12
Object Oriented Programming

Object-Oriented Programming (OOP) is a powerful paradigm that organizes code around objects, making programs modular, reusable, and easier to maintain. Unlike procedural programming, which focuses on functions and data separately, OOP encapsulates both within objects, allowing for better structure and scalability. Python supports OOP alongside other paradigms, making it a flexible language for various applications.

In this chapter, you will learn the fundamental principles of OOP and how to apply them effectively in Python.

- **Procedural vs. Object-Oriented Programming** – Understanding how OOP differs from procedural programming and its advantages.
- **Classes and Objects** – The foundation of OOP, where classes define blueprints and objects represent real-world instances.
- **Attributes and Methods** – Defining object properties (attributes) and behaviors (methods) to encapsulate functionality.
- **Encapsulation and Data Hiding** – Controlling access to attributes using public, protected, and private variables.
- **Inheritance and Code Reusability** – Reusing and extending existing code through parent and child class relationships.
- **Polymorphism and Method Overriding** – Allowing different classes to implement the same interface for flexible code.
- **Special Methods and Operator Overloading** – Enhancing object behavior with Python's built-in dunder methods.
- **Class and Static Methods** – Understanding @classmethod and @staticmethod for class-level functionality.
- **Abstract Classes and Interfaces** – Enforcing method implementation with abstract base classes (abc module).
- **Composition vs. Inheritance** – Structuring objects with the "has-a" relationship instead of deep inheritance trees.
- **Design Principles in OOP** – Applying best practices such as the SOLID principles for maintainable code.
- **Common Pitfalls and Best Practices** – Avoiding mistakes like deep inheritance, tight coupling, and overusing getters/setters.

By the end of this chapter, you will be able to design, implement, and optimize object-oriented programs in Python, creating scalable and efficient applications.

Procedural vs. Object-Oriented Programming

Before diving into OOP, it's helpful to understand how it differs from procedural programming, a more traditional approach. In procedural programming, code is organized around functions and data structures. A program follows a step-by-step execution, where functions manipulate data that is often stored in global variables or structured data types like dictionaries or lists.

For example, a simple procedural approach to managing a bank account might involve storing account balances in a dictionary and defining functions to deposit or withdraw money:

Procedural approach

```
accounts = {"Alice": 1000, "Bob": 500}


def deposit(name, amount):
    if name in accounts:
        accounts[name] += amount


def withdraw(name, amount):
    if name in accounts and accounts[name] >= amount:
        accounts[name] -= amount


deposit("Alice", 200)
withdraw("Bob", 100)
print(accounts)  # {'Alice': 1200, 'Bob': 400}
```

This approach works for small programs, but as complexity increases, managing data and logic separately becomes challenging. Functions need to keep track of global state, which can lead to errors, unintended modifications, and difficulty maintaining the code.

OOP addresses these issues by **encapsulating data and behavior within objects**, making it easier to reason about and modify the program.

The Core Idea of OOP

In OOP, everything revolves around **objects**—self-contained units that bundle **state** (data) and **behavior** (methods). These objects are instances of **classes**, which define a blueprint for creating similar objects. Instead of handling data and functions separately, OOP allows for a more natural representation of entities.

Here's how the same bank account functionality looks in an OOP approach:

Object-Oriented approach

```
class BankAccount:
    def __init__(self, owner, balance=0):
```

```
        self.owner = owner
        self.balance = balance

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

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

# Creating instances of the class
alice_account = BankAccount("Alice", 1000)
bob_account = BankAccount("Bob", 500)

alice_account.deposit(200)
bob_account.withdraw(100)

print(alice_account.balance)  # 1200
print(bob_account.balance)    # 400
```

By wrapping data (balance) and related functionality (deposit and withdraw) into a **BankAccount** class, each object (account) manages its own state, preventing global state modifications and making the system easier to scale.

Benefits of Object-Oriented Programming

OOP provides several advantages that make it an ideal choice for many software applications:

- **Modularity:** Since data and functions are grouped into classes, the code is more organized and easier to manage.
- **Code Reusability:** By defining a class, multiple objects can be created without rewriting the same logic, reducing redundancy.
- **Scalability:** Large systems can be built by composing smaller, reusable objects, making it easier to extend functionality.
- **Encapsulation:** Internal implementation details are hidden within objects, preventing unintended interference and enforcing cleaner code design.

- **Flexibility with Inheritance and Polymorphism:** New classes can inherit properties and behaviors from existing ones, promoting code reuse and reducing duplication.

How Python Supports OOP

Python is a multi-paradigm language, meaning it supports both procedural and object-oriented styles of programming. Unlike some languages like Java or C++, where OOP is mandatory, Python allows developers to choose whether to use OOP, procedural, or even functional programming paradigms depending on the problem at hand.

Python makes OOP particularly **flexible and intuitive** by:

- Allowing dynamic typing, meaning objects don't require explicit type declarations.
- Supporting multiple inheritance, enabling a class to derive from more than one parent class.
- Providing special methods (also called **dunder methods**, like __init__ and __str__) to customize object behavior.
- Enabling **duck typing**, which allows polymorphism without strict type enforcement—if an object behaves like a type, it is treated as one. If it looks like a duck, swims like a duck, and quacks like a duck, then it probably *is* a duck.

Because of these features, Python is widely used in various fields, including **web development (Django, Flask), data science (Pandas, NumPy), and game development (Pygame)**, all of which leverage OOP principles.

Summary

Object-Oriented Programming offers a powerful way to structure and organize code by modeling real-world entities as objects. Unlike procedural programming, where data and functions are separate, OOP encapsulates behavior within objects, making programs more modular, reusable, and maintainable. Python's flexible OOP support allows developers to adopt OOP where it fits naturally, making it an excellent choice for building scalable and efficient applications.

Classes and Objects

At the heart of Object-Oriented Programming (OOP) are **classes** and **objects**. A **class** serves as a blueprint for creating objects, while **objects** are specific instances of a class, each with its own state and behavior. This section explores these concepts in detail and demonstrates how they work in Python.

What is a Class?

A **class** defines a template for creating objects. It specifies the properties (also called **attributes**) and behaviors (**methods**) that objects of that class will have. Think of a class as a blueprint for constructing real-world entities.

For example, consider a **Car** class. A car has attributes like color, brand, and speed, and it has behaviors like starting, accelerating, and stopping. Instead of defining individual cars manually, we can define a **Car** class once and create multiple car objects from it.

In Python, a class is defined using the class keyword:

```
class Car:
    def __init__(self, brand, color, speed=0):
        self.brand = brand  # Attribute
        self.color = color  # Attribute
        self.speed = speed  # Attribute

    def accelerate(self, increase):
        self.speed += increase  # Behavior
        print(f"{self.brand} is now going at {self.speed} km/h.")

    def brake(self):
        self.speed = 0  # Behavior
        print(f"{self.brand} has stopped.")
```

What is an Object?

An **object** is an instance of a class. When we create an object, we are bringing the blueprint to life by assigning specific values to its attributes. Each object maintains its own state and can perform actions defined by the class methods.

