



Chapter 3

Object Oriented Programming Using Python

Student Learning Outcomes:

By the end of this chapter, students will be able to :

- Develop Python applications using object-oriented programming (OOP) principles.

Introduction

In this chapter, students will learn the fundamentals of Object-Oriented Programming (OOP) using Python. They will explore key OOP concepts like classes and objects to model real-world entities in code. The chapter covers encapsulation, which helps protect data and ensure privacy, and inheritance, allowing for code reuse and extension. Students will also study polymorphism to create flexible, adaptable programs. These concepts will help students develop well-structured and maintainable Python applications. By the end of the chapter, students will be proficient in applying OOP principles to design efficient and scalable programs.

3.1 Object-Oriented Programming (OOP) Using Python

Object-Oriented Programming is a method of designing programs using objects and classes. Each object represents a real-world entity with specific properties and behaviors. Classes act as templates for creating multiple objects with similar characteristics. OOP emphasizes encapsulation, inheritance, and polymorphism to manage data and behavior efficiently. This approach helps in modeling real-world problems in programming.



CLASS ACTIVITY

Create a chart comparing procedural programming and OOP. Have students explain scenarios where OOP is more useful.

Benefits of OOP: Modularity, Reusability, and Easier Maintenance

OOP allows code to be divided into modules, making it easier to manage. Classes and objects can be reused in multiple programs or parts of a program, reducing repetition. Programs built with OOP are easier to maintain and update because changes in one part do not affect the entire system. The structure of OOP supports scalability and adaptability in software development. It also improves collaboration by making code more understandable and organized.



TIDBIT

Object-Oriented Programming helps programmers think in terms of real-life objects, like students or cars, making programs easier to understand and manage.

3.2 Core Concepts of OOP

Object-Oriented Programming (OOP) has several core concepts that make programs organized, reusable, and easier to manage. These concepts include classes and objects, encapsulation, inheritance, and polymorphism.

DO YOU KNOW?

A class is like a blueprint, while an object is the real thing created from that blueprint.

Classes and Objects – Representing Real-World Entities in Code

A class is a blueprint or template that defines the properties (attributes) and actions (methods) of an entity. An object is an instance of a class that holds actual values for its attributes (as shown in Figure 3.1). Classes and objects allow programmers to model real-world entities in code. Using classes and objects helps organize data and related functionality together, making programs more structured.

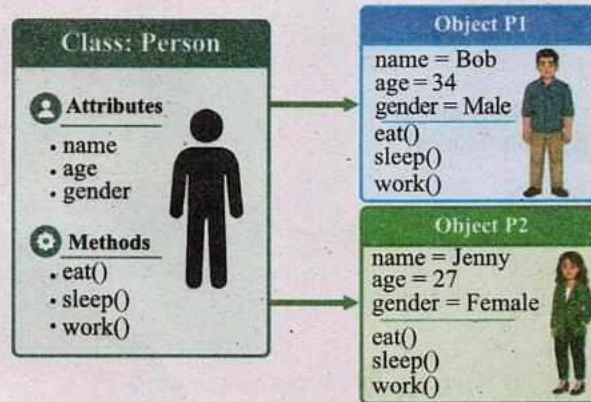


Figure 3.1: Diagram Showing Class and Object relationship

Example

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

    def display(self):
        print("Student Name:", self.name)

