Mastering Object-Oriented Programming with Python

By Sarabjit Kaur

Acknowledgements

I would like to thank all those who supported me in the development of this book.
This includes my peers, educators, and the programming community whose resources and examples helped shape this content.
Special appreciation goes to the learners who continuously inspire the need for clarity and practical examples in programming education.

Table of Contents

Chapter 1 – Introduction to Python & OOP Paradigms

Why Python?
Python is beginner-friendly, readable, and powerful. It supports multiple paradigms including procedural and object-oriented programming.

Procedural Example:

```
name = "Sarabjit Kaur"
def greet():
    print("Hello,", name)
greet()
```

OOP Example:

```
class Person:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print("Hello,", self.name)
p1 = Person("Sarabjit Kaur")
p1.greet()
```

Mini Project – Library Book Tracker

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def display(self):
        print(f"'{self.title}' by {self.author}")

book1 = Book("Python Basics", "Manveer Singh")
book2 = Book("Advanced OOP", "Sahibveer Singh")

book1.display()
book2.display()
```

Summary:

- Python supports OOP for real-world modeling
- Classes and objects are foundational

Chapter 2 – Classes and Objects

What is a Class?

A class is a blueprint for creating objects. It defines the structure and behavior (via methods and attributes) that the created objects will have.

```
class Car:

    def __init__(self, brand, color):

        self.brand = brand

        self.color = color



    def drive(self):

        print(f"The {self.color} {self.brand} is driving.")



car1 = Car("Toyota", "Red")

car2 = Car("Honda", "Blue")



car1.drive()

car2.drive()
```

output:

```
The Red Toyota is driving.

The Blue Honda is driving.
```

- **`__init__()` is the constructor.**
- **`self` refers to the instance.**
- **`car1` and `car2` are instances (objects) of the `Car` class.**

Mini Project: Student Information System

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

    def display_info(self):
        print(f"Name: {self.name}, Roll No: {self.roll_number}, Course: {self.course}")

student1 = Student("Sarabjit Kaur", 101, "Python Programming")
student2 = Student("Manveer Singh", 102, "Data Science")

student1.display_info()
student2.display_info()
```

Output:

```
Name: Sarabjit Kaur, Roll No: 101, Course: Python Programming
Name: Manveer Singh, Roll No: 102, Course: Data Science
```

Chapter 3 – Attributes and Methods

Attributes

- **Instance Attributes**: Unique to each object
- **Class Attributes**: Shared by all instances

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

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

dog1 = Dog("Buddy", "Golden Retriever")
dog1.bark()

class Cat:
    species = "Felis catus"  # class attribute

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

```
cat1 = Cat("Whiskers")

cat2 = Cat("Mittens")

print(cat1.species, cat2.name)
```

Methods

- **Instance Method**: Acts on instance data
- **Class Method**: Acts on class data
- **Static Method**: Independent utility

Mini Project: Employee Record System

```
class Employee:
    company = "TechCorp"  # class attribute

    def __init__(self, name, emp_id):
        self.name = name
        self.emp_id = emp_id

    def display(self):
        print(f"Name: {self.name}, ID: {self.emp_id}, Company: {Employee.company}")

    @classmethod
    def change_company(cls, new_name):
```

```
        cls.company = new_name

    @staticmethod
    def is_valid_id(emp_id):
        return emp_id > 100

emp1 = Employee("Sarabjit Kaur", 101)
emp2 = Employee("Manveer Singh", 95)

emp1.display()
emp2.display()

print(Employee.is_valid_id(101))
print(Employee.is_valid_id(95))

Employee.change_company("NextGenTech")
emp1.display()
```

Chapter 4 – Encapsulation

What is Encapsulation?

Encapsulation is the concept of **bundling data and methods** that operate on that data within one unit (a class). It also refers to **restricting access** to internal variables to protect object integrity.

Benefits:

- Protects internal object state
- Improves modularity and maintainability
- Enables abstraction

Python Access Modifiers:

- `public` → accessible anywhere (`self.name`)
- `_protected` → accessible in subclasses (`self._age`)
- `__private` → accessible only within class (`self.__salary`)

```
class Person:
    def __init__(self, name, age):
        self.name = name       # public
        self._age = age        # protected
        self.__salary = 50000  # private

    def display(self):
        print(f"Name: {self.name}, Age: {self._age}")
```

```
    def get_salary(self):
        return self.__salary

    def set_salary(self, new_salary):
        if new_salary > 0:
            self.__salary = new_salary

p = Person("Sarabjit Kaur", 30)
p.display()
print(p.name)
print(p._age)
print(p.get_salary())
```

