Object-Oriented Programming in Python: Demystifying Classes and Objects

A Practical Music Playlist Case Study

Minor In AI, IIT Ropar
24th March, 2025

1 The Evolution of Data Management in Programming

Programming isn't just about storing data; it's about organizing and interacting with it efficiently. Let's explore how our approach to managing data evolves:

1.1 Traditional Approach: Simple Lists

Initially, programmers use simple data structures like lists to store information:

```
# A naive approach to storing song information
favorite_songs = ["Not Like Us - Kendrick Lamar",
                  "Kodak - Seedymau",
                  "Wavy - We end current Hojila"]

# Iterating and playing songs
for song in favorite_songs:
    print("Now playing:", song)
```

Limitations of this Approach:

- No clear separation between song attributes
- Difficult to add more details like duration, genre, or release year
- Limited ability to perform song-specific actions

2 Object-Oriented Programming: A Paradigm Shift

OOP introduces a revolutionary way of thinking about code. Instead of viewing data as passive information, we treat it as active, self-contained entities with their own properties and behaviors.

2.1 Classes: The Blueprint of Objects

A class is like a blueprint that defines the structure and behavior of objects:

```
class Song:
    def __init__(self, title, artist, duration):
        # Constructor method: Initializes object attributes
        self.title = title       # Public attribute
        self.artist = artist     # Public attribute
        self.duration = duration  # Public attribute

    # Method to represent object's behavior
    def play(self):
        print(f"Now playing: {self.title} by {self.artist}")
```

Key Concepts:

- `__init__` is a special method called when creating a new object
- `self` refers to the instance being created
- Methods define actions that objects can perform

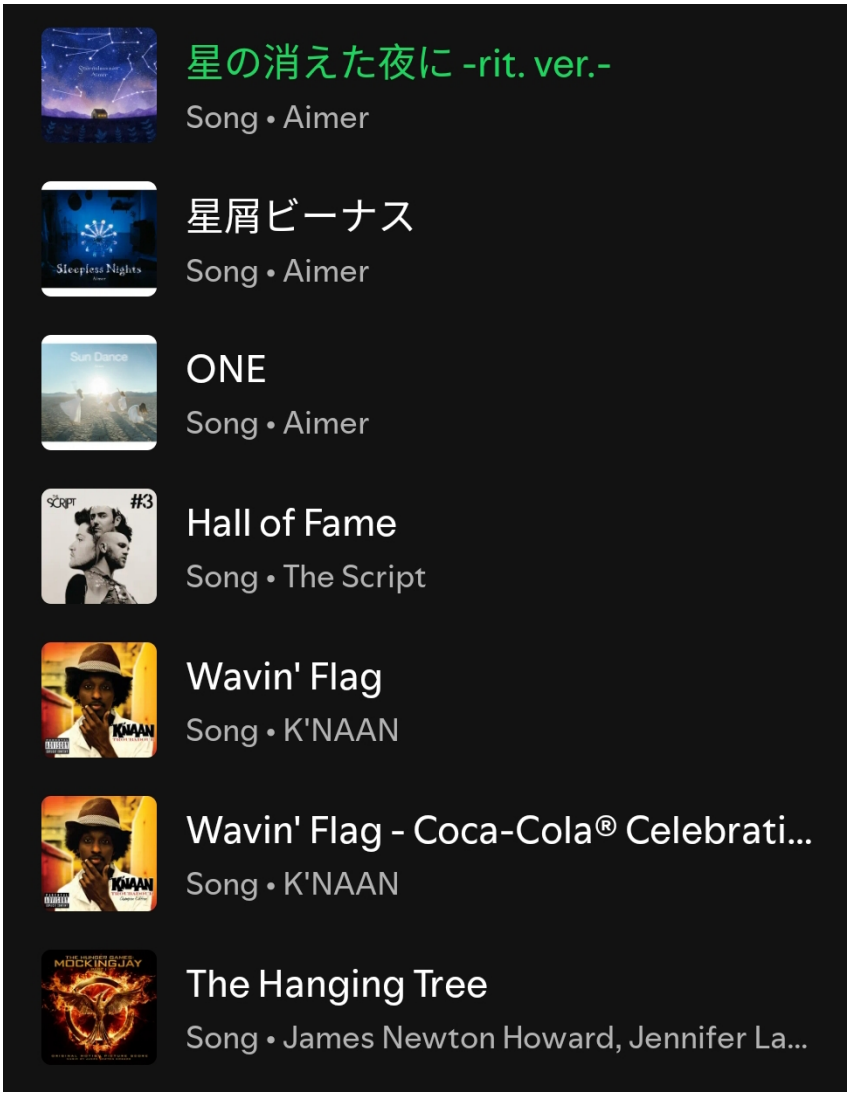

Figure 1: A list of songs!

3 Creating and Using Objects

Objects are specific instances of a class, each with its own unique data:

```
# Creating song objects
song1 = Song("Not Like Us", "Kendrick Lamar", 180)
song2 = Song("Kodak", "Seedymau", 200)
song3 = Song("Wavy", "We end current Hojila", 220)

# Accessing object attributes
print(song1.title)     # Output: Not Like Us
print(song2.artist)    # Output: Seedymau

# Invoking object methods
song1.play()  # Plays the specific song
```

Object Characteristics:

- Each object is a unique instance of the class

- Objects can have the same structure but different data
- Methods can be called directly on individual objects

4 Encapsulation: Data Protection and Control

Encapsulation is about protecting an object's internal state and providing controlled access:

```
class Song:
    def __init__(self, title, artist, duration):
        self.title = title
        self.artist = artist
        self.__duration = duration  # Private attribute

    # Getter method
    def get_duration(self):
        return self.__duration

    # Setter method with validation
    def set_duration(self, duration):
        if duration > 0:
            self.__duration = duration
        else:
            print("Invalid duration value.")
```

Encapsulation Benefits:

- Prevents direct manipulation of sensitive data
- Allows implementing validation logic
- Provides a clean interface for interacting with objects

5 Building Complex Systems: The Playlist Class

Combining multiple classes to create more complex applications:

```
class Playlist:
    def __init__(self):
        self.songs = []  # List to store Song objects

    def add_song(self, song):
        self.songs.append(song)

    def play_all(self):
        for song in self.songs:
            song.play()

# Creating and using a playlist
my_playlist = Playlist()
my_playlist.add_song(song1)
```

```
my_playlist.add_song(song2)
my_playlist.play_all()
```

Advanced OOP Concepts Demonstrated:

- Composition (Playlist contains Song objects)
- Separation of concerns
- Modular and extensible design

6 The Power of Object-Oriented Programming

- **Abstraction:** Simplify complex systems by modeling real-world entities
- **Modularity:** Break down problems into manageable, independent components
- **Reusability:** Create flexible, adaptable code structures
- **Maintainability:** Easier to understand, modify, and extend code

Remember: OOP is not just a programming technique, it's a way of thinking about software design!