# Creating objects
object1 = Student("Ali")
object2 = Student("Ahmed")

# Using objects
object1.display()
object2.display()
```

- class Student → defines a blueprint for students
- `__init__` → sets the name when an object is created
- `self.name` → stores the student's name
- `display()` → prints the student's name
- `object1` and `object2` → two different student objects
- Each object has its own data (Ali, Ahmed)



Encapsulation

Encapsulation is a concept in Object Oriented Programming (OOP) where we bundle data (variables) and methods (functions) into a single unit (class) and restrict direct access to the data.

Unlike languages like Java or C++, Python does not have strict "private" or "protected" keywords. Instead, it relies on naming conventions to signal how data should be accessed.



TIDBIT

Hide the data and control access using methods.

Why Encapsulation is important

- Protects data from unwanted access
- Improves security
- Makes code easier to manage
- Allows controlled modification

Example:

Think of a bank account:

Your balance is hidden

You can only access it using:

- deposit()
- withdraw()

You cannot directly change the balance; this is encapsulation.

1. Levels of Access Control

In Python, you can control the visibility of class members using specific naming prefixes:

- **Public Members:** These are accessible from anywhere—inside or outside the class. By default, all members in Python are public (as shown in Figure 3.2).
 - *Syntax:* `self.attribute_name`
- **Protected Members:** These are intended for internal use within the class and its subclasses. It is a convention (not a restriction) to signal to other developers that the member should not be accessed from outside.
 - *Syntax:* `self._attribute_name` (single underscore)
- **Private Members:** These are highly restricted and cannot be easily accessed from outside the class. Python uses a mechanism called Name Mangling to change the internal name of the variable to prevent accidental access.
 - *Syntax:* `self.__attribute_name` (double underscore)

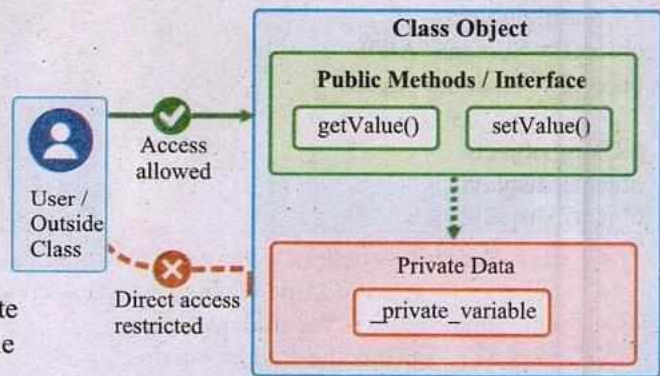


Figure 3.2: Encapsulation diagram showing private and public variables

Example:

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

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

    def get_balance(self):
        return self.__balance

# Usage
account = BankAccount(1000)

account.deposit(500)
account.withdraw(300)

print(account.get_balance())    # Output: 1200

# Direct access will fail
# print(account.__balance)    # Error
```

Inheritance – Reusing and Extending Functionality

Inheritance allows a class to take properties and methods from another class. The new class, called a subclass, can add or modify features of the parent class. Inheritance promotes code reuse and reduces redundancy. It also makes it easier to extend programs without rewriting existing code.

There are the following types of inheritance:

Single Inheritance: Single inheritance occurs when a child class (subclass) inherits attributes and methods from exactly one parent class (superclass). This is the most fundamental form of inheritance in Python, promoting code reusability by allowing the child class to extend or modify the behavior of the parent (as shown in Figure 3.3).

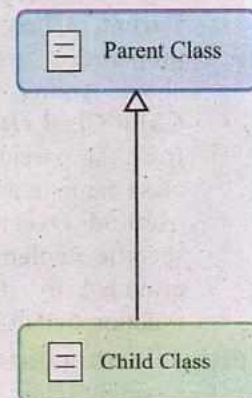


Figure 3.3: Single Inheritance



Example:

In this example, the Dog class inherits the basic functionality of an Animal but adds its own specific behavior.

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

    def eat(self):
        print(f'{self.name} is eating.')

# Child class (Subclass) inheriting from Animal
class Dog(Animal):
    def bark(self):
        print(f'{self.name} says Woof!')

# Create an object of the child class
my_dog = Dog("Buddy")

# Access method from Parent class
my_dog.eat() # Output: Buddy is eating.

# Access method from Child class
my_dog.bark() # Output: Buddy says Woof!
```



TIDBIT

self is the reference to the current object, passed automatically to instance methods.

- **Parent Class (Base Class):** The class whose properties are inherited. In the example above, Animal is the parent class.
- **Child Class (Derived Class):** The class that inherits from the parent. It is defined by passing the parent class name in parentheses: `class Child(Parent):`.
- **Method Overriding:** A child class can provide a specific implementation of a method that is already provided by its parent. You can use the `super()` function to call the parent's version of that method.

Multiple Inheritance: Multiple inheritance in Python allows a single class to inherit features (attributes and methods) from more than one parent class. This enables a child class to combine behaviors from different sources, such as a Smartphone class inheriting from both a Camera and a Computer class (as shown in Figure 3.4).

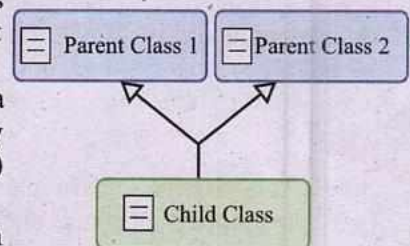


Figure 3.4: Multiple Inheritance



Example:

In this example, the Child class inherits from both Father and Mother classes. It can access methods from both parent classes.