Mini Project: Student Management with Encapsulation

```
class Student:
    def __init__(self, name, roll_no, marks):
        self.name = name
        self.__roll_no = roll_no
        self.__marks = marks

    def get_roll_no(self):
        return self.__roll_no
```

```
 def set_roll_no(self, new_roll_no):
        if isinstance(new_roll_no, int):
            self.__roll_no = new_roll_no

    def get_marks(self):
        return self.__marks

    def set_marks(self, new_marks):
        if 0 <= new_marks <= 100:
            self.__marks = new_marks

    def display(self):
        print(f"Name: {self.name}, Roll No: {self.__roll_no}, Marks: {self.__marks}")

s = Student("Manveer Singh", 101, 85)
s.display()
s.set_marks(90)
print("Updated Marks:", s.get_marks())
```

Chapter 5 – Inheritance

What is Inheritance?

Inheritance allows a class (child) to **inherit attributes and methods** from another class (parent), promoting code reuse.

Types of Inheritance:

- Single
- Multiple
- Multilevel
- Hierarchical
- Hybrid

```
class Animal:
    def speak(self):
      print("Animal speaks")
class Dog(Animal):
    def bark(self):
        print("Dog barks")
d = Dog()
d.speak()
d.bark()
```

Method Overriding Example

```
class Animal:
    def speak(self):
        print("Animal speaks")

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

d = Dog()
d.speak()
```

Mini Project: Employee Management System

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

    def display(self):
        print(f"Name: {self.name}, ID: {self.emp_id}")

class FullTimeEmployee(Employee):
    def __init__(self, name, emp_id, salary):
```

```
        super().__init__(name, emp_id)
        self.salary = salary

    def display(self):
        super().display()
        print(f"Salary: {self.salary}")

class PartTimeEmployee(Employee):
    def __init__(self, name, emp_id, hourly_rate):
        super().__init__(name, emp_id)
        self.hourly_rate = hourly_rate

    def display(self):
        super().display()
        print(f"Hourly Rate: {self.hourly_rate}")

ft = FullTimeEmployee("Sarabjit Kaur", 101, 50000)
pt = PartTimeEmployee("Manveer Singh", 102, 200)

ft.display()
pt.display()
```

Chapter 6 – Polymorphism

What is Polymorphism?

Polymorphism allows objects of different types to be accessed through a common interface. In Python, it is mainly achieved via:

- **Method Overriding**
- **Duck Typing**

Method Overriding Example

```
class Animal:
    def speak(self):
        print("Animal speaks")

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

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

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

Duck Typing Example

```
class Bird:
    def fly(self):
        print("Bird can fly")

class Airplane:
    def fly(self):
        print("Airplane can fly")

def lift_off(flying_object):
    flying_object.fly()
```

```
lift_off(Bird())
lift_off(Airplane())
```

Mini Project: Drawing Shapes with Polymorphism

```
class Shape:
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Drawing a Circle")

class Square(Shape):
    def draw(self):
        print("Drawing a Square")

class Triangle(Shape):
    def draw(self):
        print("Drawing a Triangle")

shapes = [Circle(), Square(), Triangle()]

for shape in shapes:
    shape.draw()
```

Chapter 7 – Abstraction

What is Abstraction?

Abstraction hides internal implementation details and shows only the essential features. Python uses the `abc` module for abstraction via **abstract base classes**.

Benefits:

- Simplifies complex systems
- Enforces implementation rules
- Supports clean, maintainable design

Abstract Class Example

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

class Car(Vehicle):
    def start_engine(self):
        print("Starting engine of Sarabjit Kaur's car")

car = Car()
car.start_engine()
```

Mini Project: Online Course Platform

```
from abc import ABC, abstractmethod

class Course(ABC):
    @abstractmethod
    def course_info(self):
        pass

class PythonCourse(Course):
    def course_info(self):
        print("Python for Beginners by Manveer Singh")

class DataScienceCourse(Course):
    def course_info(self):
        print("Intro to Data Science by Sahibveer Singh")

course1 = PythonCourse()
course2 = DataScienceCourse()

course1.course_info()
course2.course_info()
```

Chapter 8 – OOP in Practice: Putting It All Together

Why Integrate All OOP Principles?

Combining **abstraction**, **inheritance**, **encapsulation**, and **polymorphism** allows us to design well-structured, maintainable applications.

Example: University Management System

```
from abc import ABC, abstractmethod

class Person(ABC):
    def __init__(self, name, id_number):
        self.name = name
        self._id_number = id_number  # Protected

    @abstractmethod
    def get_details(self):
        pass

class Student(Person):
    def __init__(self, name, id_number, course):
        super().__init__(name, id_number)
        self.__course = course  # Private

    def get_details(self):
        print(f"Student: {self.name}, ID: {self._id_number}, Course: {self.__course}")

class Professor(Person):
    def __init__(self, name, id_number, subject):
        super().__init__(name, id_number)
        self.subject = subject

    def get_details(self):
        print(f"Professor: {self.name}, ID: {self._id_number}, Subject: {self.subject}")

