Methods in OOP

In Object-Oriented Programming (OOP), a **method** is a function that is associated with an object and defined inside a class. Methods can operate on data that belongs to the object (attributes) or accept additional arguments to perform a specific task.

Types of Methods:

1. **Instance Methods**:

   - These methods are the most common and are called on instances (objects) of the class.

   - They require the **`self`** parameter, which represents the instance calling the method.

2. **Class Methods**:

   - These methods take **`cls`** (class itself) as the first argument and are bound to the class rather than the object.

   - Decorated with **`@classmethod`**.

3. **Static Methods**:

   - These methods do not take `self` or `cls` as the first argument and can be called on the class or object directly.

   - Decorated with **`@staticmethod`**.

Constructor in OOP

A **constructor** is a special type of method that is called automatically when an object of the class is created. In Python, the constructor is defined by the `__init__()` method.

- **`__init__()` method**:

   - Automatically called when an object is created.

   - Used to initialize the attributes of the object.

`self` Keyword

The `self` keyword represents the instance of the class. It allows access to the attributes and methods of the class within instance methods.

Example with Explanation

```python
class Car:
    # Constructor - automatically called when object is created
    def __init__(self, brand, model):
        # Instance variables (attributes) are always prefixed with 'self'
        self.brand = brand  # 'self.brand' refers to the attribute of the object
        self.model = model

    # Instance method - can access instance attributes
    def show_details(self):
        print(f"Car brand: {self.brand}, Model: {self.model}")

    # Instance method with additional arguments
    def drive(self, speed):
        print(f"The {self.brand} {self.model} is driving at {speed} km/h")

    # Class method - can access class attributes but not instance attributes
    @classmethod
    def car_type(cls):
        return "Vehicle"

    # Static method - does not access class or instance attributes
    @staticmethod
    def info():
        print("This is a car class")
```

```
# Creating an object of the Car class
car1 = Car("Toyota", "Corolla")  # __init__ method is called automatically
car2 = Car("Honda", "Civic")

# Calling instance methods
car1.show_details()   # Outputs: Car brand: Toyota, Model: Corolla
car2.show_details()   # Outputs: Car brand: Honda, Model: Civic

# Calling instance method with additional argument
car1.drive(80)  # Outputs: The Toyota Corolla is driving at 80 km/h

# Calling class method
print(Car.car_type())  # Outputs: Vehicle

# Calling static method
Car.info()  # Outputs: This is a car class
```

Key Points:

1. **Constructor (`__init__`)**:
   - Automatically called when creating an object, used to initialize attributes like `self.brand` and `self.model`.
   - Here, `self.brand` and `self.model` are used to define the properties of each object separately.

2. **Methods**:
   - **Instance Methods** (e.g., `show_details`, `drive`) use the `self` parameter to access and modify object attributes.
   - When calling `car1.drive(80)`, the `speed` argument is passed directly into the method.

3. **Arguments**:

- Inside methods, additional arguments like `speed` can be passed directly when calling the method, while `self` is automatically passed when the method is called on an object (like `car1` or `car2`).

4. **Static and Class Methods**:

- Class methods use `cls` to access class-level information.

- Static methods do not need `self` or `cls` as they don't operate on object-specific or class-specific data.

In summary, methods in OOP allow objects to interact with their own data (via `self`) and perform operations based on external inputs or arguments provided during method calls.

In Python, the constructor is defined using the special method `__init__(self)`. This method is automatically called when a new object is created from a class. The constructor is used to initialize the object's attributes.

Here are two examples to illustrate the constructor:

Example 1: Constructor to Initialize Car Attributes

```python
class Car:
    # Constructor - called automatically when object is created
    def __init__(self, brand, model, year):
        # Initializing instance variables (attributes)
        self.brand = brand
        self.model = model
        self.year = year

    # Method to display car details
    def show_details(self):
        print(f"Brand: {self.brand}, Model: {self.model}, Year: {self.year}")

# Creating an object of the Car class
```

```
car1 = Car("Toyota", "Corolla", 2020)  # __init__ method is called
car2 = Car("Honda", "Civic", 2022)

# Accessing attributes
car1.show_details()  # Output: Brand: Toyota, Model: Corolla, Year: 2020
car2.show_details()  # Output: Brand: Honda, Model: Civic, Year: 2022
```

In this example:
- `__init__(self, brand, model, year)` is the constructor.
- It initializes the attributes `brand`, `model`, and `year` for each car object when created.

