# **object Oriented Programming**

<ul><li>O Created</li></ul>	@June 28, 2025 5:55 AM
≔ Tags	

# **Oops**

Object-Oriented Programming (OOP) is a paradigm based on the concept of "objects", which can contain data (attributes) and code (methods). Python is a multi-paradigm language, but it supports OOP as a first-class citizen.

Below is an in-depth explanation of OOP concepts in Python:



# 1. Classes and Objects

#### ➤ Class:

A blueprint for creating objects. It defines attributes and methods.

```
class Dog:
  def __init__(self, name, breed):
    self.name = name
    self.breed = breed
  def bark(self):
    return f"{self.name} says Woof!"
```

### ➤ Object:

An instance of a class.

```
my_dog = Dog("Buddy", "Golden Retriever")
print(my_dog.bark()) # Buddy says Woof!
```



### 2. Four Pillars of OOP

# A. Encapsulation

Encapsulation binds data (attributes) and methods (functions) into a single unit and hides internal details.

```
class Account:
  def __init__(self, balance):
    self.__balance = balance # private variable
  def deposit(self, amount):
    if amount > 0:
       self.__balance += amount
  def get_balance(self):
    return self. balance
```

#### **Key Concepts:**

- \_balance is private.
- Access controlled via methods (get\_balance, deposit).

## B. Abstraction

Hides complex implementation and shows only necessary features.

```
from abc import ABC, abstractmethod
class Vehicle(ABC):
  @abstractmethod
  def start_engine(self):
    pass
class Car(Vehicle):
```

```
def start_engine(self):
return "Car engine started"
```

#### **Key Concepts:**

- Vehicle is an abstract base class.
- Enforces implementation in subclasses.

## C. Inheritance

Allows a class to inherit attributes and methods from another class.

```
class Animal:
    def speak(self):
        return "Animal sound"

class Dog(Animal):
    def speak(self):
    return "Bark"
```

#### **Types of Inheritance in Python:**

- **Single**: One base class → One derived class.
- **Multiple**: Multiple base classes → One derived class.
- Multilevel: Derived class inherits from a class that inherits from another.
- **Hierarchical**: One base → Multiple derived.
- **Hybrid**: A mix of the above.

# **O. Polymorphism**

Polymorphism allows different classes to be treated as instances of the same class through a common interface.

```
class Bird:

def make_sound(self):
```

```
return "Tweet"

class Cat:
    def make_sound(self):
        return "Meow"

def animal_sound(animal):
    print(animal.make_sound())

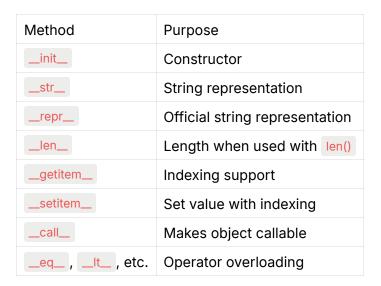
animal_sound(Bird())
animal_sound(Cat())
```

#### **Key Concepts:**

- Duck typing: "If it looks like a duck and quacks like a duck..."
- Method Overriding is supported (runtime polymorphism).
- Python doesn't support method overloading natively (no compile-time polymorphism).

# 3. Special (Dunder) Methods

These methods enable Python's syntactic sugar.



```
class Point:
  def __init__(self, x, y):
     self.x = x
     self.y = y
  def __str__(self):
     return f"({self.x}, {self.y})"
p = Point(1, 2)
print(p) # (1, 2)
```

# 4. Access Modifiers in Python

Python doesn't enforce strict access control, but uses conventions:

Modifier	Prefix	Visibility
Public	none	Everywhere
Protected	_name	Subclass only
Private	name	Class only (name mangling)

# 🔰 5. Static, Class, and Instance Methods

```
class MyClass:
  class_var = "Class Level"
  def __init__(self, value):
    self.instance_var = value
  @staticmethod
  def static_method():
    return "No self or cls"
  @classmethod
```

```
def class_method(cls):
    return f"Accessing {cls.class_var}"

def instance_method(self):
    return f"Accessing {self.instance_var}"
```

- static\_method(): No access to class or instance.
- class\_method(): Accesses class state (cls).
- instance\_method(): Accesses instance state (self).

# 6. Composition vs Inheritance

#### ➤ Inheritance:

"Is-a" relationship.

```
class Engine: ...
class Car(Engine): ... # Car is an Engine
```

### > Composition:

"Has-a" relationship.

```
class Engine: ...
class Car:
   def __init__(self):
     self.engine = Engine() # Car has an Engine
```

# 7. Multiple Inheritance and MRO (Method Resolution Order)

Python uses **C3 Linearization** for resolving the order in which base classes are initialized.

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass
print(D.__mro__)
```

# 8. Example: Real-World Simulation

```
class Employee:
    def __init__(self, name, emp_id):
        self.name = name
        self.emp_id = emp_id

    def get_details(self):
        return f"{self.name}, ID: {self.emp_id}"

class Manager(Employee):
    def __init__(self, name, emp_id, department):
        super().__init__(name, emp_id)
        self.department = department

def get_details(self):
    base = super().get_details()
    return f"{base}, Dept: {self.department}"
```

# 9. Why OOP in Python?

- Improves modularity and code reuse
- Makes large codebases manageable
- Encourages clean design patterns (e.g., MVC, Factory)
- Python's syntax makes OOP concise and intuitive