To create an object in Python, we call the class name as if it were a function:

```
# Creating objects (instances) of the Car class
car1 = Car("Toyota", "Red")
car2 = Car("BMW", "Blue", 50)

# Accessing attributes
print(car1.brand)  # Toyota
print(car2.color)  # Blue

# Calling methods
car1.accelerate(30)  # Toyota is now going at 30 km/h.
car2.brake()         # BMW has stopped.
```

Each object has its own **state** (values for brand, color, and speed). Even though both car1 and car2 are instances of the **Car** class, they operate independently.

Understanding __init__ and self

The __init__ method is a special function, also known as a **constructor**, that gets called when an object is created. It initializes the attributes of an object with specific values.

The self parameter refers to the current instance of the class and allows us to access its attributes and methods. It must always be the first parameter of instance methods.

For example, in Car.__init__, self.brand = brand assigns the value of brand to the object's brand attribute, making it unique for each instance.

Summary

A **class** is a blueprint that defines attributes and methods, while an **object** is an instance of a class with its own specific values and behaviors. Python allows us to define and instantiate classes easily, making it simple to model real-world entities in a structured and reusable way.

Would you like a more complex example or further explanations on any part?

Attributes and Methods

In Object-Oriented Programming, **attributes** define the state of an object, while **methods** define its behavior. Python provides flexibility in defining different types of attributes and methods to structure code efficiently. This section explores the distinction between **instance attributes** and **class attributes**, explains the role of **instance methods**, and introduces the self parameter.

Instance Attributes vs. Class Attributes

An **instance attribute** is unique to each object of a class. It is defined inside the __init__ method and assigned using self, ensuring that each instance maintains its own independent copy of the attribute.

A **class attribute**, on the other hand, is shared across all instances of a class. It is defined outside the __init__ method and belongs to the class itself rather than any single object.

Consider the following example:

```
class Animal:
    species = "Mammal"  # Class attribute (shared by all instances)

    def __init__(self, name, age):
        self.name = name  # Instance attribute (unique to each
instance)
        self.age = age    # Instance attribute

# Creating objects
dog = Animal("Buddy", 5)
cat = Animal("Whiskers", 3)
```

```
# Accessing attributes
print(dog.name)      # Buddy (unique to this instance)
print(cat.name)      # Whiskers (unique to this instance)
print(dog.species)   # Mammal (shared attribute)
print(cat.species)   # Mammal (shared attribute)

# Modifying class attribute (affects all instances)
Animal.species = "Reptile"
print(dog.species)   # Reptile
print(cat.species)   # Reptile
```

In this example, species is a class attribute that is shared among all instances of Animal, while name and age are instance attributes unique to each object.

Instance Methods and Their Role

Instance methods allow objects to interact with and modify their attributes. They are defined inside a class and always take self as the first parameter, which represents the current instance. These methods operate on instance attributes and can update the object's state.

Consider this example:

```
class Car:
    def __init__(self, brand, speed=0):
        self.brand = brand  # Instance attribute
        self.speed = speed  # Instance attribute

    def accelerate(self, increase):
        self.speed += increase  # Modifying the instance attribute
        print(f"{self.brand} is now going at {self.speed} km/h.")

    def brake(self):
        self.speed = 0  # Resetting the instance attribute
        print(f"{self.brand} has stopped.")

# Creating objects
car1 = Car("Toyota")
car2 = Car("BMW", 50)
```

```
# Using instance methods
car1.accelerate(30)  # Toyota is now going at 30 km/h.
car2.brake()         # BMW has stopped.
```

Here, accelerate and brake are instance methods that modify the object's speed attribute. Each object maintains its own state, and changes to one instance do not affect another.

Understanding self and Its Significance

The self parameter is a reference to the instance that calls a method. It allows each object to access and modify its own attributes and ensures that methods operate on the correct instance. Without self, Python would not know which object's attributes to access.

For example, consider what happens if we remove self:

```
class Example:
    def set_value(value):  # Missing self
        self.value = value  # Error: self is not defined

obj = Example()
obj.set_value(10)  # TypeError: set_value() missing 1 required positional argument
```

Python raises an error because the method does not explicitly take self. To fix this, we must include self as the first parameter:

class Example:

```
    def set_value(self, value):
        self.value = value  # Correct: self allows attribute assignment

obj = Example()
obj.set_value(10)
print(obj.value)  # 10
```

Summary

- **Instance attributes** are unique to each object, while **class attributes** are shared among all instances.
- **Instance methods** modify an object's state and always require self as the first parameter.
- The self parameter allows instance methods to interact with the correct object's attributes.
- Python enforces the use of self to distinguish between instance and class scope.

Would you like to see more advanced examples or variations of attributes and methods?

Encapsulation and Data Hiding

Encapsulation is one of the core principles of Object-Oriented Programming (OOP). It refers to the bundling of data (attributes) and methods (functions that operate on data) within a single unit, typically a class. Encapsulation also involves restricting direct access to certain attributes to ensure better control over how data is modified.

By hiding internal details and providing controlled access through methods, encapsulation improves **data security, maintainability, and flexibility** in software design.

Why is Encapsulation Important?

Without encapsulation, an object's attributes would be freely accessible and modifiable from anywhere in the program. This can lead to unintended changes, making debugging difficult and breaking program consistency. Encapsulation prevents direct modification of an object's internal state, ensuring that data remains **valid** and **consistent** by enforcing controlled access through methods.

For example, consider a **BankAccount** class. Without encapsulation, an account's balance could be changed arbitrarily:

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance  # No restriction on direct access

# Creating an object
account = BankAccount("Alice", 1000)

# Directly modifying balance (Unsafe!)
account.balance = -500  # This should not be allowed

print(account.balance)  # -500 (Invalid state)
```

This issue can be avoided using **access modifiers** and **getter/setter methods**.

Access Modifiers in Python

Unlike languages like Java or C++, Python does not enforce strict access control but follows a **convention-based approach** using special naming patterns.

Public Attributes (No Underscore)

By default, all attributes in Python are **public**, meaning they can be accessed and modified freely:

```
class Car:
```

```
    def __init__(self, brand, speed):
        self.brand = brand  # Public attribute
        self.speed = speed  # Public attribute

# Creating an object
car = Car("Toyota", 120)

# Accessing and modifying public attributes
print(car.brand)  # Toyota
car.speed = 150   # Allowed (but might not be desirable)
print(car.speed)  # 150
```

Since there's no restriction, any part of the code can modify speed, potentially leading to inconsistent behavior.

Protected Attributes (_underscore)

A **protected** attribute is conventionally indicated by a **single underscore (_attribute)**. It suggests that the attribute should not be accessed directly but **can still be accessed if necessary**.

```
class Car:
    def __init__(self, brand, speed):
        self.brand = brand
        self._speed = speed  # Protected attribute

    def get_speed(self):
        return self._speed  # Controlled access

# Creating an object
car = Car("BMW", 100)

# Accessing protected attribute (not recommended but possible)
print(car._speed)  # 100

# Using the getter method (preferred)
print(car.get_speed())  # 100
```

Although Python does not enforce protection, the underscore serves as a **warning** that _speed is meant for internal use.

Private Attributes (__double_underscore)

A **private** attribute is indicated by a **double underscore (__attribute)**, which **prevents direct access** from outside the class. Python applies **name mangling**, making it harder to modify private attributes directly.

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

    def get_balance(self):  # Getter method
        return self.__balance

    def deposit(self, amount):  # Controlled modification
        if amount > 0:
            self.__balance += amount
        else:
            print("Invalid deposit amount")

# Creating an object
account = BankAccount("Alice", 2000)

# Attempting direct access (fails)
# print(account.__balance)  # AttributeError

# Accessing via a method (allowed)
print(account.get_balance())  # 2000
```

Trying to access __balance directly results in an **AttributeError** because Python internally renames the attribute to _BankAccount__balance (name mangling). However, we can still access it using:

print(account._BankAccount__balance) # 2000 (Not recommended!)

