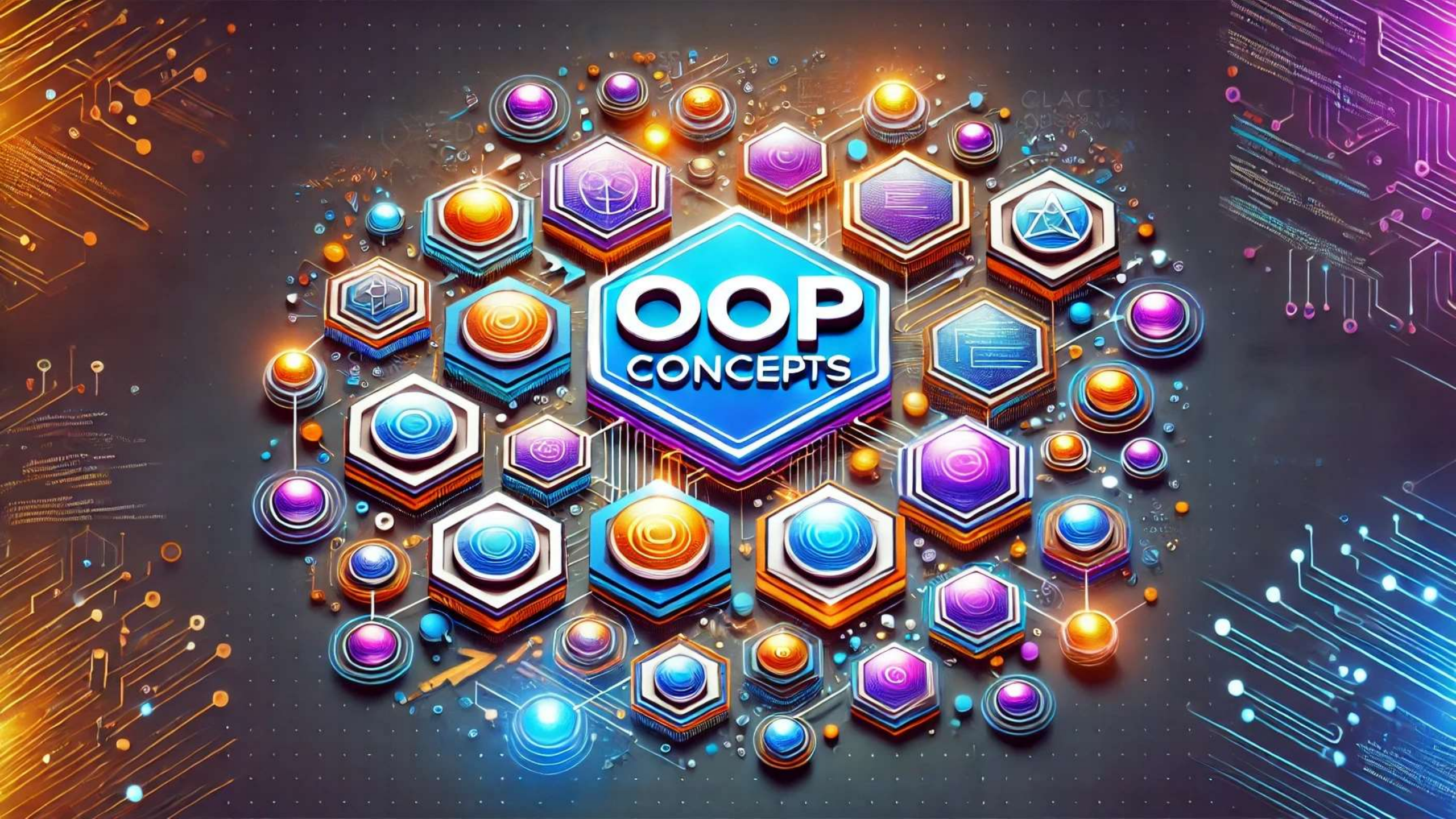
OOP
CONCEPTS

Mapping out OOP

- Vatsal Shah

Table of Contents

What is OOP?

