Minor in AI
Title: Revision: OOP in Action

LO: OOP in Action: Case Studies on Encapsulation, Inheritance, and Object Design

24 March 2025

Case Study: Olympics 2024
Link: https://www.olympics.com/en/olympic-games/paris-2024

Observations:
Think of the Olympic Games. You have different people taking part, different sports, and different countries involved.

- You can think of each athlete, each sporting event, and each country as a separate type of thing with its own details and actions.
- Some athletes specialize in certain sports. For example, a swimmer, a runner, and a gymnast are all athletes, but they perform in different ways.
- Certain details about an athlete, like their health or training data, are kept private and are not shown to everyone.
- When it's time to compete, each sport has its own rules and ways of playing, so the way athletes take part in different events can vary.

This is similar to how programmers design systems—by thinking of real-world things, giving them properties and actions, keeping some details hidden, and allowing different behaviors depending on the situation.

Case Study: Music Play List App
What is Object Design?

Object design is about thinking in terms of real-world objects and modeling them in code. Each object has attributes (data) and methods (actions). This makes code organized, reusable, and easy to understand.

Without Object Design (Procedural Code)

```
songs = ["Song A", "Song B", "Song C"]

for song in songs:
    print("Playing:", song)
```

Problem:
We can’t store artist, duration, or add behaviors. Limited flexibility.
With Object Design (Using Classes and Objects)

```
class Song:
    def __init__(self, title, artist):
        self.title = title
        self.artist = artist

    def play(self):
        print("Playing:", self.title, "by", self.artist)

playlist = [Song("Song A", "Artist X"), Song("Song B", "Artist
Y")]

for song in playlist:
    song.play()
```

Explanation:

- Song is an object with title and artist.
- play is a method that prints the song.
- Objects make it easier to manage and scale the app.

What is self?

- self refers to the current object.
- It allows you to access or update the object’s own data inside methods.

Example:
In self.title, it means "this song’s title."
__init__ Function

- This is a special function that is called automatically when you create a new object from a class.
- It sets up the object with the details you give it.
- In this case, when you create a song, you provide its title and artist, and the function stores them for later use.

Notes on self:

- self is a name used inside a class to refer to the current object.
- Think of it like saying "me" when you talk about yourself. In the same way, self.title means "my title" — the title that belongs to this object.
- Without self, Python thinks you're talking about a temporary variable that disappears after the function ends.

Encapsulation:

What is Encapsulation?

Encapsulation means hiding data inside the object and only allowing access through methods. This helps protect data and control how it's used.

```
class Song:
    def __init__(self, title, artist, duration):
        self.title = title
        self.artist = artist
        self.duration = duration  # in seconds

song = Song("Song A", "Artist X", 180)
song.duration = -50  # Invalid, but allowed
print(song.duration)
```

Problem:

Anyone can change duration to a negative number. No protection.

With Encapsulation (Using Private Variable)

```
class Song:
    def __init__(self, title, artist, duration):
        self.title = title
        self.artist = artist
        self.duration = duration

song = Song("Song A", "Artist X", 180)
print(song.title)
print(song.duration)
```

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

    def get_duration(self):
        return self.__duration

    def set_duration(self, duration):
        if duration > 0:
            self.__duration = duration
        else:
            print("Invalid duration")

song = Song("Song A", "Artist X", 180)
song.set_duration(-50)  # Invalid duration
print(song.title)
print(song.get_duration())
```

Encapsulation means hiding details that don’t need to be shown to everyone.

Just like:

- You use a remote control without knowing exactly how it works inside.
- A car has an engine, but you just use the steering wheel and pedals — you don’t need to see the engine while driving.

Similarly in programming, some data or functions are meant to be kept private (hidden inside), and some can be used outside.

How do we hide data in Python?
We mark it private by adding two underscores __ in front of it.

What is Inheritance?
Inheritance lets you create new classes based on existing ones. You can reuse and extend the behavior of a base class. This reduces code duplication.

```
class Song:
    def __init__(self, title, artist):
        self.title = title
        self.artist = artist

class Podcast:
    def __init__(self, title, host):
        self.title = title
        self.host = host
```

Problem:
Both classes have title. Code is repeated.

```
class AudioContent:
    def __init__(self, title):
        self.title = title

class Song(AudioContent):
    def __init__(self, title, artist):
        super().__init__(title)
        self.artist = artist

    def play(self):
        print("Playing song:", self.title, "by", self.artist)

class Podcast(AudioContent):
    def __init__(self, title, host):
        super().__init__(title)
        self.host = host

    def play(self):
          print("Playing podcast:", self.title, "hosted by",
self.host)

song = Song("Song A", "Artist X")
podcast = Podcast("Episode 1", "Host Y")

song.play()
podcast.play()
```

What is super()?

- super() is used to call the parent class (also called base class) from the child class.
- It helps you reuse code from the parent class without rewriting it