While this workaround exists, it is **strongly discouraged**, as it breaks encapsulation principles.

Using Getters and Setters for Encapsulation

Encapsulation is typically implemented using **getter and setter methods**, which provide controlled access to private attributes.

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

    def get_balance(self):  # Getter method
        return self.__balance

    def set_balance(self, amount):  # Setter method
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid balance amount")

# Creating an object
account = BankAccount("Bob", 5000)

# Accessing and modifying balance using methods
print(account.get_balance())  # 5000
account.set_balance(3000)  # Balance updated
print(account.get_balance())  # 3000
account.set_balance(-1000)  # Invalid balance amount
```

The getter method **retrieves** the balance, while the setter method **modifies** it only under valid conditions. This ensures data integrity.

Using @property for Encapsulation

Python provides a more Pythonic way of defining getters and setters using the @property decorator.

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
```

```
        self.__balance = balance  # Private attribute

    @property
    def balance(self):  # Getter method
        return self.__balance

    @balance.setter
    def balance(self, amount):  # Setter method
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid balance amount")

# Creating an object
account = BankAccount("Charlie", 4000)

# Accessing the balance as if it's a public attribute
print(account.balance)  # 4000

# Updating balance using property setter
account.balance = 2500
print(account.balance)  # 2500

# Trying to set an invalid balance
account.balance = -1000  # Invalid balance amount
```

The @property decorator allows balance to be accessed like a regular attribute while still enforcing encapsulation behind the scenes.

Summary

Encapsulation ensures better control over an object's data by restricting direct access and enforcing controlled modification. Python follows convention-based access control:

- **Public attributes** (no underscore) can be accessed and modified freely.
- **Protected attributes** (_attribute) signal that they should not be modified directly.
- **Private attributes** (__attribute) enforce data hiding using **name mangling**.

- **Getters and setters** provide controlled access to private attributes, ensuring data integrity.
- The @property decorator provides a Pythonic way to manage attribute access while maintaining encapsulation.

Encapsulation is key to **maintainability, security, and modularity**, making software more robust and less prone to unintended modifications.

Would you like additional real-world examples or deeper exploration of any part?

Inheritance and Code Reusability

One of the key advantages of Object-Oriented Programming (OOP) is **inheritance**, which allows a class to derive properties and behaviors from another class. This promotes **code reuse, modularity, and scalability**, reducing redundancy by enabling classes to share common functionality.

Inheritance enables **hierarchical relationships** between classes, where a **base class (parent class)** provides fundamental behavior, and a **derived class (child class)** extends or modifies it as needed.

Understanding Inheritance

Inheritance allows a new class to acquire the attributes and methods of an existing class. This eliminates the need to rewrite common functionality, as the child class automatically inherits everything from the parent class while having the flexibility to define its own unique features.

For example, consider a **general Animal class** that provides basic attributes like name and sound. A **Dog class** can inherit from Animal while introducing additional behavior.

```
# Base (Parent) Class
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        return "Some generic sound"

# Derived (Child) Class
class Dog(Animal):  # Dog inherits from Animal
    def make_sound(self):  # Overriding parent method
        return "Bark"

# Creating objects
generic_animal = Animal("Unknown")
dog = Dog("Buddy")
```

print(generic_animal.make_sound()) # Some generic sound

print(dog.make_sound()) # Bark

Here, Dog inherits from Animal, gaining the name attribute and the make_sound method. However, the make_sound method is **overridden** in Dog to provide specific behavior.

Types of Inheritance in Python

Python supports different types of inheritance, including **single inheritance, multiple inheritance, multilevel inheritance, and hierarchical inheritance**.

Single Inheritance

A single derived class inherits from one base class.

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def display_brand(self):
        print(f"Brand: {self.brand}")

class Car(Vehicle):  # Single inheritance
    def __init__(self, brand, model):
        super().__init__(brand)  # Call parent constructor
        self.model = model

    def display_info(self):
        print(f"Car Model: {self.model}")

# Creating an object
my_car = Car("Toyota", "Corolla")
my_car.display_brand()  # Brand: Toyota
my_car.display_info()   # Car Model: Corolla
```

The Car class inherits from Vehicle, meaning it automatically gets the display_brand method while also adding its own behavior.

Multiple Inheritance

A child class can inherit from more than one parent class, gaining functionality from multiple sources.

```
class Engine:
    def engine_type(self):
        return "V8 Engine"

class Wheels:
    def wheel_count(self):
        return 4

class Car(Engine, Wheels):  # Multiple inheritance
    def __init__(self, brand):
        self.brand = brand

    def display_info(self):
        print(f"{self.brand} with {self.engine_type()} and {self.wheel_count()} wheels.")

# Creating an object
my_car = Car("Ford Mustang")
my_car.display_info()  # Ford Mustang with V8 Engine and 4 wheels.
```

Here, Car inherits from both Engine and Wheels, gaining access to methods from both parent classes.

Multilevel Inheritance

A class can be derived from another derived class, forming a **chain of inheritance**.

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

class Mammal(Animal):
    def has_fur(self):
        return True
```

```
class Dog(Mammal):  # Multilevel inheritance
    def speak(self):
        return "Bark"

# Creating an object
dog = Dog()
print(dog.speak())  # Bark (Overridden method)
print(dog.has_fur())  # True (Inherited from Mammal)
```

Here, Dog inherits from Mammal, which itself inherits from Animal. This allows Dog to access methods from both parent classes.

Hierarchical Inheritance

Multiple child classes inherit from the same parent class.

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

class Dog(Animal):  # Derived from Animal
    def speak(self):
        return "Bark"

class Cat(Animal):  # Derived from Animal
    def speak(self):
        return "Meow"

dog = Dog()
cat = Cat()

print(dog.speak())  # Bark
print(cat.speak())  # Meow
```

Both Dog and Cat inherit from Animal, but each provides its own implementation of speak.

Method Overriding in Inheritance

Overriding allows a child class to provide a specific implementation of a method that is already defined in the parent class.

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

class Dog(Animal):
    def speak(self):  # Overriding the parent method
        return "Bark"

dog = Dog()
print(dog.speak())  # Bark (Overrides parent's method)
```

The speak method is overridden in the Dog class to provide specialized behavior.

Using super() to Call Parent Class Methods

The super() function allows a derived class to access methods from its parent class, making it useful for extending functionality rather than replacing it completely.