# Creating objects
s1 = Student("Sarabjit Kaur", 1001, "OOP in Python")
p1 = Professor("Manveer Singh", 2001, "Software Engineering")
```

```
# Polymorphic behavior

people = [s1, p1]
for person in people:
    person.get_details()
```

Chapter 9 – Real-World Projects

Project 1: Bank Account Management System

```
from abc import ABC, abstractmethod

class Account(ABC):
    def __init__(self, name, acc_number, balance=0):
        self.name = name
        self._acc_number = acc_number
        self.__balance = balance

    @abstractmethod
    def account_type(self):
        pass

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount

    def get_balance(self):
        return self.__balance

class SavingsAccount(Account):
    def account_type(self):
        return "Savings"

class CurrentAccount(Account):
    def account_type(self):
        return "Current"

# Creating accounts
acc1 = SavingsAccount("Sarabjit Kaur", 10001, 5000)
acc2 = CurrentAccount("Sahibveer Singh", 10002, 8000)
```

```
# Perform transactions
acc1.deposit(2000)
acc2.withdraw(3000)

print(f"{acc1.name}'s Balance: {acc1.get_balance()} ({acc1.account_type()})")
print(f"{acc2.name}'s Balance: {acc2.get_balance()} ({acc2.account_type()})")
```

Project 2: School Grading System

```
from abc import ABC, abstractmethod

class Person(ABC):
    def __init__(self, name, id_no):
        self.name = name
        self._id_no = id_no

    @abstractmethod
    def display_details(self):
        pass

class Student(Person):
    def __init__(self, name, id_no, marks):
        super().__init__(name, id_no)
        self.__marks = marks

    def display_details(self):
        print(f"Student: {self.name}, ID: {self._id_no}, Marks: {self.__marks}")

class Teacher(Person):
    def __init__(self, name, id_no, subject):
        super().__init__(name, id_no)
        self.subject = subject

    def display_details(self):
        print(f"Teacher: {self.name}, ID: {self._id_no}, Subject: {self.subject}")

# Create instances
student1 = Student("Manveer Singh", 501, 88)
teacher1 = Teacher("Sarabjit Kaur", 101, "Computer Science")
```

```
# Use polymorphism
people = [student1, teacher1]
for p in people:
    p.display_details()
```

Chapter 10 – Best Practices & Advanced Tips

Introduction

Mastering OOP isn't just about syntax—it's about writing **clean, maintainable, and professional code**. This chapter outlines essential best practices and some powerful advanced features.

OOP Best Practices in Python

1. Follow the DRY Principle

Don't Repeat Yourself—reuse code via functions, inheritance, and composition.

2. Use Descriptive Naming

- Classes: `CamelCase` → `StudentManager`
- Functions/variables: `snake_case` → `calculate_average`

3. Encapsulate Data Thoughtfully

Use `_protected` or `__private` for internal attributes. Expose only what's needed via getters/setters.

4. Prefer Composition Over Inheritance

If "has-a" makes more sense than "is-a", use composition.

```
class Engine:
    def start(self):
        print("Engine started")
class Car:
    def __init__(self):
        self.engine = Engine()

    def drive(self):
        self.engine.start()
        print("Car is moving")
```

SOLID Design Principles (adapted for Python)

1. **S - Single Responsibility**
   A class should only have one job.

2. **O - Open/Closed Principle**
   Open for extension, closed for modification.

3. **L - Liskov Substitution**
   Subclasses should replace base classes without breaking functionality.

4. **I - Interface Segregation**
   Use specific abstract interfaces (or base classes), not bulky ones.

5. **D - Dependency Inversion**
   Depend on abstractions, not concrete classes.

Debugging & Refactoring Tips

- Use `__str__()` for better print formatting
- Use tools like `pylint`, `black`, and `flake8`
- Use `type()` and `isinstance()` for safe runtime checks
- Refactor regularly to keep code clean and modular

Advanced Features to Explore

1. Magic Methods

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

    def __str__(self):
        return f"Student: {self.name}"

s = Student("Sarabjit Kaur")
print(s)
```

2. Property Decorators

```
class Course:
    def __init__(self, title):
        self._title = title

    @property
    def title(self):
        return self._title

    @title.setter
    def title(self, new_title):
        if new_title:
            self._title = new_title
```

3. Mixins

```
class LoggerMixin:
    def log(self, msg):
        print(f"[LOG]: {msg}")

class User(LoggerMixin):
    def __init__(self, name):
        self.name = name

    def display(self):
        self.log(f"User: {self.name}")
```