Basic Programming with Python

Prepared By:
Professor Dr. Md. Mijanur Rahman
Department of Computer Science & Engineering
Jatiya Kabi Kazi Nazrul Islam University, Bangladesh.
www.mijanrahman.com

8

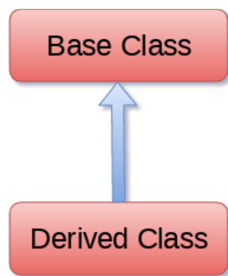
OOP Concepts in Python

CONTENTS

8.10. Inheritance in Python 1
- 8.10.1 Implementation of Inheritance in Python 2
- 8.10.2 Types of Inheritances in Python 4
- 8.10.3 Single Inheritance in Python 6
- 8.10.4 Multiple Inheritance in Python 7
- 8.10.5 Multilevel Inheritance in Python 8
- 8.10.6 Hierarchical Inheritance in Python 9
- 8.10.7 Hybrid Inheritance in Python 10

8.10. INHERITANCE IN PYTHON

Inheritance in Python is a fundamental concept in object-oriented programming (OOP) that enables the creation of new classes (called child classes or derived classes) based on existing classes (called parent classes or base classes). Inheritance allows child classes to inherit attributes and methods from their parent classes, facilitating code reuse, modularity, and the creation of hierarchical relationships among classes.



The key concepts related to inheritance in Python:

- **Parent Class (Base Class):** The parent class is the class whose attributes and methods are inherited by other classes. It serves as a template or blueprint for creating new classes. Parent classes can have their own attributes and methods.
- **Child Class (Derived Class):** The child class is the new class created based on the parent class. It inherits attributes and methods from the parent class. Child classes can also define their own additional attributes and methods or override existing ones inherited from the parent class.

8.10.1 Implementation of Inheritance in Python

In Python, inheritance is implemented using the syntax **class ChildClassName(ParentClassName):**. The child class is defined by specifying the name of the parent class in parentheses after the child class name.

The syntax of simple inheritance in Python involves creating a child class that inherits attributes and methods from a parent class. The basic syntax:

```
class ParentClass:
  # Parent class attributes and methods
  pass

class ChildClass(ParentClass):
  # Child class attributes and methods
  Pass
```

Explanation of the syntax:

1. **class ParentClass: -** This line defines the parent class. We can define attributes and methods inside this class.
2. **class ChildClass(ParentClass): -** This line defines the child class, ChildClass, which inherits from ParentClass. By specifying ParentClass in parentheses after ChildClass, it is indicated that ChildClass is inheriting from ParentClass.
3. Inside both classes, we can define attributes and methods as needed. The child class inherits all attributes and methods from the parent class, but it can also have its own additional attributes and methods.

Here's a simple example demonstrating this syntax:

```
# Parent class
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        pass

# Child class inheriting from Animal
class Dog(Animal):
    def __init__(self, name):
        super().__init__('Dog')
        self.name = name

    def make_sound(self):
        return "Woof!"
```

In this example, **Animal** is the parent class with attributes and methods, **Dog** is the child class that inherits from **Animal** and has its own attributes and methods. The **super()** function is used in the **Dog** class's constructor to call the constructor of the parent class. **Dog** class overrides the **make_sound()** method inherited from Animal with its own implementation.

Accessing Parent Class Methods

In Python, child classes can access methods and attributes of the parent class using the **super()** function. The **super()** function provides a way to call methods defined in the parent class from within the child class. Hence, we can access parent class methods from within a child class using the **super()** function.

Here's how we can access parent class methods in Python:

```
class ParentClass:
    def parent_method(self):
        return "This is a method from the parent class."

class ChildClass(ParentClass):
    def child_method(self):
        # Call parent class method using super()
        parent_result = super().parent_method()
        return f"Child class calling parent method: {parent_result}"

# Create an instance of the child class
child_obj = ChildClass()

# Call the child class method
print(child_obj.child_method())

# Output: Child class calling parent method: This is a method from the parent class.
```

Method Overriding

Method overriding in inheritance allows a child class to provide a specific implementation for a method that is already defined in its parent class. When a method is overridden in a child class, the version of the method defined in the child class is executed when the method is called on an instance of the child class. This allows child classes to customize the behavior of inherited methods according to their own requirements.