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def description(self):
        return f"Brand: {self.brand}"

class Car(Vehicle):
    def __init__(self, brand, model):
        super().__init__(brand)  # Call parent constructor
        self.model = model

    def description(self):
        return f"{super().description()}, Model: {self.model}"  # Extend parent method

# Creating an object
car = Car("Tesla", "Model S")
print(car.description())  # Brand: Tesla, Model: Model S
```

The super() function:

1. Calls the __init__ method of the parent class to initialize common attributes.
2. Extends the description() method instead of completely overriding it.

Summary

Inheritance enables code reuse by allowing a child class to acquire properties and behaviors from a parent class. Python supports:

- **Single inheritance** (one parent, one child).
- **Multiple inheritance** (a child inheriting from multiple parents).
- **Multilevel inheritance** (a chain of inheritance).
- **Hierarchical inheritance** (multiple child classes inheriting from the same parent).
- **Method overriding** allows child classes to redefine inherited methods.
- The super() function helps call parent class methods, ensuring that shared functionality is retained.

Inheritance simplifies code, **reduces duplication, and improves maintainability**, making it easier to extend existing systems without modifying original implementations.

Would you like to explore any advanced inheritance concepts, such as mixins or method resolution order (MRO)?

Polymorphism and Method Overriding

Polymorphism is one of the fundamental principles of Object-Oriented Programming (OOP). It allows objects of different classes to be treated uniformly, enabling a more flexible and extensible design. In Python, polymorphism is mainly achieved through **method overriding** and **duck typing**.

What is Polymorphism?

Polymorphism (meaning "many forms") allows different classes to define methods with the same name but different implementations. This enables a function or method to interact with different object types without needing to know their specific class.

For example, if multiple classes have a speak() method, we can call speak() on any object without worrying about its class.

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


class Cat:
    def speak(self):
        return "Meow"
```

```
# Function that works with any object having a speak() method
def make_animal_speak(animal):
    print(animal.speak())

# Creating objects
dog = Dog()
cat = Cat()

# Calling the function with different objects
make_animal_speak(dog)  # Bark
make_animal_speak(cat)  # Meow
```

Here, make_animal_speak() works with any object that implements speak(), demonstrating **polymorphism**.

Method Overriding in Polymorphism

Method overriding occurs when a **child class provides a different implementation** of a method that already exists in the parent class. This allows the child class to **customize behavior** while maintaining a consistent interface.

Consider an example with a Vehicle base class and two derived classes, Car and Bike:

```
class Vehicle:
    def move(self):
        return "Vehicle is moving"

class Car(Vehicle):
    def move(self):  # Overriding parent method
        return "Car is driving on the road"

class Bike(Vehicle):
    def move(self):  # Overriding parent method
        return "Bike is cycling on the path"

# Using polymorphism
vehicles = [Car(), Bike(), Vehicle()]
```

```
for v in vehicles:
    print(v.move())

# Output:
# Car is driving on the road
# Bike is cycling on the path
# Vehicle is moving
```

Here, the move() method is overridden in both Car and Bike, but a common interface (move()) ensures that we can loop through different vehicles and call move() without checking their types.

Duck Typing in Python and Polymorphism

Python follows a **duck typing** philosophy, which means **an object's type is determined by its behavior, not its inheritance**. If an object implements a method, it can be used regardless of its class.

This is famously captured by the phrase:
"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

Unlike statically typed languages where polymorphism is achieved through explicit interfaces or inheritance, Python allows polymorphism based on method presence.

For example:

```
class Bird:
    def fly(self):
        return "Bird is flying"

class Airplane:
    def fly(self):
        return "Airplane is soaring through the sky"

class Fish:
    def swim(self):
        return "Fish is swimming"

# Function that works with any object having a fly() method
def take_off(flying_object):
    print(flying_object.fly())
```

```
# Creating objects
bird = Bird()
airplane = Airplane()

take_off(bird)       # Bird is flying
take_off(airplane)   # Airplane is soaring through the sky

# take_off(Fish())  # Error! Fish does not have a fly() method
```

Here, take_off() works with both Bird and Airplane because they implement fly(), even though they are unrelated classes. However, if we try to pass a Fish object, Python raises an error because it lacks a fly() method.

Advantages of Polymorphism and Method Overriding

- **Code Reusability**: Functions and methods can operate on different types without modification.
- **Extensibility**: New classes can be introduced without changing existing code.
- **Better Maintainability**: Code is cleaner and avoids unnecessary conditionals (if-else statements to check types).

Summary

Polymorphism enables different classes to provide their own implementation of a shared method, making code more flexible and reusable.

- **Method Overriding** allows a child class to redefine a parent class method for specialized behavior.
- **Duck Typing** in Python means an object's usability is determined by its behavior (methods) rather than explicit inheritance.
- Python's dynamic nature makes polymorphism easy to implement, improving **code maintainability and readability**.

Special Methods and Operator Overloading

Python provides a set of special methods, also known as **dunder methods** (short for "double underscore"), which allow objects to integrate seamlessly with built-in functions and operators. These methods begin and end with double underscores (__), such as __init__, __str__, __len__, and __eq__.

One of the most powerful applications of dunder methods is **operator overloading**, which allows objects to interact with standard operators like +, -, *, ==, and more.

Understanding Special Methods (Dunder Methods)

Special methods define how objects of a class behave when used with Python's built-in features. Here are some commonly used dunder methods:

__init__: Object Initialization (Constructor)

The __init__ method initializes an object's attributes when it is created.

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

# Creating an object
p = Person("Alice", 30)
print(p.name)  # Alice
print(p.age)   # 30
```

__str__: String Representation (str())

The __str__ method defines how an object is represented as a string when print() or str() is called.

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

    def __str__(self):
        return f"Person(Name: {self.name}, Age: {self.age})"

# Creating an object
p = Person("Bob", 25)
print(p)  # Person(Name: Bob, Age: 25)
```

Without __str__, print(p) would output something like <__main__.Person object at 0x...>, which is not user-friendly.

__len__: Object Length (len())

The __len__ method allows objects to return a length when len() is called.

```
class Book:
```

```
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __len__(self):
        return self.pages

# Creating an object
b = Book("Python Essentials", 350)
print(len(b))  # 350
```

Now, calling len() on a Book object returns the number of pages.

__eq__: Equality Comparison (==)

The __eq__ method defines object equality, allowing objects to be compared using ==.

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

    def __eq__(self, other):
        return self.name == other.name and self.age == other.age

# Creating objects
p1 = Person("Alice", 30)
p2 = Person("Alice", 30)
p3 = Person("Bob", 25)

print(p1 == p2)  # True (same name and age)
print(p1 == p3)  # False (different attributes)
```

Without __eq__, == would compare object memory addresses rather than their attribute values.

Operator Overloading

Operator overloading allows us to redefine standard operators (+, -, *, ==, >, etc.) for custom objects. Python calls the appropriate dunder method when an operator is used on an object.

Overloading the + Operator (__add__)

We can define how objects behave when they are added using + by implementing the __add__ method.

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

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

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

# Creating objects
v1 = Vector(3, 4)
v2 = Vector(1, 2)

# Adding two vectors
result = v1 + v2
print(result)  # Vector(4, 6)
```

Here, v1 + v2 calls v1.__add__(v2), returning a new Vector object with the sum of the coordinates.

Overloading the * Operator (__mul__)

We can define how an object behaves with the * operator.

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

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

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

# Creating an object
v = Vector(2, 3)

# Scaling a vector
scaled_v = v * 3
print(scaled_v)  # Vector(6, 9)
```

Here, v * 3 calls v.__mul__(3), multiplying both components by 3.

Overloading Comparison Operators (>, <, >=, <=)

We can define how objects are compared using >, <, >=, <= by implementing __gt__, __lt__, __ge__, and __le__.

