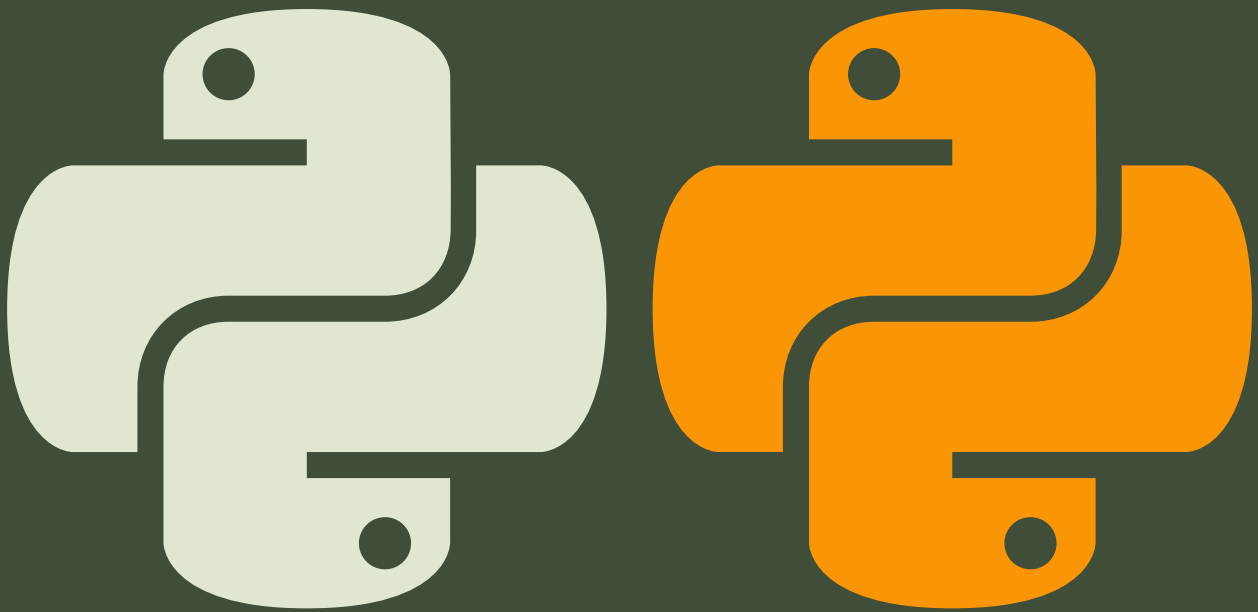


Python series



# Object-Oriented Programming Concepts

With Code Examples

Advanced Level

# Introduction


Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which are instances of classes.

The fundamental principles of OOP include:

1. **Classes and Objects:** A class is a blueprint or template for creating objects, which are instances of that class. Objects have attributes (data) and methods (behavior).
2. **Inheritance:** Inheritance allows new classes to be based on existing classes, inheriting and reusing their attributes and methods. This promotes code reuse and helps in creating hierarchical relationships between classes.
3. **Encapsulation:** Encapsulation is the mechanism of hiding the implementation details of an object's internal state and providing a well-defined interface for interacting with the object.
4. **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling code to work with objects of different types without knowing their specific implementation details.

# Classes and Objects

A class is a blueprint for creating objects. An object is an instance of a class, with its own attributes and methods.



```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} says: Woof!")

my_dog = Dog("Buddy", "Labrador")
my_dog.bark() # Output: Buddy says: Woof!
```

# Inheritance

Inheritance allows a class to inherit attributes and methods from another class, promoting code reuse and creating a hierarchical relationship between classes.

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

    def speak(self):
        print(f"{self.name} makes a sound.")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} says: Woof!")

my_animal = Animal("Buddy")
my_animal.speak() # Output: Buddy makes a sound.

my_dog = Dog("Buddy")
my_dog.speak() # Output: Buddy says: Woof!
```

# Encapsulation

Encapsulation is the mechanism of hiding data implementation details and restricting access to object's internal state. It promotes data abstraction and code modularity.

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance # Underscore convention for "private"
        attribute

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if amount <= self._balance:
            self._balance -= amount
        else:
            print("Insufficient balance.")

account = BankAccount(1000)
account.deposit(500)
account.withdraw(2000) # Output: Insufficient balance.
```

# Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables code to work with objects of different types without knowing their specific implementation details.

```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement this method")

class Dog(Animal):
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def speak(self):
        print("Meow!")

animals = [Dog(), Cat()]
for animal in animals:
    animal.speak()
```

# Methods

Methods are functions defined within a class that operate on the objects of that class. They allow objects to perform actions and manipulate their internal state.



```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * (self.radius ** 2)

    def circumference(self):
        return 2 * 3.14 * self.radius

circle = Circle(5)
print(circle.area())           # Output: 78.5
print(circle.circumference())  # Output: 31.4
```

# Class and Static Methods

Class methods operate on the class itself, while static methods are utility functions that operate independently of any specific instance or class.



```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

    @classmethod
    def multiply(cls, a, b):
        return cls(a * b)

print(Math.add(2, 3)) # Output: 5
result = Math.multiply(2, 3)
print(result) # Output: 6
```



# Inheritance and Overriding

Subclasses can override methods from their superclasses to provide custom behavior while still inheriting and reusing code from the parent class.

```
class Animal:
    def speak(self):
        print("The animal makes a sound.")


class Dog(Animal):
    def speak(self):
        print("The dog barks.")

class Cat(Animal):
    def speak(self):
        print("The cat meows.")

animals = [Dog(), Cat(), Animal()]
for animal in animals:
    animal.speak()
```

# Multiple Inheritance

Python allows a class to inherit from multiple base classes, enabling code reuse and the combination of behaviors from different classes.



```
class Flyable:
    def fly(self):
        print("I can fly!")

class Walkable:
    def walk(self):
        print("I can walk!")

class Bird(Flyable, Walkable):
    pass

bird = Bird()
bird.fly()      # Output: I can fly!
bird.walk()    # Output: I can walk!
```

# Composition

Composition is an alternative to inheritance, where an object contains instances of other objects as attributes, allowing for code reuse and flexible object composition.



```
class Engine:
    def start(self):
        print("Engine started.")

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        self.engine.start()
        print("Car started.")

car = Car()
car.start()
```

# Abstract Base Classes

Abstract base classes define a common interface that must be implemented by concrete subclasses. They provide a way to enforce contracts and ensure consistent behavior across related classes.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

rectangle = Rectangle(5, 3)
print(rectangle.area())           # Output: 15
print(rectangle.perimeter())      # Output: 16
```

# Property Decorators

Property decorators in Python provide a way to define attributes with getter, setter, and deleter methods, allowing for controlled access and validation of object attributes.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative.")
        self._radius = value

    @radius.deleter
    def radius(self):
        del self._radius

circle = Circle(5)
print(circle.radius)  # Output: 5
circle.radius = 10
print(circle.radius)  # Output: 10
del circle.radius
```

# Method Overloading

Abstract base classes define a common interface that must be implemented by concrete subclasses. They provide a way to enforce contracts and ensure consistent behavior across related classes.

```
class Calculator:
    def add(self, a, b, c=0):
        return a + b + c

    def add(self, *args):
        total = 0
        for arg in args:
            total += arg
        return total

calc = Calculator()
print(calc.add(1, 2))           # Output: 3
print(calc.add(1, 2, 3))       # Output: 6
print(calc.add(1, 2, 3, 4))    # Output: 10
```

# Operator Overloading

Python allows you to overload operators for custom classes by defining special methods, enabling more intuitive and expressive code when working with objects of those classes.



```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector2D(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"

v1 = Vector2D(1, 2)
v2 = Vector2D(3, 4)
v3 = v1 + v2
print(v3) # Output: (4, 6)
```

# Exception Handling

Exception handling is a crucial aspect of robust and maintainable code. Python provides a built-in mechanism for catching and handling exceptions using try-except blocks.

```
class CustomException(Exception):  
    pass  
  
def divide(a, b):  
    if b == 0:  
        raise CustomException("Cannot divide by zero.")  
    return a / b  
  
try:  
    result = divide(10, 0)  
    print(result)  
except CustomException as e:  
    print(e)  
except ZeroDivisionError:  
    print("Division by zero error.")
```



# Pitfalls and Best Practices

While OOP offers many benefits, it's essential to be mindful of potential pitfalls, such as complexity due to deep inheritance hierarchies, tight coupling between classes, and misuse of design patterns. Striking the right balance and following best practices is crucial for maintainable and scalable code.