```
class Father:
    def skill1(self):
        print("Father's skill: Gardening")
class Mother:
    def skill2(self):
        print("Mother's skill: Cooking")

# Derived class inheriting from two base classes
class Child(Father, Mother):
    def own_skill(self):
        print("Child's skill: Coding")
# Creating an instance of Child
c = Child()
c.skill1()          # Accessing Father's method
c.skill2()          # Accessing Mother's method
c.own_skill()       # Accessing its own method
```

Multilevel Inheritance: Multilevel inheritance in Python occurs when a class inherits from another derived class, creating a chain of inheritance similar to a family tree (Grandparent → Parent → Child). In this structure, the "grandchild" class has access to all methods and attributes from both its immediate parent and its grandparent (as shown in Figure 3.5).

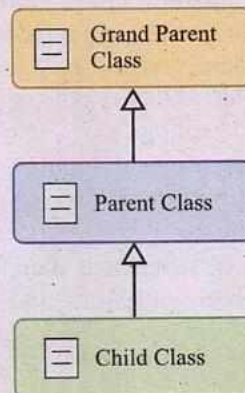


Figure 3.5: Multilevel Inheritance

Example:

In this example, ElectricCar inherits from Car, which in turn inherits from Vehicle.

```
Base Class (Grandparent)
class Vehicle:
    def start_engine(self):
        print("Engine started.")
```

```
# Intermediate Class (Parent)
class Car(Vehicle):
    def drive(self):
        print("Car is driving.")

# Derived Class (Child)
class ElectricCar(Car):
    def charge(self):
        print("Battery is charging.")

# Creating an instance of the bottom-most class
my_tesla = ElectricCar()

# Accessing methods from all levels
my_tesla.start_engine() # Inherited from Vehicle
my_tesla.drive()        # Inherited from Car
my_tesla.charge()       # Defined in ElectricCar
```

CLASS ACTIVITY

Create a Vehicle class and extend it with Car and Bike subclasses. Add unique methods to each subclass and demonstrate inheritance.

Polymorphism

Polymorphism means "many forms". It refers to the ability of an entity (like a function or object) to perform different actions based on the context. Technically, Polymorphism allows the same method, function, or operator to behave differently depending on the object it is working with. This makes code more flexible and reusable.

Example: In a backend payment system, multiple payment options are available, such as Credit Card, UPI, NetBanking, and Wallet. All payment types use a common method named `processPayment()`, but different implementations:

- NetBanking Payment: redirects to bank login

The method name remains the same, but the action changes based on the payment type.

Types of Polymorphism

Polymorphism refers to the ability of the same method or operation to behave differently based on object or context. It mainly includes compile-time and runtime polymorphism (as shown in Figure 3.6).

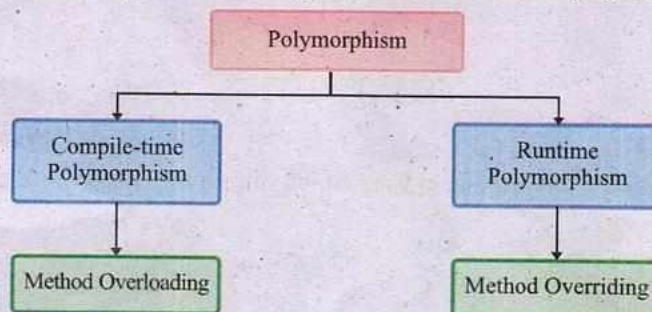


Figure 3.6: Polymorphism example showing the same function behaving differently



1. Compile-time Polymorphism

Compile-time polymorphism means deciding which method or operation to run during compilation, usually through method or operator overloading. Languages like Java or C++ support this. But python does not support true method overloading based on function signatures like statically typed languages (e.g., Java or C++). Because Python is dynamically typed, a single function can accept arguments of different types, making signature-based overloading unnecessary. Instead, similar behavior can be achieved using default arguments, variable-length arguments (*args, **kwargs), or conditional logic inside the function.

Example: This code demonstrates method overloading using default and variable-length arguments. The multiply() method works with different numbers of inputs, mimicking compile-time polymorphism.

<pre>class Calculator: def multiply(self, a=1, b=1, *args): result = a * b for num in args: result *= num return result # Create object calc = Calculator() # Using default arguments print(calc.multiply()) print(calc.multiply(4)) # Using multiple arguments print(calc.multiply(2, 3)) print(calc.multiply(2, 3, 4))</pre>	Output: 1 4 6 24
---	-------------------------------------

2. Runtime Polymorphism (Overriding)

Runtime polymorphism means that the behavior of a method is decided while program is running, based on the object calling it. This happens through Method Overriding a child class provides its own version of a method already defined in the parent class.

Example: This code shows runtime polymorphism using method overriding. The sound() method is defined in base class Animal and overridden in Dog and Cat. At runtime, the correct method is called based on object's class.

<pre>class Animal: def sound(self): return "Some generic sound" class Dog(Animal): def sound(self):</pre>	Output: Bark Meow Some generic sound
--	--