```
class Box:
    def __init__(self, volume):
        self.volume = volume

    def __gt__(self, other):  # Greater than
        return self.volume > other.volume

    def __lt__(self, other):  # Less than
        return self.volume < other.volume

# Creating objects
box1 = Box(100)
box2 = Box(50)

print(box1 > box2)  # True
print(box1 < box2)  # False
```

Now, we can compare Box objects based on their volume.

Overview

In Python, **a protocol** refers to a set of special methods that, when implemented by a class, allow its instances to behave like built-in types or be compatible with certain language features. The term comes from the idea that classes can follow certain *contracts* (or *interfaces*) without explicit inheritance.

Here are the main **protocols** (i.e., sets of special methods) defined by Python:

Object Construction Protocol

- __new__, __init__
- Allows custom creation and initialization of objects.

String Representation Protocol

- __str__, __repr__, __format__
- Used by str(), repr(), and format().

Attribute Access Protocol

- __getattr__, __getattribute__, __setattr__, __delattr__
- Controls how attributes are accessed and assigned.

Callable Protocol

- __call__
- Makes an instance behave like a function.

Context Management Protocol

- __enter__, __exit__
- Enables use in with statements.

Iterator Protocol

- __iter__, __next__
- Enables iteration in for loops and with next().

Container and Sequence Protocols

- __len__, __getitem__, __setitem__, __delitem__, __contains__
- Make objects behave like sequences or mappings (lists, dicts, etc.).

Arithmetic Protocol

- __add__, __sub__, __mul__, etc.
- __radd__, __iadd__, etc.

- Implements arithmetic operations and in-place variants.

Comparison and Hashing Protocol

- __eq__, __ne__, __lt__, __le__, __gt__, __ge__
- __hash__
- Used by comparison operators and for hashable containers like set and dict.

Each protocol allows your class to "opt in" to a specific kind of behavior. For instance, if you implement __iter__ and __next__, your class is an *iterator* by protocol—even if it doesn't inherit from a specific base class. This is core to Python's *duck typing* philosophy.

Summary

- **Dunder methods** customize object behavior in Python.
- __init__, __str__, __len__, and __eq__ are common special methods.
- **Operator overloading** allows objects to use standard operators (+, *, ==, etc.).
- Python's **flexible magic methods** make it easy to integrate user-defined classes with built-in Python features.

Class and Static Methods

In Python, methods within a class can be categorized into three types: **instance methods, class methods, and static methods**. Each serves a distinct purpose and determines how a method interacts with the class and its instances.

This section explores the differences between these methods, explains when to use them, and provides practical examples.

Instance Methods vs. Class Methods vs. Static Methods

Method Type	Decorator	First Parameter	Accesses Instance Attributes?	Accesses Class Attributes?
Instance Method	None	self	✅ Yes	✅ Yes
Class Method	@classmethod	cls	❌ No	✅ Yes
Static Method	@staticmethod	None	❌ No	❌ No

- **Instance methods** operate on individual objects and can modify instance attributes.
- **Class methods** operate at the class level and affect all instances.
- **Static methods** are independent utility functions that belong to a class but do not modify class or instance attributes.

Instance Methods (self)

Instance methods are the most common type of method in a class. They take self as their first parameter, allowing them to access and modify attributes of an instance.

```
class Car:
    def __init__(self, brand, speed=0):
        self.brand = brand
        self.speed = speed  # Instance attribute

    def accelerate(self, amount):
        self.speed += amount
        print(f"{self.brand} is now going at {self.speed} km/h.")

# Creating an instance
car1 = Car("Toyota")
car1.accelerate(20)  # Toyota is now going at 20 km/h.
```

Here, accelerate() modifies the speed of a specific car instance.

Class Methods (@classmethod) and the cls Parameter

Class methods operate on the class itself rather than on individual instances. They use cls (short for "class") as their first parameter, which refers to the class rather than an instance.

Class methods are commonly used when we need to **modify class attributes** or create instances in an alternative way.

Example: Using @classmethod to Modify a Class Attribute

```
class Car:
    count = 0  # Class attribute (shared across instances)

    def __init__(self, brand):
        self.brand = brand
        Car.count += 1  # Increment count for each new instance

    @classmethod
    def total_cars(cls):  # cls refers to the class
        print(f"Total cars created: {cls.count}")
```

```
# Creating instances
car1 = Car("Tesla")
car2 = Car("BMW")

# Calling class method
Car.total_cars()  # Total cars created: 2
```

Here, total_cars() is a class method that operates on cls.count, a class attribute shared by all instances.

Example: Using @classmethod as an Alternative Constructor

Class methods can also be used as alternative constructors to create instances in different ways.

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

    @classmethod
    def from_birth_year(cls, name, birth_year):
        current_year = 2025
        age = current_year - birth_year
        return cls(name, age)  # Calls the __init__ method

# Creating an instance using the normal constructor
p1 = Person("Alice", 30)

# Creating an instance using the alternative constructor
p2 = Person.from_birth_year("Bob", 1995)

print(p2.name, p2.age)  # Bob 30
```

Here, from_birth_year() provides an alternative way to instantiate a Person object.

Static Methods (@staticmethod)

Static methods are independent functions within a class. They do **not** take self or cls as a parameter because they do not modify instance or class attributes.

Static methods are useful for utility functions that are **related to the class but do not require class-specific data**.

Example: Using @staticmethod for Utility Functions

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

# Calling static methods
print(MathOperations.add(5, 3))  # 8
print(MathOperations.multiply(4, 2))  # 8
```

Here, add() and multiply() are general utility functions placed inside a class for organizational purposes.

When to Use Each Method Type?

Method Type	Use When...
Instance Method	You need to access or modify instance attributes (self).
Class Method	You need to modify or access class attributes (cls).
Class Method (Alt Constructor)	You want to provide an alternative way to instantiate objects.
Static Method	You need a helper function that does not access instance or class attributes.

Example: Combining All Three Method Types

To see instance, class, and static methods working together, consider a BankAccount class:

```
class BankAccount:
    interest_rate = 0.03  # Class attribute (shared)

    def __init__(self, owner, balance):
        self.owner = owner
```

```
        self.balance = balance  # Instance attribute

    def deposit(self, amount):  # Instance method
        self.balance += amount
        print(f"{self.owner} deposited {amount}. New balance:
{self.balance}")

    @classmethod
    def set_interest_rate(cls, rate):  # Class method
        cls.interest_rate = rate
        print(f"Updated interest rate: {cls.interest_rate}")

    @staticmethod
    def bank_policy():  # Static method
        print("Bank policy: Maintain a minimum balance of $100.")

