Python

Unit- 3

KEYUR BHATIYA

Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is a programming paradigm based on objects, which contain data (attributes) and behavior (methods). Python supports OOP through **classes** and **objects**.

---

1. Classes and Objects

What is a Class?

A **class** is a blueprint for creating objects. It defines attributes (variables) and methods (functions) that describe an object's behavior.

What is an Object?

An **object** is an instance of a class. It has its own state and behavior defined by the class.

Example: Defining a Class and Creating an Object

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Attribute
        self.model = model  # Attribute

    def display_info(self):  # Method
        print(f"Car: {self.brand} {self.model}")

# Creating an object
car1 = Car("Tesla", "Model S")
car1.display_info()
```

Output:

```
Car: Tesla Model S
```

2. Abstract Data Types (ADTs)

Definition:

An Abstract Data Type (ADT) is a mathematical model for a data structure that specifies the type of data stored, the operations allowed on them, and the behavior of these operations, without specifying how the data structure is implemented.

Common ADTs:

1. List – Dynamic array with indexed access.
2. Stack – Last In, First Out (LIFO) principle.
3. Queue – First In, First Out (FIFO) principle.
4. Deque – Double-ended queue.
5. Set – Collection of unique elements.
6. Map (Dictionary) – Collection of key-value pairs.
7. Graph – Nodes (vertices) connected by edges.
8. Tree – Hierarchical data structure.

Example: Implementing Stack ADT using a Class

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop() if not self.is_empty() else "Stack is empty"

    def peek(self):
        return self.items[-1] if not self.is_empty() else "Stack is empty"

    def is_empty(self):
        return len(self.items) == 0

    def size(self):
        return len(self.items)
```

Classes in Python

Definition:

A class is a blueprint for creating objects, encapsulating data (attributes) and functions (methods) that operate on the data.

Example:

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

    def display_info(self):
        return f"Car: {self.brand} {self.model}"

car1 = Car("Toyota", "Crysta")
print(car1.display_info())  # Output: Car: Toyota Crysta
```

3. Encapsulation and Information Hiding

Encapsulation is the concept of **restricting direct access** to data within a class and modifying it only through methods.

Access Modifiers in Python

Modifier	Meaning
public	Accessible anywhere (default)
_protected	Suggests limited access (convention only)
__private	Name-mangled to prevent direct access

Example of Encapsulation

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

    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

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # 1500
```

Trying to access __balance directly will result in an error.

4. Inheritance (Code Reusability)

Inheritance allows one class to inherit attributes and methods from another class.

Types of Inheritance

1. **Single Inheritance** (One class inherits from another)
2. **Multiple Inheritance** (One class inherits from multiple classes)
3. **Multilevel Inheritance** (Class inherits from another derived class)
4. **Hierarchical Inheritance** (Multiple classes inherit from a single parent)
5. **Hybrid Inheritance** (Combination of multiple types)

Example: Single Inheritance

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

    def speak(self):
        return "Animal speaks"

class Dog(Animal):  # Inheriting from Animal
    def speak(self):
        return "Woof! Woof!"

dog = Dog("Buddy")
print(dog.name)      # Buddy
print(dog.speak())   # Woof! Woof!
```

Example: Multiple Inheritance

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

class Car:
    def drive(self):
        print("Car is moving")

class ElectricCar(Engine, Car):  # Inherits from both Engine and Car
    pass

tesla = ElectricCar()
tesla.start()  # Engine started
tesla.drive()  # Car is moving
```

Concept	Description	Example
Class	Blueprint for creating objects	class Car: pass
Object	Instance of a class	car1 = Car()
Encapsulation	Restricting direct access to data	self.__balance = balance
Inheritance	One class acquires attributes/methods from another	class Dog(Animal): pass
Public Attributes	Accessible anywhere	self.name
Protected Attributes	Suggested limited access	self._name
Private Attributes	Name-mangled to restrict access	self.__name

Encapsulation and Information Hiding in Python

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of data and methods that operate on that data into a single unit, typically a class. Encapsulation helps in **data hiding**, preventing direct access to certain variables and restricting modifications to them.

1. Understanding Encapsulation

Encapsulation allows an object's internal state to be hidden from the outside world, exposing only what is necessary. This is achieved using:

- **Public Members**: Accessible from anywhere.
- **Protected Members (_variable)**: Indicated by a single underscore, suggesting they should not be accessed directly.
- **Private Members (__variable)**: Indicated by a double underscore, making them inaccessible directly from outside the class.

Example of Encapsulation

```
class Car:
    def __init__(self, brand, speed):
        self.brand = brand       # Public attribute
        self._speed = speed      # Protected attribute
        self.__engine = "V8"     # Private attribute

    def get_speed(self):
        return self._speed

    def set_speed(self, speed):
        if speed > 0:
            self._speed = speed
        else:
            print("Speed must be positive!")

    def get_engine(self):
        return self.__engine  # Private attribute accessed through a method

car = Car("Toyota", 100)
print(car.brand)      # Accessible
print(car.get_speed()) # Accessible through method
print(car._speed)     # Not recommended, but possible
# print(car.__engine)  # Raises AttributeError
print(car.get_engine())  # Correct way to access private data
```

2. Information Hiding

Information hiding is a concept that supports encapsulation by restricting direct access to some parts of an object. In Python, information hiding is enforced using:

- **Private attributes and methods** (double underscores __)
- **Getter and setter methods** to control attribute access
- **Property decorators** to manage access elegantly

Example of Information Hiding

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

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount}. New balance: ${self.__balance}")
        else:
            print("Deposit amount must be positive!")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount}. Remaining balance: ${self.__balance}")
        else:
            print("Insufficient funds or invalid amount!")

