Artificial Intelligence

AI-2002

Lab 02

OOP in Python

Course Code: AI-2002	Artificial Intelligence Lab

1.

1. Objective

1. Introduction to Object-Oriented Programming (OOP) concepts using Python.
2. To develop basic OOP programming skills in Python using classes and objects.
3. To implement core OOP principles such as encapsulation, inheritance, and polymorphism.
4. To solve basic programming problems using OOP in Python

2. Introduction to OOP

Object-Oriented Programming (OOP) is a programming paradigm that organizes code using classes and objects to model real-world entities. In Python, OOP allows programmers to create reusable and modular code by encapsulating data and functions within classes. Key concepts of OOP include encapsulation (hiding internal details), inheritance (reusing code from existing classes), polymorphism (using the same interface for different objects), and abstraction (focusing on essential features while hiding complexity). Learning OOP in Python helps students design structured programs, improve problem-solving skills, and develop software that is easier to maintain and extend.

2.1 Class

A class is a blueprint or template for creating objects. It defines the attributes (data) and methods (functions) that the objects created from it will have. Think of it as a plan for building something, like a blueprint for a house.

How to Create a Class in Python

```
class Student:
    pass
```

2.2 Object

An object is an instance of a class. It represents a real entity created using the class blueprint, with its own values for the attributes.

How to Call and Object in Python

```
s1 = Student()    # s1 is an object of class Student
s2 = Student()   # s2 is another object
```

2.3 Attributes and Methods

In Python, **attributes** are variables that belong to a class and hold the data or properties of an object. **Methods** are functions defined inside a class that describe the behaviors or actions of an object. Attributes store information specific to each object, while methods allow objects to perform operations using that data. Using attributes and methods together helps in organizing code in a structured and reusable way, making programs easier to understand and maintain.

Creating/Declaring Attributes

```
class MyClass:
  x = 5 #x is an attribute
```

Creating/Declaring Method

```
class MyClass:
  x = 5 #x is an attribute

  #Method
  def method_one(self):
    pass
```

Accessing Attributes and Methods

```
print(m1.x)
m1.method_one()
```

Full Code

```
class MyClass:
  x = 5 #x is an attribute

  #Method
  def method_one(self):
    print("This is method one")

m1= MyClass()  #m1 is attribut
print(m1.x)
m1.method_one()
```

Self in Python

In Python, self is a convention-based parameter used in instance methods of a class to refer to the current instance (object) of the class being operated on

2.4 Constructor in Python

A constructor is a special method in a class that is automatically called when an object is created. It is used to initialize the attributes of an object. In Python, the constructor method is always named __init__().

Creating and Accessing Constructor

```
class Student:
    def __init__(self, name, age):
        self.name = name  # instance attribute
        self.age = age    # instance attribute

# Creating objects
s1 = Student("Ali", 20)
s2 = Student("Sara", 22)

print(s1.name, s1.age)  # Output: Ali 20
print(s2.name, s2.age)  # Output: Sara 22
```

2.5 Types of Attributes in a Class

In Python, attributes are variables that belong to a class. They are used to store data related to **objects**. There are **two main types of attributes**:

1. Class Attribute
2. Instance Attribute

Class Attribute

- Defined directly inside the class, outside any method.
- Shared by all instances of the class.

```
class Student:
    school = "ABC High School"  # class attribute
```

```
s1 = Student()
s2 = Student()
print(s1.school)  # ABC High School
print(s2.school)  # ABC High School
```

Instance Attribute

- Defined inside the __init__ constructor using self.
- Each object (instance) has its own copy of these attributes.

```
class Student:
    def __init__(self, name, age):
        self.name = name    # instance attribute
        self.age = age      # instance attribute

s1 = Student("Ali", 20)
s2 = Student("Sara", 22)
print(s1.name, s1.age)  # Ali 20
print(s2.name, s2.age)  # Sara 22
```

Modifying Class and Instance Attribute

=

```
class Student:
    school_name = "ABC School"  # class attribute (shared by all
students)

    def __init__(self, name, age):
        self.name = name    # instance attribute
        self.age = age      # instance attribute

    def display_info(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"School: {Student.school_name}")

# Creating student objects (instances)
student1 = Student("Ali", 15)
student2 = Student("Sara", 16)

# Displaying info
student1.display_info()
```

```
print("----------------")
student2.display_info()

# Modifying an instance attribute
student1.age = 16
print("\nAfter updating age of student1:")
student1.display_info()

# Modifying the class attribute
Student.school_name = "XYZ School"
print("\nAfter updating school name:")
student1.display_info()
student2.display_info()
```

Example

Suppose you are asked to create a program to **manage bank accounts**. Each account has a **holder's name, account number, and balance**, and can perform actions like viewing account info and depositing money.