# Creating an object
account = BankAccount("Alice", 1000)

# Calling instance method
account.deposit(500)  # Alice deposited 500. New balance: 1500

# Calling class method
BankAccount.set_interest_rate(0.05)  # Updated interest rate: 0.05

# Calling static method
BankAccount.bank_policy()  # Bank policy: Maintain a minimum balance
of $100.
```

- deposit() modifies the balance for a specific account (instance method).
- set_interest_rate() modifies a class-level attribute interest_rate (class method).
- bank_policy() provides a general rule unrelated to instances or class attributes (static method).

Summary

- **Instance Methods (self)** operate on individual objects and modify instance attributes.
- **Class Methods (@classmethod, cls)** operate on the class itself and can modify class attributes.
- **Static Methods (@staticmethod)** are utility functions that do not modify class or instance attributes.
- Class methods are useful for alternative constructors and modifying shared attributes.
- Static methods help organize related utility functions within a class.

Abstract Classes and Interfaces

In Object-Oriented Programming (OOP), **abstract classes** and **interfaces** define a blueprint for other classes, enforcing structure and consistency across different implementations. Python provides support for abstraction through the abc (**Abstract Base Class**) module, which allows the creation of abstract classes and methods that must be implemented by subclasses.

What is an Abstract Class?

An **abstract class** is a class that **cannot be instantiated** on its own. It may contain **abstract methods**—methods that have no implementation in the base class but must be implemented by any subclass. This ensures that all derived classes follow a consistent structure.

Abstract classes are useful when designing a **common interface** for a group of related classes. They promote **code organization, enforce method consistency, and improve maintainability**.

Key Features of Abstract Classes

- Cannot be instantiated directly.
- Can have both **abstract methods** (which must be implemented by subclasses) and **concrete methods** (which provide default behavior).
- Serve as a blueprint for child classes, ensuring they implement required functionality.

The abc Module and ABC Class

Python's abc module (short for **Abstract Base Class**) provides tools to create abstract classes and enforce method implementation.

To define an abstract class:

1. Inherit from ABC (provided by the abc module).
2. Use the @abstractmethod decorator to mark methods that **must** be implemented by subclasses.

Example: Creating an Abstract Class

```
from abc import ABC, abstractmethod
```

```
class Animal(ABC):  # Abstract Base Class
    @abstractmethod
    def make_sound(self):  # Abstract method (must be implemented by subclasses)
        pass

    def sleep(self):  # Concrete method (default implementation)
        print("Sleeping...")

# Attempting to instantiate Animal will raise an error
# animal = Animal()  # TypeError: Can't instantiate abstract class Animal

class Dog(Animal):
    def make_sound(self):
        return "Bark"

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

# Creating instances
dog = Dog()
cat = Cat()

print(dog.make_sound())  # Bark
print(cat.make_sound())  # Meow
dog.sleep()  # Sleeping...
```

In this example:

- Animal is an **abstract class** with an abstract method make_sound().
- Dog and Cat **inherit from Animal** and implement make_sound(), making them **concrete classes**.
- The sleep() method is a **concrete method** that provides default behavior and is inherited by all subclasses.

Why Use Abstract Classes?

1. **Enforcing a Structure**: Guarantees that all subclasses implement required methods.
2. **Code Reusability**: Abstract classes can define common functionality to be shared by subclasses.
3. **Prevents Instantiation**: Ensures that base classes are only used as blueprints, avoiding accidental object creation.

For example, if we define a Shape class, we want every shape (Circle, Rectangle, etc.) to have an area() method, but each shape will compute its area differently.

```
from abc import ABC, abstractmethod


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


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


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


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


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


# Creating instances
circle = Circle(5)
rectangle = Rectangle(4, 6)
```

```
print(circle.area())  # 78.5
print(rectangle.area())  # 24
```

Here:

- Shape enforces that all subclasses must define area().
- Circle and Rectangle implement area() in their own way.

Abstract Classes vs. Interfaces

An **interface** is a collection of abstract methods that define a contract for classes. Python does not have built-in interfaces like Java or C#, but abstract classes can serve the same purpose.

If we want a **pure interface** (without concrete methods), we can define an abstract class with only abstract methods:

```
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

class CreditCardPayment(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing credit card payment of ${amount}")

class PayPalPayment(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing PayPal payment of ${amount}")

# Creating instances
payment1 = CreditCardPayment()
payment2 = PayPalPayment()

payment1.process_payment(100)  # Processing credit card payment of $100
payment2.process_payment(50)   # Processing PayPal payment of $50
```

Here:

- PaymentProcessor acts as an **interface**, ensuring all payment methods have a process_payment() function.
- CreditCardPayment and PayPalPayment implement process_payment() in their own way.

Key Takeaways

- **Abstract classes** define a blueprint for derived classes but cannot be instantiated.
- The abc module provides ABC and @abstractmethod to enforce method implementation in subclasses.
- Abstract classes can include both **abstract methods** (which must be implemented) and **concrete methods** (which provide default behavior).
- Python does not have explicit interfaces, but abstract classes can act as interfaces by containing only abstract methods.
- Using abstract classes helps enforce **consistent design, prevent incomplete implementations, and promote reusability**.

Composition vs. Inheritance

In Object-Oriented Programming, **inheritance** and **composition** are two fundamental techniques for organizing and structuring code. While inheritance allows a class to derive behavior from a parent class, **composition involves creating objects within objects** to build modular and flexible relationships.

Understanding when to use **composition instead of inheritance** is key to writing maintainable, scalable, and reusable code.

What is Composition?

Composition is a design principle where one class contains an instance of another class as an attribute, instead of inheriting from it. This approach is often described as a **"has-a" relationship**, in contrast to inheritance, which represents an **"is-a" relationship**.

For example:

- A Car **has a** Engine.
- A Person **has a** Address.
- A Library **has a** collection of Books.

Composition allows objects to be **assembled dynamically**, making them more flexible than rigid class hierarchies.

Inheritance vs. Composition

Feature	Inheritance ("is-a")	Composition ("has-a")
Definition	A class derives from another class.	A class contains an instance of another class.
Flexibility	Less flexible, as changes in the base class affect all derived classes.	More flexible, as components can be swapped or replaced easily.
Code Reuse	Methods and attributes are inherited, reducing redundancy.	Encourages modular design by combining independent objects.
Extensibility	Can become complex with deep hierarchies.	Avoids deep hierarchies by focusing on small, interchangeable parts.
Example	Dog **is a** Animal.	Car **has a** Engine.

When to Prefer Composition Over Inheritance

While inheritance is useful for modeling relationships where one class **truly extends** another, composition is often the **better choice in these cases**:

1. **Avoiding Deep Inheritance Trees** – If classes become deeply nested (A → B → C → D), managing changes becomes difficult.
2. **Improving Flexibility** – Composition allows **objects to be replaced dynamically** without affecting the entire hierarchy.
3. **Code Reusability** – Composition enables **reusing components** across different classes without duplication.
4. **Encapsulation and Decoupling** – Changes in one component do not necessarily affect others, improving maintainability.

Example: Using Inheritance (Rigid Design)

Consider a scenario where we model a Car and a ElectricCar using **inheritance**.

```
class Car:
    def __init__(self, brand, fuel_type):
        self.brand = brand
        self.fuel_type = fuel_type

    def start(self):
        print(f"{self.brand} starts using {self.fuel_type}.")

class ElectricCar(Car):  # Inheriting from Car
```

```
    def __init__(self, brand, battery_capacity):
        super().__init__(brand, "Electric")
        self.battery_capacity = battery_capacity

    def charge(self):
        print(f"{self.brand} is charging with
{self.battery_capacity} kWh battery.")

# Creating instances
tesla = ElectricCar("Tesla", 100)
tesla.start()   # Tesla starts using Electric.
tesla.charge()  # Tesla is charging with 100 kWh battery.
```

While this works, if we later want to add **HybridCars**, **HydrogenCars**, or different battery types, the inheritance tree grows complex and becomes hard to maintain.

Example: Using Composition (Better Design)

Now, let's refactor the above example using **composition** instead of inheritance.