- **Object-Oriented Programming (OOP)** is a programming paradigm that uses "objects" (which contain both data and behavior) to design and build applications.

- **Why OOP?**
  - Improves **code organization** by grouping related data (attributes) and behaviors (methods).
  - Makes **maintenance** and **scaling** easier through concepts like **inheritance** and **polymorphism**.
  - Helps us think in **real-world terms**: objects in your program often map to objects in the real world.

- **Real-Life Analogy**
- Think of **OOP** like setting up a **factory** that produces cars:
- A **car blueprint** (class) defines what every car should look like and how it behaves.
- Each **car** (object/instance) built from that blueprint has its own paint color, mileage, license plate, etc., but shares the same fundamental structure and features.

Classes and Objects

- **Class**
  - A **class** is like a **blueprint** or **template** that defines the *structure* (attributes) and *behavior* (methods) that the objects created from it will have.

- **Object (Instance)**
  - An **object** is an **instance** of a class. When you create an object, you make a "real thing" based on that blueprint.

- **Basic Syntax**

```
class ClassName:
# Class body: define attributes and methods here
    pass
# Creating an object (instance) of ClassName
    obj = ClassName()
```

Example: Creating a Simple Class

- Let's start with a simple real-life object: a **Dog**.

```
class Dog:
    pass  # We'll fill this in later
    # Create two Dog objects (instances)
    dog1 = Dog()
    dog2 = Dog()
    print(type(dog1))  # <class '__main__.Dog'>
    print(type(dog2))  # <class '__main__.Dog'>
```

- Right now, **Dog** is an empty class. dog1 and dog2 are objects of that class (they exist in memory), but they have no special attributes or behaviors yet.

Attributes (Instance Variables) and Methods

- **Attributes** (sometimes called “properties” or “fields”) store the data about an object.
- **Methods** are functions defined inside a class that describe the behaviors of the object.

- **The __init__ Method (Constructor)**
  - In Python, __init__ is a special method that **initializes** each instance of a class. You typically pass in parameters that set up the initial state (attributes) of the object.

Example with Attributes and Methods

```
class Dog:
    def __init__(self, name, breed, age):
        self.name = name      # instance attribute
        self.breed = breed    # instance attribute
        self.age = age        # instance attribute
    def bark(self):
        print(f"{self.name} says Woof!")
    def get_info(self):
        return f"Dog(name={self.name}, breed={self.breed}, age={self.age})“
# Creating objects (instances)
dog1 = Dog("Buddy", "Golden Retriever", 3)
dog2 = Dog("Lucy", "Poodle", 5)
print(dog1.name)                    # Buddy
print(dog2.breed)                   # Poodle
dog1.bark()                  # Buddy says Woof!
print(dog2.get_info())       # Dog(name=Lucy, breed=Poodle, age=5)
```

- **Real-Life Scenario**

Each dog (dog1 or dog2) has **its own attributes**: name, breed, age.

They share the **same behaviors**: e.g., bark, get_info.

Encapsulation

- **Encapsulation** is about **bundling data (attributes) and methods** that work on that data within one unit—a class. It helps in:
  - **Hiding the internal state** of objects from the outside world.
  - **Controlling access** or modification of the data.

- **Private Attributes (Convention in Python)**
  - Although Python doesn't enforce private attributes like some languages, the **convention** is to use an underscore prefix _ or double underscore __ to signal "this is private or internal."

Private Attributes (Convention in Python)

```
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        self.account_holder = account_holder
        self.__balance = initial_balance  # intended as "private"
    def deposit(self, amount):
        self.__balance += amount
    def withdraw(self, amount):
        if amount > self.__balance:
            print("Insufficient funds.")
        else:
            self.__balance -= amount
    def get_balance(self):
        return self.__balance
account = BankAccount("Alice", 1000)
account.deposit(500)
print(account.get_balance())  # 1500
```

Private Attributes

If we try to directly access the private balance, it won't show up as expected:

print(account.__balance) # AttributeError in many cases

- We use __balance to indicate it’s private.
- Encapsulation ensures everything to do with **balance** is handled internally by deposit and withdraw methods.

