# Music Playlist Manager

Building a Python Application with Operator Overloading and File I/O

## Project Overview

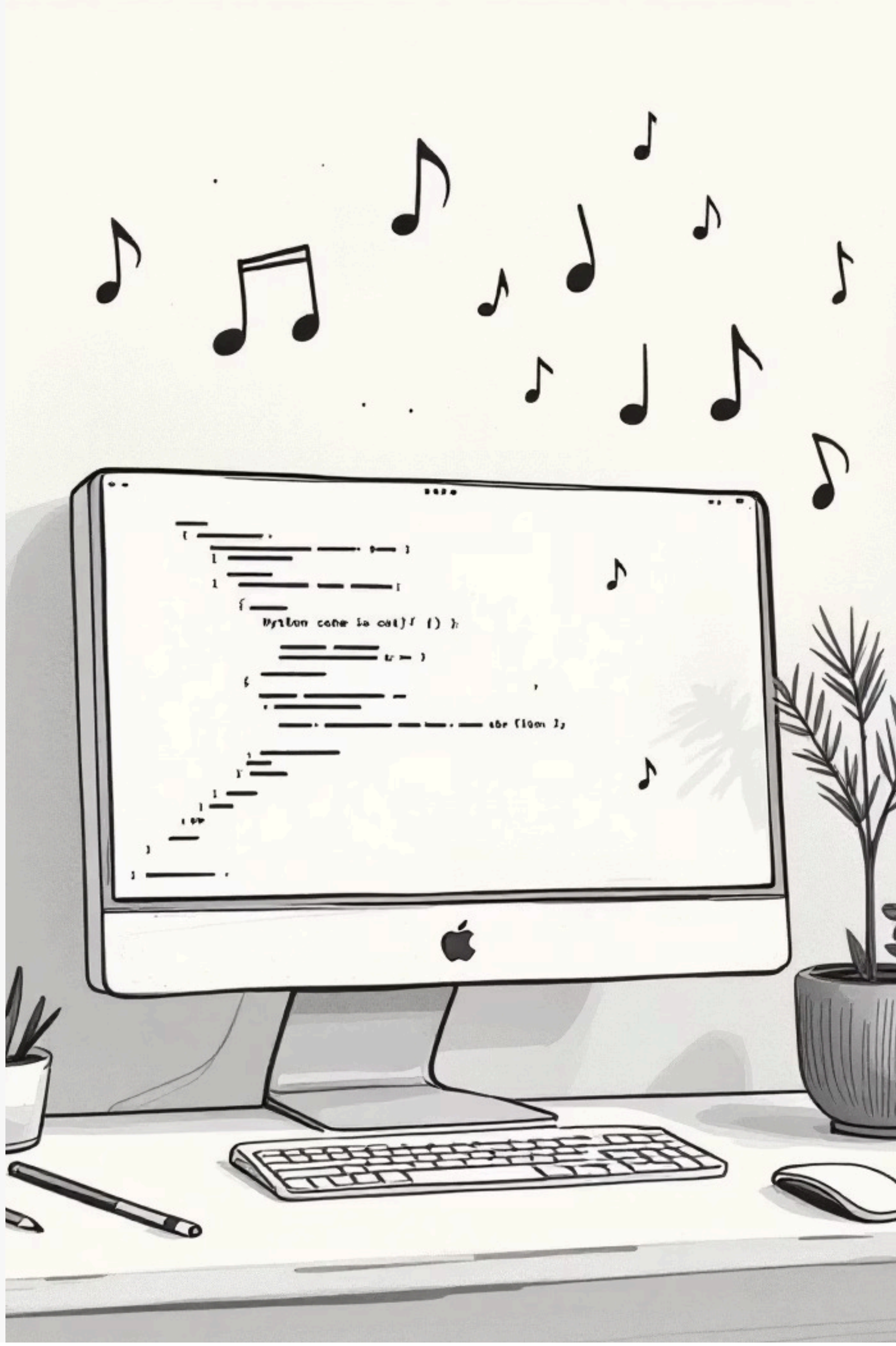
### Core Components

- Custom Playlist class with track management
- Operator overloading for intuitive operations
- File I/O for persistent storage
- Error handling mechanisms

### Key Requirements

- Add and remove tracks using operators
- Save playlists to JSON/CSV formats
- Load existing playlists from files
- Handle exceptions gracefully

This project demonstrates how Python's special methods enable elegant, user-friendly code whilst maintaining robust functionality for real-world music management.