Example 2: Constructor to Initialize Bank Account Details

```python
class BankAccount:
    # Constructor - called when object is created
    def __init__(self, account_holder, balance=0):
        # Initializing instance variables (attributes)
        self.account_holder = account_holder
        self.balance = balance  # Default balance is 0

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

    # Method to display account details
    def show_account_info(self):
        print(f"Account Holder: {self.account_holder}, Balance: {self.balance}")

# Creating objects of BankAccount class
account1 = BankAccount("John Doe")  # No initial deposit, balance defaults to 0
```

account2 = BankAccount("Jane Smith", 500) # Initial deposit of 500

Accessing attributes and methods

account1.show_account_info() # Output: Account Holder: John Doe, Balance: 0

account2.show_account_info() # Output: Account Holder: Jane Smith, Balance: 500

Depositing money

account1.deposit(200) # Output: Deposited 200. New balance: 200

```

In this example:

- `__init__(self, account_holder, balance=0)` is the constructor, initializing the `account_holder` and `balance` attributes. The balance has a default value of 0.

### Inheritance in Python

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) where a new class (called a **child** or **derived** class) inherits properties and methods from another class (called a **parent** or **base** class). This allows code reuse and helps in creating hierarchical relationships between classes.

#### Key Types of Inheritance:

1. **Single Inheritance**: One child class inherits from one parent class.

2. **Multiple Inheritance**: One child class inherits from more than one parent class.

Python supports **multiple inheritance**, meaning a class can inherit from more than one parent class. This is not supported in languages like Java and PHP.

### Example 1: Single Inheritance

In **single inheritance**, a child class inherits from a single parent class.

```python

# Parent class

class Animal:

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

    def speak(self):
        print(f"{self.name} makes a sound.")

# Child class inheriting from Animal (single inheritance)
class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks.")

# Creating an instance of Dog class
dog = Dog("Buddy") # Inherits from Animal
dog.speak() # Output: Buddy barks
```

In this example:

- The `Dog` class inherits from the `Animal` class. It overrides the `speak()` method of the parent class to provide a more specific implementation for dogs.

Example 2: Multiple Inheritance

In **multiple inheritance**, a child class can inherit from multiple parent classes.

```python
# Parent class 1
class Engine:
    def start_engine(self):
        print("Engine started.")

# Parent class 2
```

```
class FuelSystem:
    def refuel(self):
        print("Fuel system refueled.")

# Child class inheriting from both Engine and FuelSystem (multiple inheritance)
class Car(Engine, FuelSystem):
    def drive(self):
        print("Car is driving.")

# Creating an instance of the Car class
my_car = Car()
my_car.start_engine()  # Inherits from Engine class
my_car.refuel()        # Inherits from FuelSystem class
my_car.drive()         # Car's own method

# Output:
# Engine started.
# Fuel system refueled.
# Car is driving.
```

In this example:

- The `Car` class inherits from both the `Engine` and `FuelSystem` classes, demonstrating **multiple inheritance**. The `Car` class can now access methods from both parent classes, i.e., `start_engine()` from `Engine` and `refuel()` from `FuelSystem`.

Key Points:

1. **Single Inheritance**: One parent, one child. The child class can inherit and override methods from the parent class.

2. **Multiple Inheritance**: A class can inherit from multiple parent classes, gaining access to methods and properties from all parent classes. This is supported in Python, but not in Java or PHP.

Encapsulation in Python

Encapsulation is a concept in object-oriented programming (OOP) where the internal details (data and methods) of an object are hidden, and access to them is restricted. In Python, this is achieved by making the attributes or methods **private**, meaning they cannot be accessed directly outside the class. Instead, you use **getter** and **setter** methods to access and modify these private variables.

- **Private Variables**: In Python, you can make a variable private by prefixing it with two underscores (`__`).
- **Getter Method**: Used to retrieve the value of a private variable.
- **Setter Method**: Used to modify the value of a private variable.

Example 1: Encapsulation with Constructor (Using Getter and Setter)

