Inheritance and Polymorphism

October 10, 2025

Inheritance

Objective: To demonstrate the principle of inheritance by creating a Dog class that inherits from a parent Animal class, thus reusing its methods and attributes.

Theory: Inheritance is a mechanism where a new class (child class) inherits the attributes and methods of an existing class (parent class). This promotes code reuse and creates a logical hierarchy. The super() function is often used to call the parent class's constructor (__init__) and other methods.

Types of Inheritance in Python:

- **Single Inheritance:** A child class inherits from a single parent class.
- **Multiple Inheritance:** A child class inherits from multiple parent classes.
- **Multilevel Inheritance:** A class inherits from a child class, which itself inherits from another parent class, forming a chain of inheritance.
- **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class.
- **Hybrid Inheritance:** A combination of two or more types of inheritance.

1. Single Inheritance

In single inheritance, a child class inherits from just one parent class.

Example: This example shows a child class Employee inheriting a property from the parent class Person.

```
class Person:
    def __init__(self, name):
        self.name = name
class Employee(Person):  # Employee inherits from Person
    def show_role(self):
        print(self.name, "is an employee")
emp = Employee("Sarah")
print("Name:", emp.name)
emp.show_role()
```

Output

```
Name: Sarah
Sarah is an employee
```

2. Multiple Inheritance

In multiple inheritance, a child class can inherit from more than one parent class.

Example: This example demonstrates Employee inheriting properties from two parent classes: Person and Job.

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


class Job:
    def __init__(self, salary):
        self.salary = salary


class Employee(Person, Job):  # Inherits from both Person and Job
    def __init__(self, name, salary):
        Person.__init__(self, name)
        Job.__init__(self, salary)


    def details(self):
        print(self.name, "earns", self.salary)


emp = Employee("Jennifer", 50000)
emp.details()
```

Output

```
Jennifer earns 50000
```

3. Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class (like a chain).

Example: This example shows Manager inheriting from Employee, which in turn inherits from Person.

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


class Employee(Person):
    def show_role(self):
        print(self.name, "is an employee")


class Manager(Employee):  # Manager inherits from Employee
    def department(self, dept):
        print(self.name, "manages", dept, "department")


mgr = Manager("Joy")
mgr.show_role()
mgr.department("HR")
```

Output

```
Joy is an employee
Joy manages HR department
```

4. Hierarchical Inheritance

In hierarchical inheritance, multiple child classes inherit from the same parent class.

Example: This example demonstrates two child classes (Employee and Intern) inheriting from a single parent class Person.

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


class Employee(Person):
    def role(self):
        print(self.name, "works as an employee")


class Intern(Person):
    def role(self):
        print(self.name, "is an intern")


emp = Employee("David")
emp.role()


intern = Intern("Eva")
intern.role()
```

Output

```
David works as an employee
Eva is an intern
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.

Example: This example demonstrates TeamLead inheriting from both Employee (which inherits Person) and Project, combining multiple inheritance types.

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

class Employee(Person):
    def role(self):
        print(self.name, "is an employee")

class Project:
    def __init__(self, project_name):
        self.project_name = project_name

class TeamLead(Employee, Project):  # Hybrid Inheritance
    def __init__(self, name, project_name):
        Employee.__init__(self, name)
        Project.__init__(self, project_name)

    def details(self):
        print(self.name, "leads project:", self.project_name)

lead = TeamLead("Sophia", "AI Development")
lead.role()
lead.details()
```

Output

```
Sophia is an employee
Sophia leads project: AI Development
```

Polymorphism

Objective: To understand polymorphism by creating different classes that share the same method name but have different implementations.

Theory: Polymorphism (meaning "many forms") allows objects of different classes to be treated as objects of a common base class. This is often achieved through method overriding, where a child class provides its own unique implementation for a method already defined in its parent class.

Key aspects of polymorphism in Python:

Method Overriding (Polymorphism with Inheritance):

- Subclasses can provide their own specific implementation of methods already defined in their superclass.
- When a method is called on an object, the version of the method in the object's actual class (or its closest ancestor) is executed.

Program:

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

class Dog(Animal):
    def speak(self):
        print("Woof, woof!")

class Cat(Animal):
    def speak(self):
        print("Meow, meow!")

def make_sound(animal):
    animal.speak()

dog = Dog()
cat = Cat()

make_sound(dog) # Output: Woof, woof!
make_sound(cat) # Output: Meow, meow!
```

Output:

```
Woof, woof!
Meow, meow!
```

Polymorphism with Functions and Objects (Duck Typing):

- Python's dynamic typing system allows functions to operate on objects of different types as long as those objects provide the necessary methods or attributes. This is often referred to as "duck typing" – "If it walks like a duck and quacks like a duck, then it must be a duck."

- There is no need for explicit inheritance or interfaces; simply having the required method is sufficient.

Program:

```
class Duck:
    def fly(self):
        print("Duck flying")

class Plane:
    def fly(self):
        print("Plane flying")

class Superman:
    def fly(self):
        print("Superman flying")

def make_it_fly(entity):
    entity.fly()

duck = Duck()
plane = Plane()
superman = Superman()

make_it_fly(duck)     # Output: Duck flying
make_it_fly(plane)    # Output: Plane flying
make_it_fly(superman) # Output: Superman flying
```

Output:

```
Duck flying
Plane flying
Superman flying
```