# Setting up the Playlist Class

The foundation of our music manager begins with a well-structured Playlist class that stores tracks and metadata efficiently.

### Class Initialisation

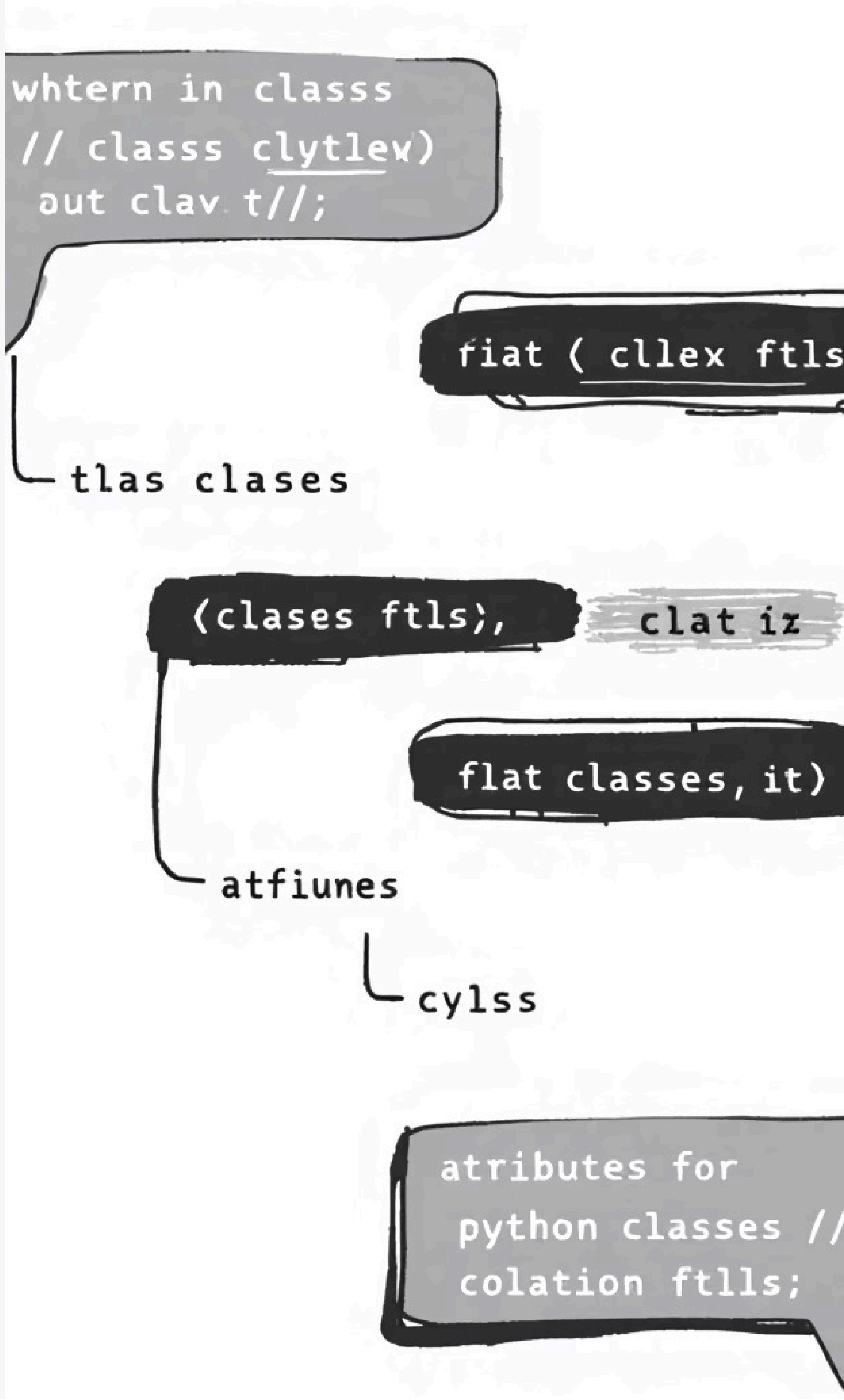
Define __init__ method with playlist name and empty track list