```python
class Employee:
    def __init__(self, name, salary):
        # Private attributes
        self.__name = name
        self.__salary = salary

    # Getter for name
    def get_name(self):
        return self.__name

    # Setter for name
    def set_name(self, name):
        self.__name = name

    # Getter for salary
    def get_salary(self):
        return self.__salary
```

```
    # Setter for salary
    def set_salary(self, salary):
        if salary > 0:
            self.__salary = salary
        else:
            print("Invalid salary amount")

# Creating an object
emp = Employee("John", 50000)

# Accessing private variables using getters
print(emp.get_name())   # Output: John
print(emp.get_salary())  # Output: 50000

# Modifying private variables using setters
emp.set_name("Alice")
emp.set_salary(60000)
print(emp.get_name())   # Output: Alice
print(emp.get_salary())  # Output: 60000

# Trying to set an invalid salary
emp.set_salary(-1000)  # Output: Invalid salary amount
```

In this example:

- **`__name`** and **`__salary`** are private attributes.

- The class provides **getter** methods (`get_name()`, `get_salary()`) to access these attributes and **setter** methods (`set_name()`, `set_salary()`) to modify them.

Example 2: Encapsulation Without Constructor (Using Getter and Setter)

```python
class Student:
    # Private attribute
    __grade = None

    # Setter for grade
    def set_grade(self, grade):
        if 0 <= grade <= 100:
            self.__grade = grade
        else:
            print("Invalid grade")

    # Getter for grade
    def get_grade(self):
        return self.__grade

# Creating an object
student = Student()

# Setting grade using setter
student.set_grade(85)

# Getting grade using getter
print(student.get_grade())  # Output: 85

# Trying to set an invalid grade
student.set_grade(150)  # Output: Invalid grade
```

In this example:

- The private variable **`__grade`** is managed via getter and setter methods, and no constructor is used.

Example 3: Encapsulation with Validation in Setter

```python
class Product:
    def __init__(self, name, price):
        self.__name = name
        self.__price = price

    # Getter for name
    def get_name(self):
        return self.__name

    # Setter for name
    def set_name(self, name):
        self.__name = name

    # Getter for price
    def get_price(self):
        return self.__price

    # Setter for price with validation
    def set_price(self, price):
        if price > 0:
            self.__price = price
        else:
            print("Price must be positive")

# Creating a Product object
```

```
product = Product("Laptop", 1200)

# Accessing and modifying price through setter and getter
print(product.get_name())    # Output: Laptop
print(product.get_price())   # Output: 1200

product.set_price(1500)      # Updating price
print(product.get_price())   # Output: 1500

product.set_price(-200)      # Invalid price
# Output: Price must be positive
```

In this example, the **setter** method for `__price` includes validation to ensure the price is positive, demonstrating encapsulation with validation logic.

Example 4: Encapsulation in a Banking Application Without Constructor

```python
class BankAccount:
    __balance = 0  # Private attribute

    # Getter for balance
    def get_balance(self):
        return self.__balance

    # Setter for balance with validation
    def set_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
```

```
        print("Invalid balance")

# Creating an object
account = BankAccount()

# Setting and accessing the balance through setter and getter
account.set_balance(1000)
print(account.get_balance())  # Output: 1000

account.set_balance(-500)     # Output: Invalid balance
```

In this example:

- The class uses a private variable **`__balance`** that is controlled via setter and getter methods without using a constructor.

Key Points:

1. **Private Variables**: Defined using a double underscore (`__`), private variables cannot be accessed directly from outside the class.

2. **Getter and Setter Methods**: Provide controlled access to private attributes. Getters retrieve values, while setters modify them with optional validation.

3. **Encapsulation**: Encapsulation ensures data is hidden and protected from direct access, while methods like getters and setters control how the data is accessed and modified.

Polymorphism in Python

Polymorphism refers to the ability of different objects to respond to the same function or method call in their own unique way. In Python, this can be achieved through method overriding (inherited classes), and in some ways, it simulates function overloading through default arguments or multiple method definitions.

Types of Polymorphism:

1. **Function Overloading** (Not natively supported in Python): A single function can handle different types or numbers of arguments.

2. **Function Overriding**: A child class provides a specific implementation of a method that is already defined in its parent class.

Function Overloading in Python

Although Python does not natively support function overloading like C++ or Java, we can mimic it using default parameters, `*args`, or `**kwargs`.

Example 1: Function Overloading Using Default Arguments