```
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type

    def start(self):
        print(f"Engine starts using {self.fuel_type}.")

class Battery:
    def __init__(self, capacity):
        self.capacity = capacity

    def charge(self):
        print(f"Battery is charging with {self.capacity} kWh
capacity.")

class Car:
    def __init__(self, brand, engine=None, battery=None):
```

```
        self.brand = brand
        self.engine = engine
        self.battery = battery

    def start(self):
        if self.engine:
            self.engine.start()
        else:
            print(f"{self.brand} has no engine.")

    def charge(self):
        if self.battery:
            self.battery.charge()
        else:
            print(f"{self.brand} has no battery.")

# Creating components
gas_engine = Engine("Gasoline")
electric_battery = Battery(100)

# Creating cars with different configurations
gas_car = Car("Toyota", engine=gas_engine)
electric_car = Car("Tesla", battery=electric_battery)

# Using composition
gas_car.start()       # Engine starts using Gasoline.
electric_car.charge() # Battery is charging with 100 kWh capacity.
```

Why is This Better?

- **More Flexible**: A car can have different combinations of engines and batteries without modifying the class.
- **Better Code Reuse**: Engine and Battery can be used in different contexts.
- **Easier to Maintain**: Adding new fuel types (e.g., **HydrogenEngine**) does not require changing the Car class.

Mixing Composition and Inheritance

In some cases, **a combination of both composition and inheritance** provides the best design. For example, using **inheritance for base behaviors** and **composition for components**.

```
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def info(self):
        print(f"Vehicle Brand: {self.brand}")

class Car(Vehicle):  # Inheriting basic vehicle properties
    def __init__(self, brand, engine=None):
        super().__init__(brand)
        self.engine = engine

    def start(self):
        if self.engine:
            self.engine.start()
        else:
            print(f"{self.brand} has no engine.")

# Creating an engine and using composition inside inheritance
diesel_engine = Engine("Diesel")
car = Car("Ford", diesel_engine)

car.info()   # Vehicle Brand: Ford
car.start()  # Engine starts using Diesel.
```

Here, Car **inherits from** Vehicle (common behavior) while also **using composition** for Engine.

Key Takeaways

- **Inheritance** is best when a class **is a specialized version** of another.
- **Composition** is better when a class **contains multiple independent parts**.
- Prefer **composition** to avoid deep inheritance trees and **increase modularity**.

- A combination of **inheritance + composition** provides an optimal design in some cases.

Design Principles in OOP

Good software design is essential for writing **maintainable, scalable, and efficient** programs. In Object-Oriented Programming (OOP), the **SOLID** principles provide a guideline for structuring code in a way that improves readability, flexibility, and reusability.

This section introduces the **SOLID principles** and explains key concepts like **Single Responsibility, Open-Closed, and Dependency Inversion Principles**, with Python examples.

Introduction to SOLID Principles

The **SOLID** principles are five design principles that help create well-structured object-oriented systems. The acronym stands for:

- **S**: **Single Responsibility Principle (SRP)**
- **O**: **Open-Closed Principle (OCP)**
- **L**: **Liskov Substitution Principle (LSP)**
- **I**: **Interface Segregation Principle (ISP)**
- **D**: **Dependency Inversion Principle (DIP)**

Applying these principles leads to **modular code**, reduces **tight coupling**, and makes software **easier to extend and maintain**.

Single Responsibility Principle (SRP)

"A class should have only one reason to change."

A class should have a **single responsibility**, meaning it should only perform **one well-defined task**. This makes the code **more maintainable and easier to test**.

Bad Example (Violating SRP)

Here, a class handles **both data storage and report generation**, which mixes responsibilities.

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

    def calculate_salary(self):
        return self.salary * 12  # Annual salary

    def generate_report(self):
```

```
        return f"Employee: {self.name}, Annual Salary:
{self.calculate_salary()}"
```

If the reporting format changes, this class must be modified, even though salary calculations remain the same.

Better Example (Following SRP)

Splitting responsibilities into separate classes:

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

    def calculate_salary(self):
        return self.salary * 12

class ReportGenerator:
    def generate_report(self, employee):
        return f"Employee: {employee.name}, Annual Salary:
{employee.calculate_salary()}"

# Creating instances
emp = Employee("Alice", 5000)
report = ReportGenerator()
print(report.generate_report(emp))  # Employee: Alice, Annual
Salary: 60000
```

Now:

- **Employee only handles employee data and salary calculations**.
- **ReportGenerator handles reporting**, keeping the responsibilities separate.

Open-Closed Principle (OCP)

"A class should be open for extension but closed for modification."

This means you should **extend functionality** without modifying existing code, reducing the risk of introducing bugs.

Bad Example (Violating OCP)

Adding new employee types requires **modifying the existing class**, making the code fragile.

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

    def calculate_bonus(self):
        if self.role == "Manager":
            return 1000
        elif self.role == "Developer":
            return 500
```

If new roles are added (e.g., "Intern"), we must **modify** the class.

Better Example (Following OCP)

Using **polymorphism** allows new behavior **without modifying** existing code:

```
from abc import ABC, abstractmethod

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

    @abstractmethod
    def calculate_bonus(self):
        pass

class Manager(Employee):
    def calculate_bonus(self):
        return 1000

class Developer(Employee):
    def calculate_bonus(self):
        return 500

# Adding a new role without modifying existing classes
```

```
class Intern(Employee):
    def calculate_bonus(self):
        return 200

# Creating instances
employees = [Manager("Alice"), Developer("Bob"), Intern("Charlie")]

for emp in employees:
    print(f"{emp.name}'s bonus: {emp.calculate_bonus()}")

# Output:
# Alice's bonus: 1000
# Bob's bonus: 500
# Charlie's bonus: 200
```

Now, **new roles** can be added by creating **new classes** instead of modifying existing code.

Dependency Inversion Principle (DIP)

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

Instead of **directly depending** on specific implementations, classes should depend on **interfaces or abstract classes**, making the system **more flexible**.

Bad Example (Violating DIP)

A PaymentProcessor class is tightly coupled to CreditCardPayment.

```
class CreditCardPayment:
    def pay(self, amount):
        print(f"Paid {amount} using Credit Card")

class PaymentProcessor:
    def __init__(self):
        self.payment_method = CreditCardPayment()  # Direct
dependency

    def process_payment(self, amount):
        self.payment_method.pay(amount)
```

What if we want to add **PayPal** payments? We'd have to modify PaymentProcessor, violating **OCP and DIP**.

Better Example (Following DIP)

Using **abstraction** allows us to switch payment methods without modifying PaymentProcessor.

```
from abc import ABC, abstractmethod


class PaymentMethod(ABC):  # Abstract base class
    @abstractmethod
    def pay(self, amount):
        pass


class CreditCardPayment(PaymentMethod):
    def pay(self, amount):
        print(f"Paid {amount} using Credit Card")


class PayPalPayment(PaymentMethod):
    def pay(self, amount):
        print(f"Paid {amount} using PayPal")


class PaymentProcessor:
    def __init__(self, payment_method: PaymentMethod):  # Dependency
Injection
        self.payment_method = payment_method


    def process_payment(self, amount):
        self.payment_method.pay(amount)


# Using different payment methods without modifying PaymentProcessor
credit_card = CreditCardPayment()
paypal = PayPalPayment()


processor1 = PaymentProcessor(credit_card)
processor1.process_payment(100)  # Paid 100 using Credit Card
```

```
processor2 = PaymentProcessor(paypal)
processor2.process_payment(200)  # Paid 200 using PayPal
```

Now, **new payment methods** can be added **without modifying** PaymentProcessor.