### Track Storage

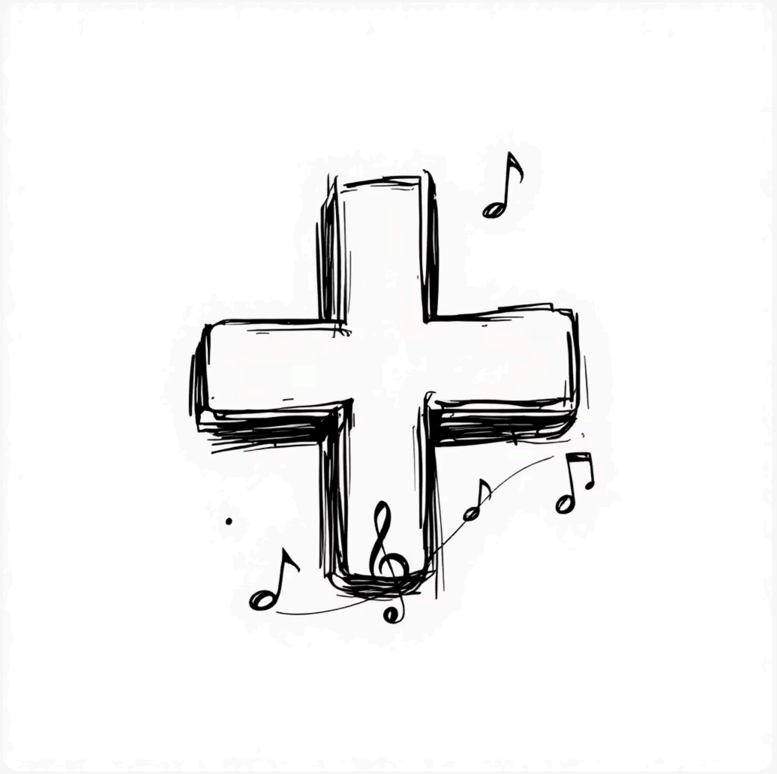
Use list data structure to store track dictionaries with title, artist, and duration

### Attributes

Include metadata like creation date, total duration, and track count


whtern in classs // classs clytlew) aut clav t//;
fiat ( cllex ftls
tlas clases
(clases ftls),
clat iz
flat classes, it)
atfiunes
cylss
atributes for python classes // colation ftlls;

# Implementing Track Addition



## The __add__ Operator

Python's __add__ special method allows us to use the + operator for adding tracks naturally, making code more readable and intuitive.