```python
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c

calc = Calculator()
print(calc.add(2, 3))      # Output: 5 (adding two arguments)
print(calc.add(2, 3, 4))   # Output: 9 (adding three arguments)
print(calc.add(5))         # Output: 5 (adding one argument, b and c default to 0)
```

- The `add` function is overloaded in a way by using default arguments (`b=0`, `c=0`). Based on the number of arguments passed, different behaviors are exhibited.

Example 2: Function Overloading Using `*args` (Variable Length Arguments)

```python
class Calculator:
    def add(self, *args):
        return sum(args)

calc = Calculator()
print(calc.add(1, 2))      # Output: 3
print(calc.add(1, 2, 3))    # Output: 6
```

print(calc.add(1, 2, 3, 4)) # Output: 10

```

- The `add` method can accept any number of arguments using `*args` and calculates the sum, effectively simulating function overloading.

### Function Overriding in Python

Function overriding happens when a child class defines a method with the same name as a method in its parent class. The child class's method overrides the parent class's method.

#### Example 3: Function Overriding (Single Inheritance)

```python
class Animal:
    def sound(self):
        return "Some sound"

class Dog(Animal):
    def sound(self):
        return "Bark"

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

# Creating objects
dog = Dog()
cat = Cat()

print(dog.sound())  # Output: Bark (overridden method)
print(cat.sound())  # Output: Meow (overridden method)
```
```

- The `sound` method is overridden in the `Dog` and `Cat` classes, changing its behavior depending on the object type.

Example 4: Function Overriding (Multi-level Inheritance)

```python
class Vehicle:
    def start(self):
        return "Vehicle started"

class Car(Vehicle):
    def start(self):
        return "Car started"

class ElectricCar(Car):
    def start(self):
        return "Electric car started silently"

# Creating objects
vehicle = Vehicle()
car = Car()
electric_car = ElectricCar()

print(vehicle.start())       # Output: Vehicle started
print(car.start())           # Output: Car started
print(electric_car.start())  # Output: Electric car started silently
```

- The `start` method is overridden in both the `Car` and `ElectricCar` classes, showing how method overriding can work in multi-level inheritance.

Example 5: Polymorphism with Functions

```python
class Rectangle:
    def area(self, length, width):
        return length * width

class Circle:
    def area(self, radius):
        return 3.1416 * radius * radius

# A single function can work with different object types
def print_area(shape):
    if isinstance(shape, Rectangle):
        print(f"Rectangle area: {shape.area(5, 3)}")
    elif isinstance(shape, Circle):
        print(f"Circle area: {shape.area(7)}")

# Polymorphism in action
rect = Rectangle()
circle = Circle()

print_area(rect)    # Output: Rectangle area: 15
print_area(circle)  # Output: Circle area: 153.9384
```

- Here, the `print_area` function works polymorphically, meaning it accepts different types of objects (`Rectangle` or `Circle`) and calls the correct method (`area`) depending on the object passed.

Example 6: Polymorphism with Method Overriding in Multiple Classes

```python
class Bird:
    def fly(self):
```

```
        print("Flying in the sky")

class Sparrow(Bird):
    def fly(self):
        print("Sparrow flying at low altitude")

class Eagle(Bird):
    def fly(self):
        print("Eagle soaring high")

# Creating objects of Sparrow and Eagle
sparrow = Sparrow()
eagle = Eagle()

# Polymorphism in action: the same method 'fly' behaves differently
for bird in [sparrow, eagle]:
    bird.fly()

# Output:
# Sparrow flying at low altitude
# Eagle soaring high
```

- This shows polymorphism in action where different objects (`Sparrow` and `Eagle`) can invoke the same method (`fly`) but with different behaviors.

Example 7: Operator Overloading (A Type of Polymorphism)

Python allows **operator overloading**, meaning we can define the behavior of operators (`+`, `-`, etc.) for user-defined types. This is another form of polymorphism.