This scenario demonstrates the importance of OOP in Python:

- **Modularity:** Code is organized into a class BankAccount, separating account logic from other code.
- **Reusability:** Multiple accounts can be created from the same class without rewriting code.

```
class BankAccount:
    bank_name = "ABC Bank"  # class attribute shared by all accounts

    def __init__(self, holder_name, account_no, balance):
        self.holder_name = holder_name    # instance attribute
        self.account_no = account_no      # instance attribute
        self.balance = balance            # instance attribute

    def display_account(self):            # method to display account
info
        print("Bank:", BankAccount.bank_name)
        print("Account Holder:", self.holder_name)
        print("Account Number:", self.account_no)
        print("Balance:", self.balance)
```

```
    def deposit(self, amount):            # method to deposit money
        self.balance += amount
        print(f"Deposited {amount}. New Balance: {self.balance}")

#Create Objects / Instances
account1 = BankAccount("Ali", 1001, 5000)
account2 = BankAccount("Sara", 1002, 7000)

#Call Methods
account1.display_account()
print("-----------------")
account2.display_account()
print("-----------------")
account1.deposit(2000)
```

2.6 Practice Task # 1

Create a program to control smart lights in a house. Each light has a room name and a status (ON/OFF). The system should allow:

1. Creating multiple **light objects** for different rooms.
2. Turning lights **ON or OFF**.
3. Displaying the **current status** of each light.

Objective:
This task demonstrates **OOP in AI automation**:

- **Classes** organize devices logically.
- **Objects** represent individual lights.
- **Methods** automate actions like turning ON/OFF.

Expected Output

```
Living Room light turned ON.
Bedroom light turned OFF.
Living Room light is ON.
Bedroom light is OFF.
```

3. Inheritance in Python

Inheritance is a feature of Object-Oriented Programming (OOP) that allows a class (called the child or subclass) to reuse the properties and methods of another class (called the parent or superclass). This helps in code reusability and reduces redundancy.

- Parent Class (Superclass): The class whose properties and methods are inherited.
- Child Class (Subclass): The class that inherits properties and methods from the parent class.
- Child classes can add new methods or override existing ones from the parent class.

Inheritance Basic Syntax

```
class ParentClass:
    # parent class code

class ChildClass(ParentClass):
    # child class code
```

Inheritance without Super

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

    def display_person(self):
        print(f"Name: {self.name}, Age: {self.age}")
# Child class without __init__
class Student(Person):
    def display_student(self):
        print("This is a student")

# Creating child object
s1 = Student("Ali", 16)
s1.display_person()      # inherited from parent
s1.display_student()     # child-specific method
```

Inheritance with Super ()

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

    def display_person(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Child class with its own __init__ (without super)
class Student(Person):
    def __init__(self, roll_no):
        self.roll_no = roll_no   # only child attribute is
initialized

    def display_student(self):
        print(f"Roll Number: {self.roll_no}")

# Creating object of child class
s1 = Student(101)
```

```
# Trying to access attributes
print(s1.roll_no)       # works, child attribute exists

# print(s1.name)        # ERROR! Parent attributes are NOT
initialized
# s1.display_person()   # ERROR! Parent method uses attributes
that are not initialized

s1.display_student()    # works
```

3.1 Types of Inheritance

1. **Single Inheritance:** A child class inherits from a single parent class.
2. **Multiple Inheritance:** A child class inherits from more than one parent class.
3. **Multilevel Inheritance:** A child class inherits from a parent class, which in turn inherits from another class.
4. **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class.
5. **Hybrid Inheritance:** A combination of two or more types of inheritance.

Single Inheritance

```
# Parent class
class Parent:
    def greet(self):
```

```
        print("Hello from Parent")

# Child class
class Child(Parent):
    def greet_child(self):
        print("Hello from Child")

c = Child()
c.greet()         # inherited from Parent
c.greet_child()   # child method
```

Multiple Inheritance

```
class Parent1:
    def greet_parent1(self):
        print("Hello from Parent1")

class Parent2:
    def greet_parent2(self):
        print("Hello from Parent2")

class Child(Parent1, Parent2):
    def greet_child(self):
        print("Hello from Child")
c = Child()
c.greet_parent1()
c.greet_parent2()
c.greet_child()
```

