OOP

Objectives

- Define Objectr-Oriented Programming
- Componets od OOP
- Pillars of OOP

What is Object-Orientred Programming (OOP)?

Object-Orientred Programming is the development of software systems centered on *objects, data, and class definitions*.

Encapsulation

Encapsulation is one of the four fundamental concepts of object-oriented programming (OOP). It refers to the **bundling of data (attributes) and methods (functions)** that operate on that data into a single unit, typically a **class**, and **restricting direct access** to some of the object's components.

In Python, we achieve encapsulation using:

- **Public members**: Accessible from anywhere.
- **Protected members** (`_member`): Conventionally internal use (not enforced).
- **Private members** (`__member`): Name-mangled to restrict external access.

Real-World Analogy: A Car

Imagine a **Car**:

- You (the driver) can **start**, **accelerate**, or **brake**.
- But you **can't directly change the engine's internal parameters** like fuel injection timing — that's encapsulated.

The **Car class** encapsulates the complexity and exposes a simple interface to the user.

Python Example Using a `Car` Class

```
class Car:
    def __init__(self, make, model):
        self.make = make              # public attribute
        self.model = model            # public attribute
        self._speed = 0               # protected attribute (convention)
        self.__engine_temp = 75       # private attribute (name mangled)
```

```
    def accelerate(self, increment):
        self._speed += increment
        self.__update_engine_temp()
        print(f"Accelerating... Current speed: {self._speed} mph")

    def brake(self):
        self._speed = max(0, self._speed - 10)
        print(f"Braking... Current speed: {self._speed} mph")

    def get_engine_temp(self):
        return self.__engine_temp

    def __update_engine_temp(self):
        # internal logic to update engine temp (hidden from user)
        self.__engine_temp += self._speed * 0.05

# Using the Car class
my_car = Car("Toyota", "Camry")
my_car.accelerate(20)
my_car.brake()

# Access public attribute
print(my_car.make)

# Access protected (possible but discouraged)
print(my_car._speed)

# Try to access private attribute (fails)
# print(my_car.__engine_temp)  # AttributeError

# Correct way to access private attribute (via method)
print("Engine temp:", my_car.get_engine_temp())
```

Why Encapsulation Matters

- **Prevents accidental modification** of internal data.
- **Improves modularity**: you can change internal implementation without affecting external code.
- **Encourages controlled access** to attributes (via getters/setters).

Summary

Access Modifier	Syntax	Access Level
Public	`self.x`	Anywhere
Protected	`self._x`	Subclasses & internal
Private	`self.__x`	Class-internal only

Abstaraction

Abstraction is the OOP concept of hiding **unnecessary internal details** and showing only the **essential features** of an object. It allows programmers to focus on **what an object does** instead of **how it does it**.

In Python, abstraction is often achieved using:

- **Abstract base classes (ABC)** with the `abc` module
- **Interfaces** (conceptually via abstract methods)

Real-World Analogy: A Car Dashboard

When you drive a car:

- You use the **steering wheel**, **pedals**, and **gear shifter** to control the car.
- You don't need to know **how** the transmission or engine works.

That's abstraction — the car hides the internal complexity and exposes only the necessary interface.

Python Example Using a `Car` Interface

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass

    @abstractmethod
    def stop(self):
        pass

class Car(Vehicle):
    def start(self):
        print("Car engine started")
```

```
    def stop(self):
        print("Car engine stopped")

# Using abstraction
my_car = Car()
my_car.start()
my_car.stop()
```

Why Abstraction Matters

- **Simplifies complex systems** by exposing only what's necessary.
- **Improves code readability and maintenance**.
- **Supports scalability** by allowing changes to internal implementation without affecting external code.

Abstraction Summary

Feature	Description
Purpose	Hide internal details
Technique	Abstract classes & methods
Result	Clean interfaces, reduced complexity

Abstraction enables developers to design systems that are easier to use and reason about — by focusing on **what** an object does, not **how** it does it.

Definition: Inheritance in Python

Inheritance allows a class (child or subclass) to acquire the properties and behaviors (attributes and methods) of another class (parent or superclass). It promotes **code reusability** and supports **hierarchical classification**.

Real-World Analogy: Vehicle → Car

- A **Vehicle** has common functionality like `start()` and `stop()`.
- A **Car**, **Bike**, or **Truck** inherits these and may add specific features.

Python Example Using Inheritance

```python
class Vehicle:
    def start(self):
        print("Vehicle starting...")

    def stop(self):
        print("Vehicle stopping...")

class Car(Vehicle):
    def play_music(self):
        print("Playing music in the car.")

# Using inheritance
my_car = Car()
my_car.start()       # Inherited method
my_car.play_music()  # Subclass-specific method
```

Why Inheritance Matters

- **Promotes code reuse**
- **Establishes relationships** between general and specialized classes
- **Supports polymorphism**

Inheritance Summary

Feature	Description
Purpose	Reuse code from existing classes
Technique	Subclass extends superclass
Benefit	Faster development, logical structure

Inheritance helps create structured and maintainable class hierarchies.

Polymorphism

Definition: Polymorphism in Python

Polymorphism allows objects of different classes to be treated as if they were instances of the same class. It enables a **single interface to control access** to different types of objects.

Real-World Analogy: A Remote Control

- A remote control can operate a **TV**, **AC**, or **fan**, even though each device reacts differently.
- The interface (remote) stays the same, but the behavior depends on the device.

Python Example Using Polymorphism

```
class Vehicle:
    def sound(self):
        print("Generic vehicle sound")

class Car(Vehicle):
    def sound(self):
        print("Vroom!")

class Bike(Vehicle):
    def sound(self):
        print("Zoom!")

# Using polymorphism
for vehicle in [Car(), Bike()]:
    vehicle.sound()
```

Why Polymorphism Matters

- **Simplifies code** by using a common interface
- **Supports flexible and scalable design**
- **Encourages interchangeable components**

Polymorphism Summary

Feature	Description
Purpose	Shared interface for varied behavior
Technique	Method overriding or duck typing
Benefit	More reusable and adaptable code

Polymorphism helps in designing systems that are **extensible**, **clean**, and **easy to maintain**.