```python
```

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Overloading the '+' operator
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

# Creating two points
p1 = Point(2, 3)
p2 = Point(4, 5)

# Adding two Point objects using the overloaded '+' operator
p3 = p1 + p2  # Equivalent to p1.__add__(p2)
print(p3)     # Output: Point(6, 8)
```

- This is **operator overloading** where the `+` operator is overloaded to work with `Point` objects. Polymorphism allows the `+` operator to behave differently based on the operands.

Summary:

- **Polymorphism** allows objects of different classes to be treated as objects of a common base class, leading to more flexible and extensible code.
- **Function Overloading** (mimicked in Python using default arguments or `*args`) allows a function to handle different numbers or types of arguments.
- **Function Overriding** allows a child class to redefine a method that exists in its parent class to suit its own behavior.

- **Operator Overloading** is another form of polymorphism, where operators like `+`, `-`, etc., can be overloaded to work with user-defined types.

These concepts promote code reusability and flexibility in object-oriented programming.

Why Do We Use the `super()` Keyword?

The `super()` keyword in Python is used to call methods from a parent (or superclass) inside a child (subclass) class. It plays an important role in **inheritance** and **polymorphism**, helping avoid redundancy and allowing for **code reuse**.

The main reasons to use `super()` include:

1. **Inheritance**: To access methods and properties of a parent class.

2. **Method Overriding**: To allow a child class to extend or modify the behavior of its parent class's method.

3. **Multiple Inheritance**: To resolve the **Method Resolution Order (MRO)** when dealing with multiple inheritance.

How `super()` Works in Inheritance

When a class inherits from a parent class, `super()` allows the child class to call the methods or constructors of its parent class without directly referring to the parent class by name. This is especially useful for multiple inheritance, where the `super()` function helps in determining the correct class in the MRO.

Example 1: Using `super()` to Call the Parent Class Constructor

```python
class Parent:
    def __init__(self, name):
        self.name = name
        print(f"Parent class constructor: {self.name}")

class Child(Parent):
```

```
    def __init__(self, name, age):
        super().__init__(name)  # Calls the Parent class constructor
        self.age = age
        print(f"Child class constructor: {self.name}, Age: {self.age}")

# Creating an object of Child class
child_obj = Child("Alice", 12)

# Output:
# Parent class constructor: Alice
# Child class constructor: Alice, Age: 12
```

In this example, `super().__init__(name)` calls the constructor of the `Parent` class, which initializes the `name` attribute. Then, the `Child` class constructor continues its own process, initializing the `age` attribute.

Example 2: Using `super()` to Call Parent Class Methods

```python
class Parent:
    def show(self):
        print("This is the Parent class method")

class Child(Parent):
    def show(self):
        super().show()  # Calls the Parent class's 'show' method
        print("This is the Child class method")

# Creating an object of Child class
child_obj = Child()
child_obj.show()
```

Output:

This is the Parent class method

This is the Child class method

```

In this example, the `show()` method of the `Child` class calls the `show()` method of the `Parent` class using `super().show()`. This allows the child class to extend the behavior of the parent class method, a common use case in **method overriding**.

### How `super()` Works in Polymorphism

`super()` is crucial in **polymorphism** because it allows a child class to call methods from its parent class while modifying or extending the behavior of those methods. This is especially useful in **method overriding**, where a child class wants to add functionality to the parent class's method.

#### Example 3: Polymorphism with Method Overriding Using `super()`

```python
class Animal:
    def sound(self):
        return "Some generic sound"

class Dog(Animal):
    def sound(self):
        parent_sound = super().sound()  # Calls the Animal's sound method
        return f"{parent_sound}, but Dog says Bark!"

class Cat(Animal):
    def sound(self):
        parent_sound = super().sound()  # Calls the Animal's sound method
        return f"{parent_sound}, but Cat says Meow!"
```
```

```
# Polymorphism in action
animals = [Dog(), Cat()]

for animal in animals:
    print(animal.sound())

# Output:
# Some generic sound, but Dog says Bark!
# Some generic sound, but Cat says Meow!
```

In this case, both `Dog` and `Cat` override the `sound()` method of the `Animal` class. They use `super().sound()` to call the parent class method and then extend it with their own specific sounds.