MultiLevel Inheritance

```
# Grandparent class
class Grandparent:
    def greet_grandparent(self):
        print("Hello from Grandparent")

# Parent class
class Parent(Grandparent):
    def greet_parent(self):
        print("Hello from Parent")

# Child class
class Child(Parent):
    def greet_child(self):
        print("Hello from Child")

c = Child()
c.greet_grandparent()  # inherited from Grandparent
c.greet_parent()       # inherited from Parent
c.greet_child()        # own method
```

Hierarchal Inheritance

```
class Parent:
    def greet_parent(self):
        print("Hello from Parent")

class Child1(Parent):
    def greet_child1(self):
        print("Hello from Child1")

class Child2(Parent):
    def greet_child2(self):
        print("Hello from Child2")
c1 = Child1()
c2 = Child2()
c1.greet_parent()
c1.greet_child1()
```

Hybrid Inheritance

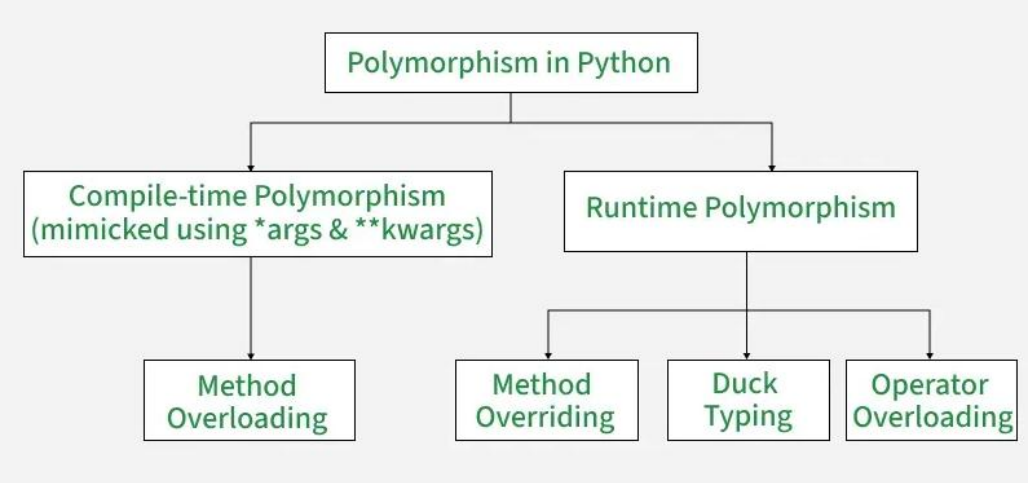
```
class Grandparent:
    def greet_grandparent(self):
        print("Hello from Grandparent")

class Parent1(Grandparent):
    def greet_parent1(self):
        print("Hello from Parent1")

class Parent2:
    def greet_parent2(self):
        print("Hello from Parent2")

class Child(Parent1, Parent2):
    def greet_child(self):
        print("Hello from Child")

c = Child()
c.greet_grandparent()
c.greet_parent1()
c.greet_parent2()
c.greet_child()
```

Practice Task # 2

You are designing a **University Staff Management System**. The system needs to keep track of different staff members: **Teachers, Administrative Staff, and Research Assistants**. All staff share common attributes like name, staff_id, and department, but each type has its own specific properties and behaviors. Teachers have courses and salary and can teach courses, Admin Staff have role and working_hours and perform tasks, while Research Assistants have research_topic and stipend and work on research. Your task is to create a **base class** Staff and derive classes for each staff type, instantiate objects, and call their respective methods to simulate daily activities. Optionally, override the display_info() method to show type-specific details.

4. Polymorphism in Python

Polymorphism means “many forms”. In Python OOP, it allows different objects or methods to respond differently to the same action.



4.1. Compile Time Polymorphism

This type of polymorphism is determined during the compilation of the program. It allows methods or operators with the same name to behave differently based on their input parameters or usage. In languages like Java or C++, compile-time polymorphism is achieved through method overloading but it's not directly supported in Python.