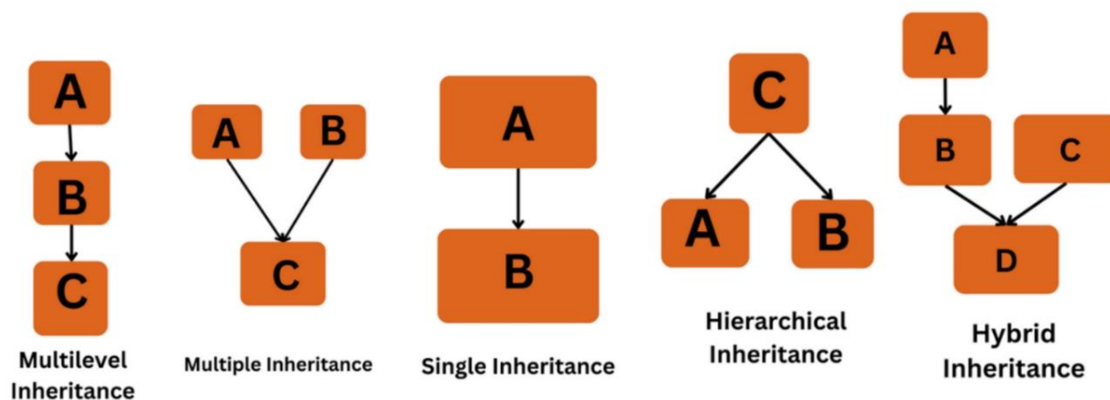
Here's how method overriding works in Python:

```
class ParentClass:
    def method(self):
        return "Parent class method"

class ChildClass(ParentClass):
    def method(self):
        return "Child class method"

# Create instances of both parent and child classes
parent_obj = ParentClass()
child_obj = ChildClass()

# Call method on instances
print(parent_obj.method())  # Output: Parent class method
print(child_obj.method())   # Output: Child class method
```

In this example, **ParentClass** defines a method called **method**(). **ChildClass** inherits from **ParentClass** and provides its own implementation of the **method**() method. When **method()** is called on an instance of **ChildClass**, the version of the method defined in **ChildClass** is executed, overriding the behavior of the method defined in **ParentClass**.

8.10.2 Types of Inheritances in Python

In Python, inheritance can take several forms, each with its own implications and use cases. Here are the common types of inheritance:



1. Single Inheritance: Single inheritance involves one class inheriting from another class. In this type of inheritance, a derived class (child class) extends a base class (parent class) by inheriting its attributes and methods. Python supports single inheritance by default.

Example:

```
class Parent:
  pass

class Child(Parent):
  pass
```

2. Multiple Inheritance: Multiple inheritance allows a class to inherit attributes and methods from more than one parent class. It enables a derived class to inherit from multiple base classes. While powerful, multiple inheritance can lead to complex class hierarchies and potential conflicts called diamond problem.

Example:

```
class Parent1:
  pass

class Parent2:
  pass

class Child(Parent1, Parent2):
  pass
```

3. Multilevel Inheritance: Multilevel inheritance involves a chain of inheritance where a derived class inherits from a base class, and another class inherits from this derived class, forming a hierarchical structure.

Example:

```
class Grandparent:
  pass

class Parent(Grandparent):
  pass

class Child(Parent):
  pass
```

4. Hierarchical Inheritance: Hierarchical inheritance occurs when more than one class inherits from a single parent class. Multiple child classes inherit from the same parent class.

Example:

```
class Parent:
  pass

class Child1(Parent):
```

```
    pass

class Child2(Parent):
    pass
```

5. Hybrid Inheritance: Hybrid inheritance is a combination of different types of inheritance. It combines single, multiple, or multilevel inheritance in a single class hierarchy.