How `super()` Works in Multiple Inheritance

In Python, `super()` also helps resolve the **Method Resolution Order (MRO)** when multiple inheritance is involved. This ensures that the parent classes are called in a specific order.

Example 4: `super()` in Multiple Inheritance

```python
class A:
    def __init__(self):
        print("Class A Constructor")

class B(A):
    def __init__(self):
        super().__init__()  # Calls A's constructor
        print("Class B Constructor")
```

```
class C(A):
    def __init__(self):
        super().__init__()  # Calls A's constructor
        print("Class C Constructor")

class D(B, C):  # Multiple inheritance
    def __init__(self):
        super().__init__()  # Calls the constructor of B and C following MRO
        print("Class D Constructor")

# Creating an object of D
d = D()

# Output (following MRO):
# Class A Constructor
# Class C Constructor
# Class B Constructor
# Class D Constructor
```

In this example:

- Class `D` inherits from both `B` and `C`, and `B` and `C` both inherit from `A`.
- The `super()` function follows the Method Resolution Order (MRO) to call the constructors in the correct order.

The `super()` call in `D` calls `B`'s constructor, which in turn calls `C`'s constructor (via MRO), and finally, `A`'s constructor is called.

Benefits of Using `super()`

1. **Simplifies code**: You don't have to refer to the parent class by name, making the code more maintainable.

2. **Avoids repetition**: You avoid repeating code present in the parent class.

3. **Multiple inheritance**: It resolves MRO in multiple inheritance scenarios.

4. **Enables polymorphism**: Allows a child class to extend or modify the parent class's methods efficiently.

Summary:

- The `super()` keyword is used to call methods from the parent class inside the child class.

- It is used to facilitate **inheritance**, allowing a child class to reuse or modify the behavior of the parent class.

- `super()` is useful in **method overriding**, where a child class can extend the behavior of a parent method.

- In **multiple inheritance**, `super()` helps resolve the **Method Resolution Order (MRO)** to ensure methods are called in the correct order.

Method Overloading vs. Method Overriding

Both **method overloading** and **method overriding** are forms of **polymorphism**, allowing methods to behave differently based on context. However, they differ in how they operate and where they're applied.

---

1. **Method Overloading**

- **Definition**: In method overloading, multiple methods with the same name exist in the same class, but they differ in the number or type of parameters. It allows a class to have more than one method with the same name but different signatures.

- **Support in Python**: Python **does not** natively support method overloading like Java or C++. However, you can achieve similar functionality using default arguments or by inspecting arguments inside a method.

Example of Method Overloading Using Default Arguments

```python
class MathOperations:
    def add(self, a, b=0, c=0):
        return a + b + c

# Creating an instance
math_op = MathOperations()

# Calling the overloaded 'add' method
print(math_op.add(2))       # Calls add(2, 0, 0) -> Output: 2
print(math_op.add(2, 3))    # Calls add(2, 3, 0) -> Output: 5
print(math_op.add(2, 3, 4))  # Calls add(2, 3, 4) -> Output: 9
```

- In this example, the method `add()` can take 1, 2, or 3 arguments, and behaves differently depending on how many arguments are provided.

Example of Overloading Using Variable-Length Arguments

You can also use `*args` or `**kwargs` to simulate overloading:

```python
class MathOperations:
    def add(self, *args):
        return sum(args)

math_op = MathOperations()

print(math_op.add(2))        # Output: 2
print(math_op.add(2, 3))      # Output: 5
```

```
print(math_op.add(2, 3, 4))    # Output: 9
```

- In this case, the method can accept any number of arguments and sums them.

---

2. **Method Overriding**

- **Definition**: In method overriding, a method in a **child class** has the same name, return type, and parameters as a method in the **parent class**. The child class **overrides** the parent class's method to provide its own specific implementation.