In Python:

- True compile-time polymorphism is not supported.
- Instead, Python mimics it using default arguments or *args/**kwargs.
- Operator overloading can also be seen as part of polymorphism, though it is implemented at runtime in Python

```
# ---------- Method Overloading (Simulated) using Default Argument
class Calculator:
    def add(self, a, b=0):
        """Add two numbers, or one number if only one is given"""
        print("Sum:", a + b)

calc = Calculator()
print("Method Overloading Examples:")
calc.add(5, 10)  # Using two parameters
calc.add(7)      # Using one parameter (default b=0)

# ---------- Method Overloading (Simulated) using Kwargs Argument
class Calculator:
    def add(self, *args):
        return sum(args)

calc = Calculator()
print(calc.add(5, 10))        # Two arguments
print(calc.add(5, 10, 15))    # Three arguments
print(calc.add(1, 2, 3, 4))   # Any number of arguments
```

4.2. Run Time Polymorphism

Run-Time Polymorphism is determined during the execution of the program. It covers multiple forms in Python:

1. **Method Overriding:** A subclass redefines a method from its parent class.
2. **Duck Typing:** If an object implements the required method, it works regardless of its type.
3. **Operator Overloading:** Special methods (__add__, __sub__, etc.) redefine how operators behave for user-defined objects.

Example: In this example, we show run-time polymorphism using method overriding with dog classes and compile-time polymorphism by mimicking method overloading in a calculator class.

Method Overriding

```
# # Parent class
class Animal:
    def sound(self):
        print("Some generic sound")

# Child class
class Dog(Animal):
    def sound(self):
        print("Bark")  # overrides parent method

class Cat(Animal):
    def sound(self):
        print("Meow")  # overrides parent method

# Runtime polymorphism
a = Dog()
b = Cat()
a.sound()  # Bark
b.sound()  # Meow
```

4.6 Encapsulation in Python

Encapsulation is an OOP principle that restricts direct access to the internal data of a class and bundles the data (attributes) and methods that operate on that data together. It helps to:

- Protect sensitive data
- Control how attributes are accessed or modified
- Make code more secure and maintainable

In Python, encapsulation is implemented by using private attributes, which are prefixed with a single _ (protected) or double __ (private).

Types of Encapsulation:

1. **Public Members**: Accessible from anywhere.
2. **Protected Members**: Accessible within the class and its subclasses.
3. **Private Members**: Accessible only within the class.

Private Methods and Attributes

```
class Student:
    def __init__(self, name, roll_no):
        self.name = name          # public attribute
        self.roll_no = roll_no    # public attribute

    # public method
    def display_info(self):
        print(f"Name: {self.name}, Roll No: {self.roll_no}")

# Accessing within the same class
s = Student("Ali", 101)
print("Accessing Public Attribute and Method in same class:")
print(s.name)           # public attribute
print(s.roll_no)
s.display_info()        # public method

# Accessing from another class (subclass)
class CollegeStudent(Student):
    def show(self):
        print("\nAccessing Public Attribute and Method from
subclass:")
        print(self.name)
        print(self.roll_no)
        self.display_info()

cs = CollegeStudent("Sara", 102)
cs.show()
```

Protected Methods and Attributes

```
class Student:
    def __init__(self, name, roll_no):
        self._name = name         # protected attribute
        self._roll_no = roll_no   # protected attribute

    # protected method
    def _display_info(self):
        print(f"Name: {self._name}, Roll No: {self._roll_no}")

# Accessing within the same class
s = Student("Ali", 101)
print("\nAccessing Protected Attribute and Method in same class:")
print(s._name)
print(s._roll_no)
s._display_info()

# Accessing from another class (subclass)
class CollegeStudent(Student):
    def show(self):
        print("\nAccessing Protected Attribute and Method from
subclass:")
        print(self._name)
        print(self._roll_no)
        self._display_info()

cs = CollegeStudent("Sara", 102)
cs.show()
```

Private Methods and Attributes

```
class Student:
    def __init__(self, name, roll_no):
        self.__name = name        # private attribute
        self.__roll_no = roll_no  # private attribute

    # private method
    def __display_info(self):
        print(f"Name: {self.__name}, Roll No: {self.__roll_no}")

    # public method to access private members
    def access_private(self):
        print("Accessing private members inside the same class:")
        print(self.__name)
        print(self.__roll_no)
        self.__display_info()

# Accessing private attribute and method **inside the same class**
via public method
s = Student("Ali", 101)
s.access_private()

# Accessing from another class (subclass)
class CollegeStudent(Student):
    def show(self):
        print("\nTrying to access private members from subclass:")
        # The following will cause error if uncommented:
        # print(self.__name)
        # self.__display_info()
        # Correct way: use a public method
        self.access_private()

cs = CollegeStudent("Sara", 102)
cs.show()
```

Practice Task # 3

You are designing a **simple banking system** where each customer has a bank account. The account balance should be **private** and not directly accessible from outside the class. Your task is to create a BankAccount class with a private attribute __balance and public methods to deposit, withdraw, and get_balance. Create a few account objects and use these methods to perform transactions, observing how encapsulation protects the balance from direct access.