```
        return "Bark"

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

# Polymorphic behavior
animals = [Dog(), Cat(), Animal()]
for animal in animals:
    print(animal.sound())
```

Here, sound method behaves differently depending on whether object is a Dog, Cat or Animal and this decision happens at runtime. This dynamic nature makes Python particularly powerful for runtime polymorphism.



Summary

Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming approach in which a program is designed using objects that represent real-world entities.

Class

A class is a blueprint or template used to create objects. It defines the properties and behaviours of an object.

Object

An object is an instance of a class. It represents a real-world entity with its own data and functions.

Modularity

Modularity means dividing a program into smaller, manageable parts so that each part can be developed and maintained separately.

Reusability

Reusability means using already written code again in different parts of a program or in other programs.

Encapsulation

Encapsulation is the process of keeping data and related methods together inside a class. It helps protect data from unauthorized access.

Inheritance

Inheritance is an OOP concept in which one class can use the properties and methods of another class.

Polymorphism

Polymorphism means using the same method or operation in different ways depending on the object. It helps make code flexible and adaptable.



EXERCISE

✓ Multiple Choice Questions

1. The OOP concept that hides data inside a class is:
(a) Inheritance (b) Polymorphism
(c) Encapsulation (d) Abstraction
2. Classes and objects in Object-Oriented Programming are best described as:
(a) Classes are real-world entities, and objects are methods.
(b) Classes are blueprints, and objects are instances of those blueprints.
(c) Classes are used for data storage, and objects are used for functionality.
(d) Objects define functionality, and classes store data.
3. The main idea behind encapsulation in OOP is:
(a) To hide data inside a class and prevent unauthorised access.
(b) To create a flexible and adaptable programme.
(c) To reuse code from other classes.
(d) To allow multiple inheritance in classes.
4. Inheritance in OOP means:
(a) Reusing and extending functionality from a parent class to a child class.
(b) Hiding data in a class.
(c) Creating multiple instances of the same class.
(d) Managing polymorphic behaviour of classes.
5. Polymorphism in OOP is best defined as:
(a) Allowing an object to take multiple forms through method overriding.
(b) Hiding data inside a class for security.
(c) Using multiple inheritance to extend classes.
(d) Ensuring that each object has only one method.
6. An example of inheritance in Python is:
(a) Creating two classes with similar attributes.
(b) A child class inheriting attributes and methods from a parent class.
(c) Using private variables within a class.
(d) Modifying a method inside the same class.
7. The purpose of polymorphism in programming is:
(a) To create reusable functions with multiple parameter types.
(b) To allow a function or method to have different implementations.
(c) To combine different classes into a single class.
(d) To manage the flow of control in a programme.
8. The keyword used to define a class in Python is:
(a) def (b) class
(c) function (d) object



9. **Encapsulation is best described as:**

- (a) Combining data and methods into a single unit and controlling access to data.
- (b) Combining multiple classes into one class.
- (c) Inheriting methods from other classes.
- (d) Allowing objects to have multiple forms.

10. **Inheritance allows a class to:**

- (a) Hide its data only.
- (b) Inherit attributes and methods from another class.
- (c) Define multiple functions only.
- (d) Create objects without a class.



Short Questions

1. What is Object-Oriented Programming (OOP)?
2. What is the purpose of a class in Python?
3. What is the difference between a class and an object?
4. What is purpose of `__init__`?
5. What is encapsulation in OOP, and why is it important?
6. How does encapsulation help in keeping data secure and private?
7. What is inheritance in OOP, and how does it help in reusing functionality?
8. How does inheritance enable the extension of functionality in Python?
9. What is polymorphism in OOP, and how does it make code more flexible?
10. How does polymorphism help in creating adaptable and reusable code?



Long Questions

1. Explain the concept of classes and objects in Object-Oriented Programming (OOP).
2. What is encapsulation in OOP?
3. How does inheritance work in OOP?
4. What are the advantages of using inheritance in Python?
5. What is polymorphism in OOP, and how does it enhance flexibility in code?
6. How can encapsulation help improve data security in OOP?
7. How does the combination of inheritance, encapsulation, and polymorphism contribute to better software design?
8. What are the key differences between a class and an object in Object-Oriented Programming?



Answer Key

- | | |
|------|-------|
| 1. C | 6. B |
| 2. B | 7. B |
| 3. A | 8. B |
| 4. A | 9. A |
| 5. A | 10. B |