- **Support in Python**: Method overriding is fully supported in Python and is a key feature in **inheritance** and **polymorphism**.

Example of Method Overriding

```python
class Animal:
    def sound(self):
        print("Animal makes a sound")

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

# Creating objects
animal = Animal()
dog = Dog()

# Calling the overridden method
```

```
animal.sound()  # Output: Animal makes a sound
dog.sound()     # Output: Dog barks
```

In this example:

- The `sound()` method in the `Dog` class **overrides** the `sound()` method in the `Animal` class.
- The child class provides its own implementation of the `sound()` method, while keeping the method name the same.

Key Differences Between Method Overloading and Method Overriding

Aspect	Method Overloading	Method Overriding
Definition	Multiple methods with the same name but different signatures (parameters).	Child class method with the same name, parameters, and return type as the parent class method.
Purpose	Achieve method behavior based on different parameter inputs.	Modify or extend the behavior of a parent class method in the child class.
Inheritance	Not related to inheritance; occurs within the same class.	Happens in a class hierarchy involving inheritance.
Support in Python	Python does not directly support it (can be achieved using default or variable arguments).	Fully supported in Python using inheritance.
Method Signature	Method names are the same, but the number/type of arguments differs.	The method in the child class has the same signature (name, parameters) as the parent class method.
Compile Time or Run Time	Resolved at **compile time** (in languages like Java).	Resolved at **runtime** (runtime polymorphism).

Polymorphism and Method Overriding Example

Let's look at how **method overriding** works in the context of polymorphism, where multiple objects can use the same interface (method name) but exhibit different behaviors.

```python
class Animal:
    def sound(self):
        return "Some sound"

class Dog(Animal):
    def sound(self):
        return "Bark"

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

# Polymorphism in action
animals = [Dog(), Cat(), Animal()]

for animal in animals:
    print(animal.sound())

# Output:
# Bark
# Meow
# Some sound
```

In this case, each object (`Dog`, `Cat`, and `Animal`) uses the same `sound()` method, but they produce different results depending on the object type.

Summary

- **Method Overloading**: Same method name, different parameter signatures. Python doesn't support it directly but it can be achieved using default arguments or variable-length arguments (`*args`, `**kwargs`).

- **Method Overriding**: Same method signature in both the parent and child class, but the child class provides its own specific implementation of the method. Fully supported in Python and crucial for inheritance and polymorphism.

Both of these concepts are essential for creating flexible, reusable code in object-oriented programming.

Applications of Polymorphism and Inheritance

1. **Polymorphism Applications**

Polymorphism allows different types of objects to be treated as instances of the same class through a shared interface. It enables methods to behave differently based on the object that calls them, which is useful in a variety of applications:

- **1.1. Implementing Interface/Abstract Classes**

  - In large applications, interfaces or abstract base classes define common methods that various subclasses can implement in their own way.

  - Example: A `Shape` class can have an abstract method `draw()`, and concrete classes like `Circle`, `Square`, and `Rectangle` will override the `draw()` method to handle their specific rendering logic.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass
```

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

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

# Polymorphism in action
shapes = [Circle(), Square()]
for shape in shapes:
    shape.draw()
```

- **1.2. Code Reusability and Flexibility**

  - Polymorphism allows writing reusable code by defining functions that can work with objects of multiple types. This makes the code more flexible and adaptable to changes.

  - Example: You can write a function that accepts an object of any subclass and calls methods without needing to know the exact type of the object.

```python
def print_sound(animal):
    print(animal.sound())

class Dog:
    def sound(self):
        return "Bark"

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

```
# Polymorphism through function
print_sound(Dog())  # Output: Bark
print_sound(Cat())  # Output: Meow
```

- **1.3. Operator Overloading**

  - Polymorphism can be used to overload operators, allowing objects of custom types to interact using built-in operators like `+`, `-`, or `*`.

  - Example: Implementing vector addition for a `Vector` class.

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2  # Uses overloaded + operator
print(v3)  # Output: Vector(4, 6)
```

- **1.4. GUI Frameworks and Widget Classes**

- Polymorphism is widely used in GUI frameworks. A base `Widget` class can be overridden by different types of widgets like buttons, sliders, text fields, etc., allowing the framework to treat all widgets uniformly while rendering or processing events.