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

account = BankAccount("Alice", 500)
account.deposit(200)
account.withdraw(100)
# print(account.__balance)  # Raises AttributeError
print(account.get_balance())  # Correct way to access balance
```

3. Using @property Decorator

The @property decorator allows defining methods that can be accessed like attributes, making encapsulation more Pythonic.

Example using @property

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

    @property
    def salary(self):
        return self.__salary

    @salary.setter
    def salary(self, new_salary):
        if new_salary > 0:
            self.__salary = new_salary
        else:
            print("Salary must be positive!")

emp = Employee("John", 5000)
print(emp.salary)  # Accessing private variable through property
emp.salary = 6000  # Modifying using setter
print(emp.salary)
# emp.__salary = 7000  # Raises AttributeError
```

Case Study: Banking Application with Mortgages

In this case study, we will build a simple **Banking Application** that includes account management and mortgage loan calculations. We'll cover key concepts like **object-oriented programming (OOP), encapsulation, inheritance, and polymorphism** in Python.

---

1. Problem Statement

A bank wants to develop an application where users can:

- **Create bank accounts** (Savings or Current)
- **Deposit and withdraw money**
- **Apply for a mortgage loan**
- **Calculate monthly payments on a mortgage**
- **View account and loan details**

2. Design & Class Structure

Main Classes:

1. **BankAccount (Base Class)**
    - Attributes: account_number, account_holder, balance
    - Methods: deposit(), withdraw(), get_balance()
2. **SavingsAccount & CurrentAccount (Derived Classes)**
    - SavingsAccount: Offers interest.
    - CurrentAccount: Allows overdraft.
3. **Mortgage**
    - Attributes: loan_amount, interest_rate, term_years
    - Methods: calculate_monthly_payment()

3. Implementing the Classes

Step 1: Bank Account Base Class

```
class BankAccount:
    def __init__(self, account_number, account_holder, balance=0):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print(f"Deposited {amount}. New balance: {self.balance}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount > 0 and amount <= self.balance:
            self.balance -= amount
            print(f"Withdrawn {amount}. Remaining balance: {self.balance}")
        else:
            print("Insufficient balance or invalid amount.")

    def get_balance(self):
        return self.balance
```

Step 2: Savings and Current Accounts (Inheritance)

```
class SavingsAccount(BankAccount):
    def __init__(self, account_number, account_holder, balance=0, interest_rate=0.02):
        super().__init__(account_number, account_holder, balance)
        self.interest_rate = interest_rate

    def add_interest(self):
        interest = self.balance * self.interest_rate
        self.balance += interest
        print(f"Interest added: {interest}. New balance: {self.balance}")

class CurrentAccount(BankAccount):
    def __init__(self, account_number, account_holder, balance=0, overdraft_limit=500):
        super().__init__(account_number, account_holder, balance)
        self.overdraft_limit = overdraft_limit

    def withdraw(self, amount):
        if amount > 0 and (self.balance - amount >= -self.overdraft_limit):
            self.balance -= amount
            print(f"Withdrawn {amount}. Remaining balance: {self.balance}")
        else:
            print("Overdraft limit exceeded or invalid amount.")
```

Step 3: Mortgage Loan Class

```
class Mortgage:
    def __init__(self, loan_amount, interest_rate, term_years):
        self.loan_amount = loan_amount
        self.interest_rate = interest_rate / 100  # Convert to decimal
        self.term_years = term_years

    def calculate_monthly_payment(self):
        months = self.term_years * 12
        monthly_rate = self.interest_rate / 12
        if monthly_rate > 0:
            monthly_payment = (self.loan_amount * monthly_rate) / (1 - (1 + monthly_rate) ** -
months)
        else:
            monthly_payment = self.loan_amount / months  # If zero interest rate
        return round(monthly_payment, 2)
```

4. Running the Application

```
# Create Bank Accounts
savings = SavingsAccount("S123", "Alice", 5000)
current = CurrentAccount("C456", "Bob", 1000)

# Perform Transactions
savings.deposit(1000)
savings.add_interest()

current.withdraw(1200)  # Allowed within overdraft limit
current.deposit(500)

# Mortgage Loan Calculation
mortgage = Mortgage(loan_amount=200000, interest_rate=3.5, term_years=30)
monthly_payment = mortgage.calculate_monthly_payment()
print(f"Monthly Mortgage Payment: {monthly_payment}")
```

5. Expected Output

Deposited 1000. New balance: 6000

Interest added: 120.0. New balance: 6120.0

Withdrawn 1200. Remaining balance: -200

Deposited 500. New balance: 300

Monthly Mortgage Payment: 898.09