**Example:** playlist + new_track creates a seamless addition experience

### Define Method

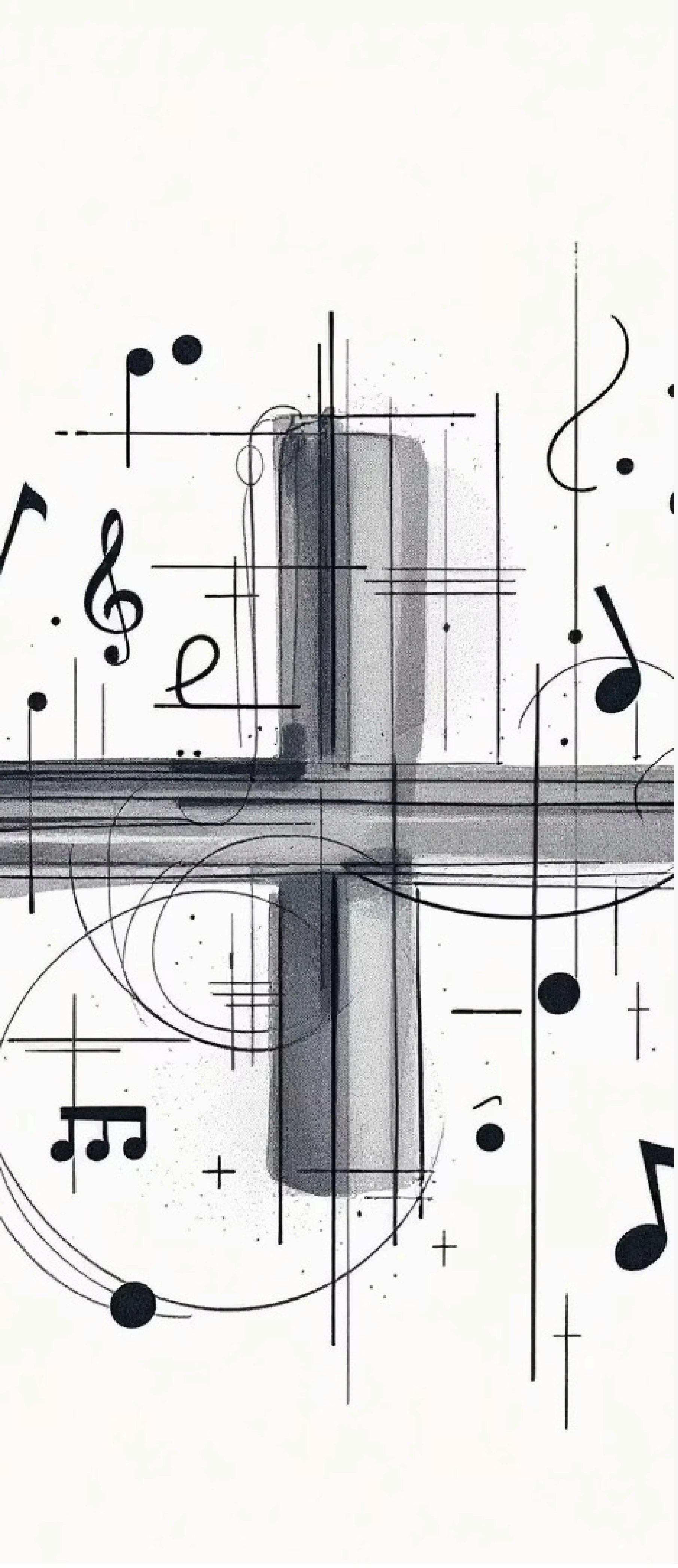
Create __add__ accepting track object

### Validate Input

Ensure track has required fields

### Append Track

Add to internal list and return self



## Track Removal Operations

The __sub__ operator enables intuitive track removal, maintaining consistency with mathematical conventions whilst providing powerful playlist editing capabilities.

01

Implement __sub__ Method

Accept track name or track object as parameter for flexible removal options

02

Search and Match

Iterate through playlist to find matching track by title or object reference

03

Remove Safely

Remove track from list and update metadata, handling non-existent tracks gracefully