- **Real-Life Scenario**
- Imagine your bank account: you can deposit/withdraw (public methods), but you **cannot** just open the bank’s vault and change your balance. Access is controlled.

Inheritance

- **Inheritance** allows you to create a **new class** (child) that reuses (inherits) attributes and methods from an **existing class** (parent).

- **Why Inheritance?**

1. Avoid code duplication.
2. Model **is-a** relationships. (e.g., a Car is a Vehicle, a Manager is an Employee).

Example: Vehicle -> Car, Truck

```
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    def start(self):
        print(f"{self.make} {self.model} starting...")
    def stop(self):
        print(f"{self.make} {self.model} stopping...")
# Child class Car inherits from Vehicle
class Car(Vehicle):
    def __init__(self, make, model, year, num_doors):
        super().__init__(make, model, year)  # call parent constructor
        self.num_doors = num_doors
    def honk(self):
        print("Car honks: Beep beep!")
# Another child class Truck
class Truck(Vehicle):
    def __init__(self, make, model, year, cargo_capacity):
        super().__init__(make, model, year)
        self.cargo_capacity = cargo_capacity
    def load_cargo(self, weight):
        if weight <= self.cargo_capacity:
            print(f"Loading {weight} kg of cargo.")
        else:
            print("Over capacity!")
```

Vehicle -> Car, Truck

- Using them

```
car1 = Car("Toyota", "Camry", 2021, 4)
car1.start()     # Toyota Camry starting...
car1.honk()      # Car honks: Beep beep!

truck1 = Truck("Ford", "F-150", 2019, 1000)
truck1.start()   # Ford F-150 starting...
truck1.load_cargo(500)  # Loading 500 kg of cargo.
```

- **Real-Life Scenario**

A **Car** *is a* **Vehicle**—it inherits start() and stop() methods.

We add unique features to each child class (like honk in Car and load_cargo in Truck).

Polymorphism

- **Polymorphism** means “many forms.” In OOP, it refers to using the **same interface** (same method name) for **different underlying forms** (different classes).

- **Example: speak() Method in Different Classes**

```
class Animal:
    def speak(self):
        pass  # "abstract" or unimplemented in base class
class Dog(Animal):
    def speak(self):
        print("Woof!")
class Cat(Animal):
    def speak(self):
        print("Meow!")
class Cow(Animal):
    def speak(self):
        print("Moo!")
# Polymorphism in action
animals = [Dog(), Cat(), Cow()]
for animal in animals:
animal.speak()
# The same method name 'speak' triggers different behavior
# Output: # Woof! # Meow! # Moo!
```

- **Real-Life Scenario**

You call speak() on any **Animal**, but the actual “sound” depends on whether it’s a Dog, Cat, or Cow.

Abstraction

- **Abstraction** is about **hiding** the "complexity" and only showing the **essential** parts. In Python, we often use **abstract base classes (ABC)** or simply use "not implemented" methods in a parent class to define a **common interface**.
- **Example with Abstract Base Class**

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14159 * (self.radius ** 2)
        shapes = [Rectangle(4, 5), Circle(3)]
        for s in shapes:
            print(s.area())
            # 20   # 28.27431
```

Here, Shape is an **abstract class** with the abstract method area().

Rectangle and Circle **implement** area() in their own ways.

Advanced Topics

- **Class Methods and Static Methods**
- **Class methods** (@classmethod) receive the class itself (cls) as the first argument. Often used for **alternative constructors** or class-wide behaviors (e.g., creating objects from a database record).
- **Static methods** (@staticmethod) don’t get self or cls; they are just utility functions that logically belong to the class.

Class Methods and Static Methods

```
class Book:
    library_name = "City Library"  # class attribute
    def __init__(self, title, author):
        self.title = title
        self.author = author
    @classmethod
    def from_string(cls, book_str):
    # example of an alternative constructor
        title, author = book_str.split(";")
        return cls(title, author)
    @staticmethod
    def is_valid_isbn(isbn):
    # Some check not involving self or cls
        return len(isbn) == 13
    # Using from_string() class method
book1 = Book("Python 101", "John Doe")
book2 = Book.from_string("OOP Mastery;Jane Smith")
print(book1.title)          # Python 101
print(book2.author)         # Jane Smith
print(Book.is_valid_isbn("1234567890123"))  # True (example)
```

Properties and Getters/Setters

- In Python, we often define getters/setters using the @property decorator instead of explicit methods like get_x(), set_x().

