2. Polymorphism:-

Poly	Many
Morphism	forms

In OOP, the same method name can perform different tasks based on **object**, **arguments**, or **context**.

Types of Polymorphism:-

1. **Compile-time Polymorphism**
2. **Run-time Polymorphism**

1. Compile-time Polymorphism

- Compile-time polymorphism is achieved **at compile time**.
- In Python, it is mainly achieved using **Method Overloading.**

Method Overloading:-

Method overloading means **same method name with different parameters**.

Note	Python does **not support true method overloading**, but we can achieve it using:- 1. Default arguments 2. Variable-length arguments

Example:-

```
class Math:
    def add(self, a=0, b=0, c=0):
        print(a + b + c)
```

```
obj = Math()
obj.add(10, 20)
obj.add(10, 20, 30)
```

Monkey Patching:-

Monkey patching is a phenomenon of **assigning a function reference (address) to a variable**, so that the variable can act like a function even after the function is redefined.

Example:

```
def sam():
    print("Polymorphism")

# Store function address
mid = sam

# Redefine function
def sam(a, b, c):
    print(a * b * c)

sam(10, 20, 30)

mid()
```

2. Run-time Polymorphism

- Run-time polymorphism is achieved **at runtime**.
- It is implemented using **Method Overriding**.

Method Overriding:-

Method overriding means **child class provides its own implementation of a parent class method**.

Rules:-

1. Method name must be same
2. Parameters must be same
3. Inheritance must be present

Example:-

```
# Parent Class
class Bank:
    def rate_of_interest(self):
        print("Rate of Interest is 5%")

# Child Class
class SBI(Bank):
    def rate_of_interest(self):
        print("Rate of Interest is 7%")

obj = SBI()
obj.rate_of_interest()
```

Now -

Problem with User-Defined Classes:-

```
class A:
    def __init__(self, a):
        self.a = a

obj1 = A(10)
obj2 = A(20)
print(obj1 + obj2)   # Error

# If we check:
dir(A)

# There is no __add__ method, so + cannot work.
```

Operator Overloading (Polymorphism)

```
a = 10
b = 20
print(a + b)   # 30
# Here, a and b are objects of inbuilt class int, so + works.
```

Objects of Inbuilt Classes

Inbuilt datatypes support operators because they have **magic methods**.

Example:

Inbuilt datatypes support operators because they have **magic methods**.

Example:

dir(int)

Output contains:	__add__, __sub__, __mul__, ...

Because of these **special (magic) methods**, operators work on inbuilt objects.

Solution: Operator Overloading

```
# Overloading + Operator (__add__)
class A:
    def __init__(self, a):
        self.a = a

    def __add__(self, other):
        return self.a + other.a

obj1 = A(10)
obj2 = A(20)

print(obj1 + obj2)   # Internally calls obj1.__add__(obj2)
print(obj1 - obj2)   # Error
```

Problem Statement:- Polymorphism and Inheritance in a Vehicle Rental System

Problem Statement:- Polymorphism and Inheritance in a Vehicle Rental System

Design a **Vehicle Rental System** using **Inheritance and Polymorphism**, where different types of vehicles calculate rental charges differently based on their category.

System Requirements:-

1. Inheritance

i. Create a base class Vehicle

- Attributes:
  - vehicle_id
  - brand
- Method:
  - calculate_rent()

ii. Hierarchical Inheritance

Create the following classes inheriting from Vehicle:

i. **Car**
   - Attribute: price_per_day
   - Overrides calculate_rent() to compute rent based on number of days

ii. **Bike**
   - Attribute: price_per_hour
   - Overrides calculate_rent() to compute rent based on number of hours

iii. **Truck**
   - Attribute: price_per_km
   - Overrides calculate_rent() to compute rent based on distance travelled

2. Polymorphism

Run-Time Polymorphism (Method Overriding)

- The method calculate_rent() is defined in the base class Vehicle
- Each derived class (Car, Bike, Truck) provides its **own implementation**
- At runtime, the method call depends on the **object type**, not the reference type