04

Return Updated Playlist

Return self to enable method chaining for multiple operations

## Additional Operators

### __len__ Operator

Returns total number of tracks in playlist, enabling len(playlist) syntax

- Provides quick track count
- Essential for iteration and validation

### __str__ Operator

Returns formatted string representation of playlist for printing and debugging

- Displays playlist name and tracks
- Shows total duration and count

### __contains__ Operator

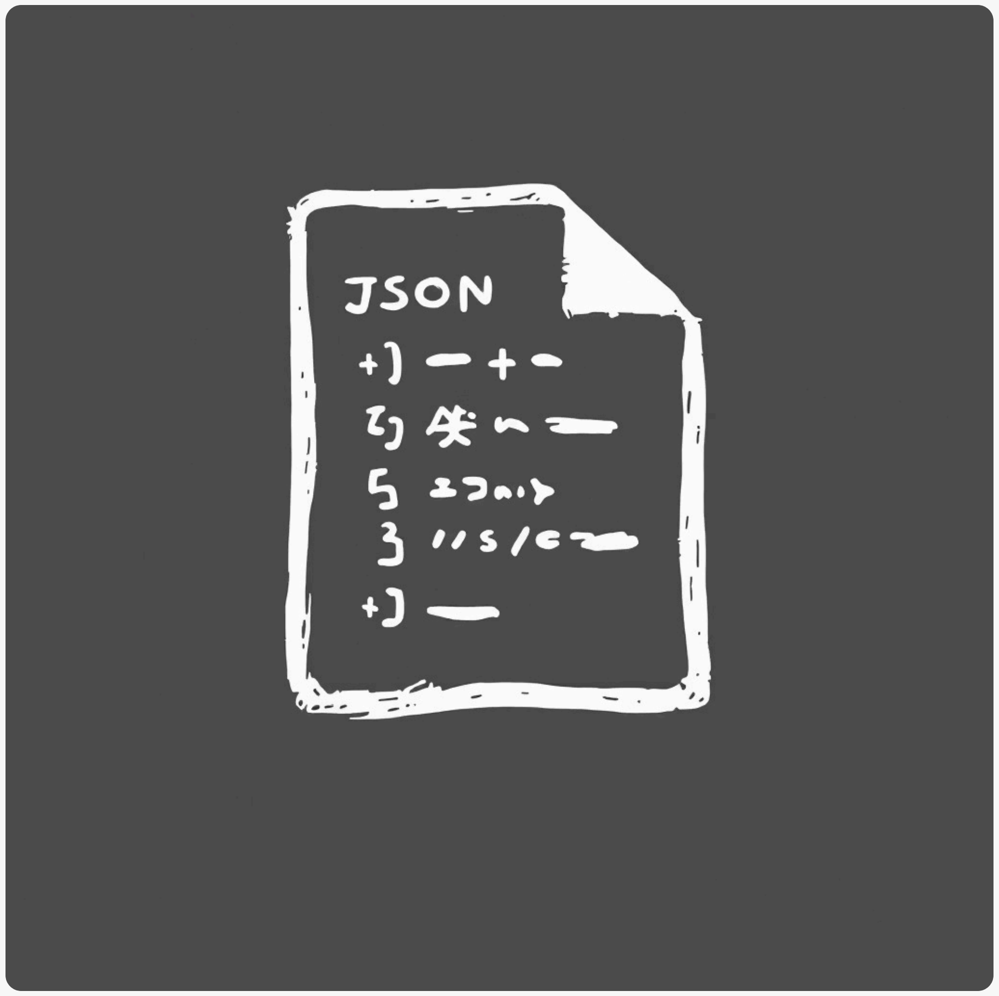
Enables 'in' keyword to check track existence: track in playlist

- Searches by track title or object
- Returns boolean result

These operators create a Pythonic interface that feels natural and intuitive, reducing cognitive load whilst maintaining powerful functionality.

## File I/O Implementation

### JSON Format



- Use json.dump() for serialisation
- Store tracks as list of dictionaries
- Include metadata and timestamps
- Human-readable and widely supported

### CSV Format

- Utilise csv.writer() for tabular data
- One track per row with headers
- Compatible with Excel and databases
- Efficient for large playlists

---

Implement save_to_file() and load_from_file() methods with format detection based on file extension, ensuring seamless data persistence across sessions.