```
class Person:
    def __init__(self, name):
        self._name = name
    @property
    def name(self):
    # This acts like a "getter"
        return self._name
    @name.setter
    def name(self, new_name):
    # This acts like a "setter"
        if len(new_name) < 2:
            raise ValueError("Name too short!")
        self._name = new_name


p = Person("Alice")
print(p.name)  # calls @property
p.name = "Bob"  # calls @name.setter
```

- **@property** allows p.name to be used as if it's just an attribute, but behind the scenes, you run code (like validation).

Multiple Inheritance

- Python supports **multiple inheritance**—a class can have **more than one parent**. Use carefully (it can get complicated).

```
class Flyer:
    def fly(self):
        print("Flying...")
class Swimmer:
    def swim(self):
        print("Swimming...")
class Duck(Flyer, Swimmer):
    pass
d = Duck()
d.fly()   # Flying...
d.swim()  # Swimming...
```

Mixins

- A **Mixin** is typically a small class that “injects” extra methods without being the primary parent. Often used to add cross-cutting functionality (e.g., logging, serialization) to many classes.

```
class JSONMixin:
    def to_json(self):
        import json
        return json.dumps(self.__dict__)
class Person(JSONMixin):
    def __init__(self, name, age):
        self.name = name
        self.age = age
p = Person("Alice", 30)
print(p.to_json())  # {"name": "Alice", "age": 30}
```

JSONMixin just provides a to_json() method, which Person can use without rewriting JSON logic.

Special Methods (Magic Methods)

- Python classes can implement **special methods** (also called “dunder methods,” e.g., __str__, __repr__, __len__, __eq__, etc.) to integrate seamlessly with Python’s built-in operations.
- For instance:

```
class Card:
    def __init__(self, suit, rank):
        self.suit = suit  # e.g., "Hearts"
        self.rank = rank  # e.g., "Ace"
    def __str__(self):
        return f"{self.rank} of {self.suit}"
    def __repr__(self):
        return f"Card({self.suit!r}, {self.rank!r})"
card = Card("Hearts", "Ace")
print(card)          # Ace of Hearts  (calls __str__)
print(repr(card))    # Card('Hearts', 'Ace')
```

__str__ is used by print().

__repr__ is the “official” string representation, used in debugging or console logs.

- You can also implement operator overloading with methods like __add__, __sub__, __mul__, etc., if it makes sense in your domain (e.g., adding two vectors or complex numbers).

Wrapping Up

- You've now seen how **Object-Oriented Programming** works in Python from basic to more advanced topics:

- **Classes and Objects**: The blueprint vs. instance concept.
- **Attributes and Methods**: Data and behaviors inside classes.
- **Encapsulation**: Controlling access to data.
- **Inheritance**: Creating subclasses that reuse parent functionality.
- **Polymorphism**: Same method name, different underlying classes.
- **Abstraction**: Hiding complexity with abstract base classes.
- **Advanced Topics**: Class/Static methods, properties, multiple inheritance, mixins, magic methods, etc.

Real-Life Application Flow

- When designing software with OOP, think about:
- **Identifying** objects/actors in your problem domain (e.g., Car, Driver, Account, Transaction).
- **Defining Classes** that represent those objects and their relationships (who "has-a" what, who "is-a" what).
- **Encapsulating** logic (methods) with the data it manipulates (attributes).
- Using **inheritance** if you need specialized versions of more general classes.
- Considering **polymorphism** if multiple classes share a common interface but differ in implementation.
- With practice, OOP becomes a powerful way to write organized, maintainable, and scalable code in Python.

Happy coding with OOP!


THANK YOU!