Example:

```
class Base1:
    pass

class Base2:
    pass

class Derived(Base1, Base2):
    pass
```

8.10.3 Single Inheritance in Python

Single inheritance in Python occurs when a class inherits from only one parent class. In this scenario, the child class inherits all the attributes and methods of the parent class. Single inheritance is the most common type of inheritance and is widely used in object-oriented programming.

An example of single inheritance:

```
# Parent class
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def drive(self):
        return f"{self.brand} {self.model} is driving."

# Child class inheriting from Vehicle
class Car(Vehicle):
    def __init__(self, brand, model, color):
        # Call the constructor of the parent class using super()
        super().__init__(brand, model)
        self.color = color

    def honk(self):
        return f"{self.brand} {self.model} in {self.color} color honks."

# Creating instances of child classes
car1 = Car("Toyota", "Camry", "Red")
car2 = Car("Honda", "Civic", "Blue")
```

```
# Accessing attributes and methods
print(car1.brand)      # Output: Toyota
print(car1.model)      # Output: Camry
print(car1.color)      # Output: Red
print(car1.drive())    # Output: Toyota Camry is driving.
print(car1.honk())     # Output: Toyota Camry in Red color honks.

print(car2.brand)      # Output: Honda
print(car2.model)      # Output: Civic
print(car2.color)      # Output: Blue
print(car2.drive())    # Output: Honda Civic is driving.
print(car2.honk())     # Output: Honda Civic in Blue color honks.
```

8.10.4 Multiple Inheritance in Python

Multiple inheritance in Python allows a class to inherit attributes and methods from more than one parent class. This means that a child class can have multiple direct parent classes, each contributing its own set of attributes and methods to the child class.

Let's explore multiple inheritance with an example:

```
# Parent class 1
class A:
    def method_a(self):
        return "Method A from class A"

# Parent class 2
class B:
    def method_b(self):
        return "Method B from class B"

# Child class inheriting from both class A and class B
class C(A, B):
    def method_c(self):
        return "Method C from class C"

# Create an instance of the child class
obj_c = C()

# Access methods from class A
print(obj_c.method_a())  # Output: Method A from class A

# Access methods from class B
print(obj_c.method_b())  # Output: Method B from class B

# Access method from class C
print(obj_c.method_c())  # Output: Method C from class C
```

Example: Multiple inheritance in Python

```
# Parent class 1
class Swimmer:
    def swim(self):
        return "Swimming..."

# Parent class 2
class Walker:
    def walk(self):
        return "Walking..."

# Child class inheriting from Swimmer and Walker
class Amphibian(Swimmer, Walker):
    def __init__(self, name):
        self.name = name

# Creating an instance of the child class
frog = Amphibian("Frog")

# Accessing attributes and methods
print(frog.name)      # Output: Frog
print(frog.swim())    # Output: Swimming...
print(frog.walk())    # Output: Walking...
```

8.10.5 Multilevel Inheritance in Python

Multilevel inheritance in Python involves a chain of inheritance where a derived class (child class) inherits from a base class (parent class), and another class inherits from this derived class, forming a hierarchical structure. In multilevel inheritance, each class serves as both a parent class and a child class.

Example: Multilevel inheritance using a class hierarchy to represent vehicles

```
# Grandparent class
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start(self):
        return f"Starting the {self.brand} {self.model}."

# Parent class inheriting from Vehicle
class Car(Vehicle):
    def drive(self):
        return f"Driving the {self.brand} {self.model}."
```

```
# Child class inheriting from Car
class ElectricCar(Car):
    def charge(self):
        return f"Charging the {self.brand} {self.model}."

# Create an instance of the child class
my_electric_car = ElectricCar("Tesla", "Model S")

# Access methods from Vehicle class (Grandparent)
print(my_electric_car.start())  # Output: Starting the Tesla Model S.

# Access methods from Car class (Parent)
print(my_electric_car.drive())  # Output: Driving the Tesla Model S.

# Access methods from ElectricCar class (Child)
print(my_electric_car.charge()) # Output: Charging the Tesla Model S.
```

8.10.6 Hierarchical Inheritance in Python

Hierarchical inheritance in Python occurs when more than one class inherits from a single parent class. This creates a hierarchical structure where multiple child classes share a common parent class.

