_formatted_song()`
    - "playlist": Display playlist information (data should be pre-formatted string)
    - "distribution": Display genre distribution dictionary
    - "named_songs": Display named tuple songs with attribute access
- For "songs"/"results": Print header and iterate through songs
- For "distribution": Print "Song Distribution:" header and show `f"{genre}: {count} songs"`
- For "named_songs": Print "Named Tuple Songs:" and format using named tuple attributes
- For unknown data_type: Print `f"Unknown data type: {data_type}"` and print data as-is
- Handle empty data lists by printing "No songs to display."
- Use appropriate headers for each section
- Print each song on a separate line

5.6 Main Program Function

17. Write a Python function to demonstrate the music playlist management system.

Define: `main()`
The function should:

- Call `initialize_data()` to get songs, new_releases, and genres
- Initialize empty list `playlists = []` to store created playlists
- Display system header: `"===== MUSIC PLAYLIST MANAGEMENT SYSTEM ====="`
- Calculate and display statistics:
    - Total songs count: `f"Total Songs: {len(songs)}"`
    - Total duration using `calculate_total_duration()`: `f"Total Duration: {hours}h {minutes}m {seconds}s"`
- Display menu options numbered 0-6:
    - "1. View Songs"
    - "2. Filter Songs"
    - "3. Create Playlist"
    - "4. Convert to Named Tuples"
    - "5. Calculate Statistics"
    - "6. Integrate New Releases"

    - "0. Exit"
- Implement menu choice handling using if/elif statements
- For each menu option, demonstrate the corresponding functions
- Handle user input validation and error catching using try/except blocks
- Use appropriate variable names throughout: `choice`, `filtered_songs`, `playlist`, `named_songs`, etc.
- Show meaningful output that demonstrates each function's capabilities
- Loop until user selects exit option (0)
- Print "Thank you for using the Music Playlist Management System!" on exit

6. EXECUTION STEPS TO FOLLOW:

1. Run the program

2. View the main menu

3. Select operations:

   - Option 1: View Songs

   - Option 2: Filter Songs

   - Option 3: Create Playlist

   - Option 4: Convert to Named Tuples

   - Option 5: Calculate Statistics

   - Option 6: Integrate New Releases

   - Option 0: Exit

4. Perform operations on the song list

5. View results after each operation

6. Exit program when finished

<u>Execution Steps to Follow:</u>

- All actions like build, compile, running application, running test cases will be through Command Terminal.

- To open the command terminal the test takers, need to go to Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal

- This editor Auto Saves the code

- If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page)

- These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.

- To launch application: python3 filename.py

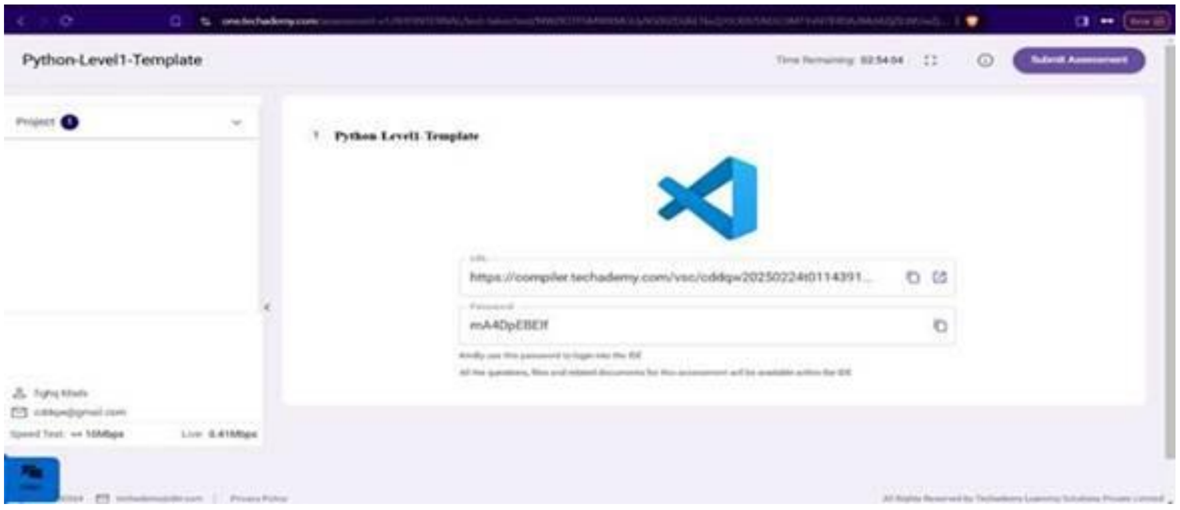
- To run Test cases: python3 -m unittest

<u>Screen shot to run the program</u>

To run the application

python3 filename.py

To run the testcase python3 -m unittest



Python-Level1-Template

Python Level1 Template

https://compiler.techademy.com/vsc/cddqw20250224t0114391...

mA4DpEBEIf

- Once you are done with development and ready with submission, you may navigate to the previous tab and submit the workspace. It is mandatory to click on “Submit Assessment” after you are done with code.