## Error Handling

File Not Found

Catch FileNotFoundError when loading and provide helpful message to user

Invalid JSON/CSV

Handle parsing errors with try-except blocks and validate data structure

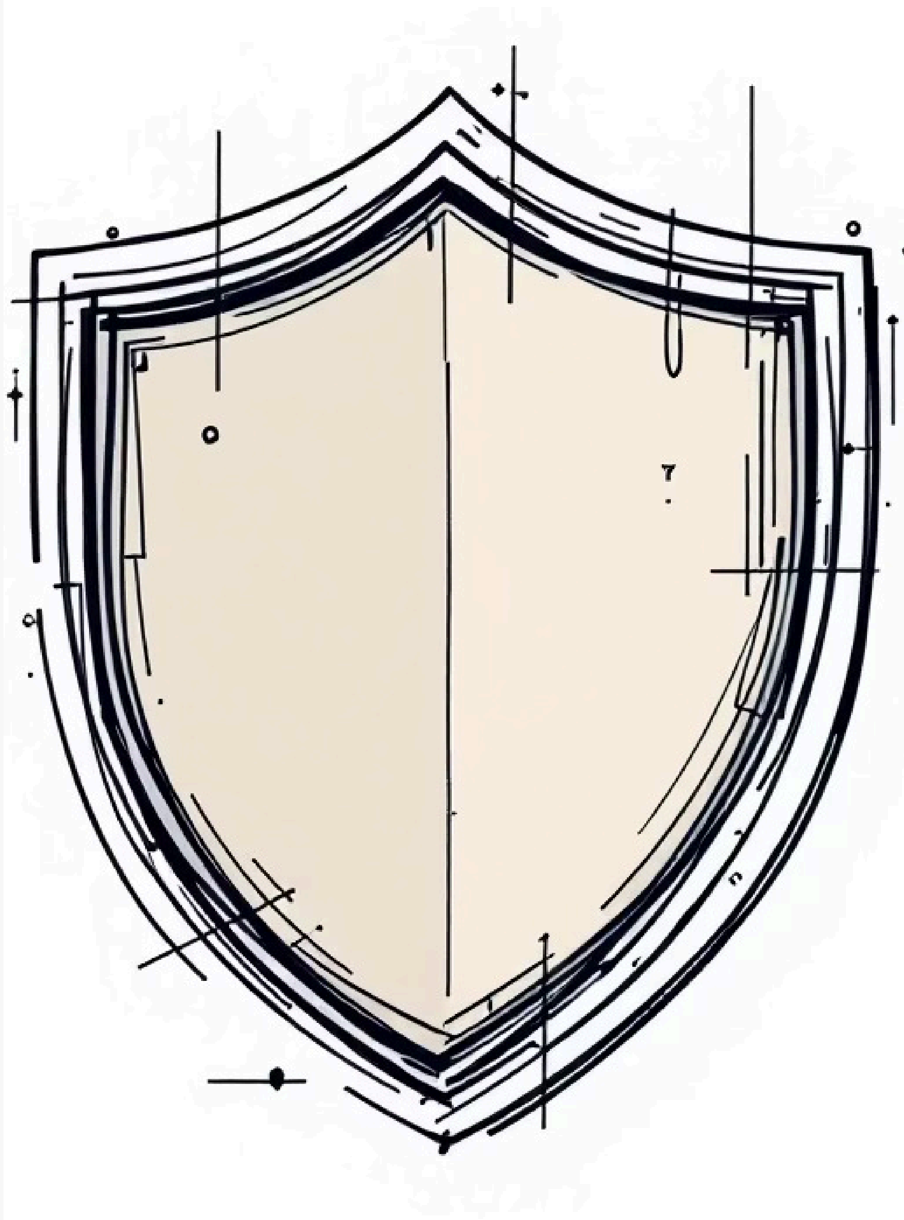
Duplicate Tracks

Check for existing tracks before addition to prevent duplicates

Permission Errors

Manage write permission issues and suggest alternative file locations

**Best Practice:** Always use context managers (with statements) for file operations to ensure proper resource cleanup, even when exceptions occur.



# Practical Demonstration

Let's see our Music Playlist Manager in action with real tracks and operations, showcasing the elegance of operator overloading.

1. **Create Playlist**

```
my_playlist = Playlist("Favourites")
```

2. **Add Tracks**

```
my_playlist + Track("Shape of You", "Ed Sheeran", 240)
```

3. **Remove Track**

```
my_playlist - "Shape of You"
```

4. **Check Contents**

```
"Blinding Lights" in my_playlist
```

5. **Save Playlist**

```
my_playlist.save_to_file("favourites.json")
```

## Key Takeaways

### Operator Overloading Power

Special methods like __add__ and __sub__ create intuitive, readable code that feels natural to Python developers

### File I/O Flexibility

Supporting multiple formats (JSON, CSV) ensures compatibility with various tools and workflows

### Robust Error Handling

Proper exception management creates reliable, production-ready applications that handle edge cases gracefully

### Next Steps for Enhancement

- Add search and filter functionality
- Implement playlist merging operators
- Create shuffle and sort methods
- Integrate with music streaming APIs
- Add graphical user interface
- Support collaborative playlists