Common Pitfalls and Best Practices

While Object-Oriented Programming (OOP) provides a powerful way to structure code, it is easy to misuse its concepts, leading to poor maintainability and inefficient design. This section highlights common pitfalls in OOP and outlines best practices to follow for writing clean, scalable, and maintainable Python code.

Common Pitfalls

Improper Use of Inheritance

Pitfall: Inheriting when composition would be a better choice.
Many beginners overuse **inheritance**, creating deep, rigid class hierarchies that make the code difficult to maintain.

Bad Example (Unnecessary Inheritance)

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


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


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

Here, **RobotDog** is unnecessarily inheriting from **Dog**, even though a robot dog has little in common with a real dog. This leads to unnecessary dependencies.

Better Approach (Using Composition)

class Battery:

  def __init__(self, capacity):

```
        self.capacity = capacity


class RobotDog:
    def __init__(self, name, battery: Battery):
        self.name = name
        self.battery = battery
```

By **favoring composition**, we keep **RobotDog** independent from **Dog**, making it easier to maintain.

Overuse of Getters and Setters

Pitfall: Writing explicit getter and setter methods for every attribute, even when direct attribute access is sufficient.

Bad Example (Unnecessary Getters and Setters)

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

    def get_name(self):
        return self._name

    def set_name(self, name):
        self._name = name
```

Python allows **direct attribute access**, making explicit getters and setters **unnecessary** unless additional logic is required.

Better Approach (Using @property)

```
class Person:
    def __init__(self, name):
        self.name = name  # Direct access

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, new_name):
```

```
        if len(new_name) > 1:
            self._name = new_name
        else:
            raise ValueError("Name must be at least 2 characters
long")
```

Here, @property is used only when **validation is required**, keeping the code cleaner.

Deep Class Hierarchies

Pitfall: Creating **overly deep** inheritance trees makes code difficult to understand and modify.

class A: pass

class B(A): pass

class C(B): pass

class D(C): pass

class E(D): pass

If a change is required in **A**, it affects all child classes. Deep hierarchies make debugging and modification difficult.

Best Practice: Favor Shallow Inheritance

Instead of deep inheritance, **use composition** and keep inheritance trees shallow.

Tight Coupling Between Classes

Pitfall: Making classes too dependent on each other, reducing flexibility.

```
class Engine:
    def start(self):
        return "Engine started"


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


    def start(self):
        return self.engine.start()
```

Here, Car is **tightly coupled** with Engine. If Engine changes, Car must also be modified.

Better Approach: Dependency Injection

```
class Car:
    def __init__(self, engine):
```

```
        self.engine = engine  # Dependency injection

    def start(self):
        return self.engine.start()

# Injecting the dependency
engine = Engine()
car = Car(engine)
```

Now, Car can work with **any** engine type, increasing flexibility.

Using Large, Bloated Classes

Pitfall: A single class handling too many responsibilities violates the **Single Responsibility Principle (SRP)**.

Bad Example (Bloated Class)

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

    def calculate_bonus(self):
        return self.salary * 0.1

    def save_to_database(self):
        print("Saving employee record to database...")

    def generate_report(self):
        print(f"Employee: {self.name}, Salary: {self.salary}")
```

This class does **too much**—it calculates salary, saves to a database, and generates reports.

Better Approach: Split into Smaller Classes

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

```
class SalaryCalculator:
    def calculate_bonus(self, employee):
        return employee.salary * 0.1


class ReportGenerator:
    def generate_report(self, employee):
        print(f"Employee: {employee.name}, Salary:
{employee.salary}")
```

Now, **each class has a single responsibility**, making the code easier to modify.

Best Practices for Writing Clean OOP Code

Favor Composition Over Inheritance

- Instead of **deep class hierarchies**, **use composition** when appropriate.
- Example: Instead of class ElectricCar(Car), use Car containing an Engine object.

Keep Classes Small and Focused

- A class should have **one responsibility**.
- Avoid **"God classes"** that do everything.

Use Meaningful Class and Method Names

- Names should clearly indicate **purpose and behavior**.
- Example: class DataProcessor is better than class Processor.

Avoid Premature Optimization

- Write **simple, readable code** first.
- Optimize only when necessary.

Use Abstract Classes for Common Interfaces

- When multiple classes share the same behavior, use an **abstract base class**.

```
from abc import ABC, abstractmethod


class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass
```

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

Follow the SOLID Principles

- **S**: Single Responsibility → Keep classes focused.
- **O**: Open-Closed → Extend, don't modify existing code.
- **L**: Liskov Substitution → Subclasses should replace parent classes without changing behavior.
- **I**: Interface Segregation → Create small, focused interfaces.
- **D**: Dependency Inversion → Depend on abstractions, not concrete classes.

Exercises

1. Creating a Simple Class

Create a class called Person with the following attributes and methods:

- Attributes: name and age
- Method: introduce() that prints "Hello, my name is [name] and I am [age] years old."
- Create an instance of Person and call the method.

2. Adding Instance Methods

Extend the Person class by adding a birthday() method that increases the person's age by 1.

- Create an instance, call birthday(), and verify the age increases.

3. Encapsulation and Data Hiding

Modify the Person class:

- Make age a **private** attribute (__age).
- Add a getter method get_age() and a setter method set_age(new_age) that only allows positive values.
- Test the encapsulation by accessing and modifying age through the setter/getter.

4. Creating a Class with Inheritance

Create a class Employee that **inherits** from Person.

- Add a new attribute salary.
- Add a method work() that prints "I am working and earning [salary]".
- Create an instance of Employee and test the methods.

5. Method Overriding

Modify Employee to override introduce() so that it also includes "I work as an employee and earn [salary]".

- Create an Employee instance and call introduce() to verify overriding.

6. Polymorphism with Multiple Classes

Create two classes, Dog and Cat, both having a speak() method that returns "Woof!" and "Meow!" respectively.

- Write a function animal_sound(animal) that calls speak() on any object.
- Test with instances of Dog and Cat to demonstrate **polymorphism**.

7. Operator Overloading

Create a class Vector with x and y attributes.

- Implement the __add__() method to add two vectors.
- Test it by adding two Vector objects using the + operator.

8. Using Class and Static Methods

Create a BankAccount class with:

- **Class attribute** interest_rate (default: 0.05).
- **Instance attributes** owner and balance.
- A @classmethod set_interest_rate(new_rate) to modify interest_rate for all accounts.
- A @staticmethod bank_policy() that prints "Minimum balance required is $100".
- Create accounts, change the interest rate, and test the methods.

9. Abstract Classes and Interfaces

Create an abstract class Vehicle with:

- An abstract method max_speed() that must be implemented by subclasses.
- Create two subclasses, Car and Bike, that implement max_speed() differently.
- Test the implementation by creating instances and calling max_speed().

10. Composition vs. Inheritance

Create a class Engine with a method start().

- Create a class Car that **uses** an Engine instance (composition).
- Ensure Car has a start() method that calls Engine.start().
- Demonstrate composition by creating and starting a Car instance.