- **1.5. Game Development**

- In game development, polymorphism is used to handle different types of in-game objects (e.g., enemies, power-ups, obstacles) using a common interface. Each object type can have its own behavior while still being processed by common game logic.

---

2. **Inheritance Applications**

Inheritance allows a class to inherit attributes and methods from another class, promoting code reuse and creating hierarchies.

- **2.1. Code Reusability**

- Inheritance is used to promote the reusability of code. A base class contains common functionality that can be shared by multiple subclasses.

- Example: In an organization, a `Person` class can be used as the parent class for both `Employee` and `Manager` classes. The common properties (name, age) and methods (get details) can be inherited, while specific methods like `manage()` or `work()` can be defined in the subclasses.

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

    def get_details(self):
        return f"Name: {self.name}, Age: {self.age}"

class Employee(Person):
    def work(self):
        return f"{self.name} is working"
```

```
class Manager(Person):
    def manage(self):
        return f"{self.name} is managing"

# Creating objects
emp = Employee("Alice", 30)
mgr = Manager("Bob", 45)

# Using inherited methods
print(emp.get_details())  # Output: Name: Alice, Age: 30
print(mgr.get_details())  # Output: Name: Bob, Age: 45
```

- **2.2. Extending Functionality**
  - Inheritance allows subclasses to extend the functionality of a parent class without modifying its code. This is particularly useful in frameworks and libraries.
  - Example: A `Vehicle` class can have subclasses like `Car`, `Bike`, and `Truck`, each with additional functionality (e.g., `Car` has a `drive()` method).

```python
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def start_engine(self):
        return f"{self.brand} engine started"

class Car(Vehicle):
    def drive(self):
        return f"Driving a {self.brand} car"
```

```
# Creating a Car object
my_car = Car("Toyota")
print(my_car.start_engine())  # Output: Toyota engine started
print(my_car.drive())        # Output: Driving a Toyota car
```

- **2.3. Object-Oriented Design Patterns**

  - Inheritance is a key component of many design patterns, such as **Factory**, **Singleton**, **Observer**, etc. These patterns promote the use of base classes and subclasses to implement flexible and maintainable code.

- **2.4. Frameworks and Libraries**

  - Many popular frameworks (e.g., Django, Flask, and web frameworks) rely on inheritance to allow users to extend and modify their functionality. For example, in Django, you can create custom views by inheriting from a base `View` class.

- **2.5. User Interface (UI) Components**

  - Inheritance is commonly used to create custom UI components in GUI libraries. A `Button` class can inherit common features like click handling from a base `Widget` class while adding specific behavior like displaying text.

- **2.6. Security and Access Control**

  - In systems where different users have different roles, inheritance can be used to define access levels. For example, a `User` class can have subclasses like `Admin`, `Editor`, and `Viewer`, each with specific permissions and functionalities.

```python
class User:
    def __init__(self, username):
        self.username = username

    def access(self):
        return f"{self.username} has limited access"
```

```
class Admin(User):
    def access(self):
        return f"{self.username} has admin access"

class Editor(User):
    def access(self):
        return f"{self.username} can edit content"

# Creating users with different access levels
user = User("user123")
admin = Admin("admin123")
editor = Editor("editor123")

print(user.access())   # Output: user123 has limited access
print(admin.access())  # Output: admin123 has admin access
print(editor.access()) # Output: editor123 can edit content
```

Summary of Applications

Concept	Applications
Polymorphism	1. Abstract classes and interfaces
	2. Code reusability and flexibility
	3. Operator overloading
	4. GUI frameworks
	5. Game development
Inheritance	1. Code reusability
	2. Extending functionality

	3. Object-oriented design patterns
	4. Frameworks and libraries
	5. UI component development
	6. Security and access control

Both polymorphism and inheritance are fundamental concepts that enable efficient code reuse, extensibility, and scalability in object-oriented programming.