Let's illustrate hierarchical inheritance with an example:

```
# Parent class
class Animal:
    def sound(self):
        return "Animal makes a sound"

# Child class inheriting from Animal
class Dog(Animal):
    def bark(self):
        return "Dog barks"

# Child class inheriting from Animal
class Cat(Animal):
    def meow(self):
        return "Cat meows"

# Create instances of child classes
dog = Dog()
cat = Cat()

# Accessing methods
print(dog.sound())  # Output: Animal makes a sound
print(dog.bark())   # Output: Dog barks
```

```
print(cat.sound())  # Output: Animal makes a sound
print(cat.meow())   # Output: Cat meows
```

8.10.7 Hybrid Inheritance in Python

Hybrid inheritance in Python is a combination of different types of inheritance, such as single, multiple, or multilevel inheritance, within a single class hierarchy. This means that a child class can inherit from multiple parent classes, and those parent classes themselves may inherit from other classes, creating a hybrid inheritance structure.

Let's illustrate hybrid inheritance with an example:

```
# Parent class 1
class A:
    def method_a(self):
        return "Method A"

# Parent class 2
class B:
    def method_b(self):
        return "Method B"

# Parent class 3 inheriting from class A
class C(A):
    def method_c(self):
        return "Method C"

# Child class inheriting from classes B and C
class D(B, C):
    def method_d(self):
        return "Method D"

# Create an instance of the child class
obj_d = D()

# Access methods from different parent classes
print(obj_d.method_a())  # Output: Method A
print(obj_d.method_b())  # Output: Method B
print(obj_d.method_c())  # Output: Method C
print(obj_d.method_d())  # Output: Method D
```

Example: Let's consider a hypothetical example where a software system is being developed to manage employees in an organization. In this scenario, hybrid inheritance can be applied to model different types of employees with varying roles and responsibilities.

```
# Parent class 1
class Employee:
    def __init__(self, name, employee_id):
        self.name = name
        self.employee_id = employee_id

    def display_info(self):
        return f"Name: {self.name}, ID: {self.employee_id}"

# Parent class 2
class Manager(Employee):
    def manage_team(self):
        return "Managing team tasks"

# Parent class 3 inheriting from Manager
class Developer(Manager):
    def write_code(self):
        return "Writing code"

# Child class inheriting from Employee and Manager
class SalesExecutive(Employee, Manager):
    def handle_clients(self):
        return "Handling client meetings"

# Child class inheriting from Employee and Developer
class QAEngineer(Employee, Developer):
    def test_code(self):
        return "Testing code"

# Create instances of child classes
sales_executive = SalesExecutive("John Doe", "SE001")
qa_engineer = QAEngineer("Alice Smith", "QA001")

# Accessing methods
print(sales_executive.display_info())    # Output: Name: John Doe, ID: SE001
print(sales_executive.manage_team())     # Output: Managing team tasks
print(sales_executive.handle_clients())  # Output: Handling client meetings

print(qa_engineer.display_info())        # Output: Name: Alice Smith, ID: QA001
print(qa_engineer.write_code())          # Output: Writing code
print(qa_engineer.test_code())           # Output: Testing code
```

In this example:

- **Employee** class represents a generic employee with basic information like name and ID.
- **Manager** class inherits from **Employee** and adds functionalities specific to managers, such as managing a team.
- **Developer** class inherits from **Manager** and adds functionalities specific to developers, such as writing code.
- **SalesExecutive** class inherits from both **Employee** and **Manager**, demonstrating multiple inheritance. It combines the functionalities of both employee and manager roles.
- **QAEngineer** class inherits from both **Employee** and **Developer**, demonstrating multiple inheritance. It combines the functionalities of both employee and developer roles.

⌘⌘⌘