Inheritance

> Inheritance is a mechanism that allows a class to inherit properties and behaviors from another class.

- PRO: Promotes code reuse and establishes relationships between classes.
- CON: Increases Coupling

Inheritance is based on a hierarchical relationship between classes, where a derived class (also known as a subclass or child class) inherits the characteristics of a base class (also known as a superclass or parent class). The derived class extends the functionality of the base class by adding new features or overriding existing ones.

The key idea behind inheritance is that the derived class inherits all the attributes (data members) and behaviors (methods) of the base class, and it can also introduce its own specific attributes and behaviors. This allows for creating a hierarchy of classes with increasing specialization.

Template

```
class parent_class:
body of parent class


class child_class( parent_class):
body of child class
```

Python Code:

```
class Car:          #parent class


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


    def description(self):
        return f"The {self.name} car gives the mileage of {self.mileage}km/l"


class BMW(Car):     #child class
    pass
```

```
class Audi(Car):     #child class
    def audi_desc(self):
        return "This is the description method of class Audi."
obj1 = BMW("BMW 7-series",39.53)
print(obj1.description())

obj2 = Audi("Audi A8 L",14)
print(obj2.description())
print(obj2.audi_desc())
```

We can check the base or parent class of any class using a built-in class attribute **__bases__**

```
print(BMW.__bases__, Audi.__bases__)
```

Inheritance in Object Oriented Programming - Print Class

As we can see here, the base class of both sub-classes is **Car.** Now, let's see what happens when using **__base__** with the parent class Car:

```
print( Car.__bases__ )Output:
```

Inheritance in Object Oriented Programming - Print sub-class

Whenever we create a new class in Python 3.x, it is inherited from a built-in basic class called **Object.** In other words, the Object class is the root of all classes.

Forms of Inheritance

There are broadly five forms of inheritance in oops based on the involvement of parent and child classes.

Single Inheritance

This is a form of inheritance in which a class inherits only one parent class. This is the simple form of inheritance and hence, is also referred to as **simple inheritance**.

```
class Parent:
  def f1(self):
    print("Function of parent class.")
```

```
class Child(Parent):
  def f2(self):
    print("Function of child class.")

object1 = Child()
object1.f1()
object1.f2()
```

Output:

Image not found or type unknown

Inheritance in Object Oriented Programming - Single Inheritance

Here oject1 is an instantiated object of class Child, which inherits the parent class ‘Parent’.

Multiple Inheritance

> Avoid using multiple inheritance in Python. If you need to reuse code from multiple classes, you can use composition.

Multiple inheritances is when a class inherits more than one parent class. The child class, after inheriting properties from various parent classes, has access to all of its objects.

One of the main problems with multiple inheritance is the diamond problem. This occurs when a class inherits from two classes that both inherit from a third class. In this case, it is not clear which implementation of the method from the third class should be used.

When a class inherits from multiple classes, it can be difficult to track which methods and attributes are inherited from which class.

```
class Parent_1:
  def f1(self):
    print("Function of parent_1 class.")

class Parent_2:
  def f2(self):
    print("Function of parent_2 class.")

class Parent_3:
  def f3(self):
    print("function of parent_3 class.")
```

```
class Child(Parent_1, Parent_2, Parent_3):
  def f4(self):
    print("Function of child class.")

object_1 = Child()
object_1.f1()
object_1.f2()
object_1.f3()
object_1.f4()
```

Output:

Inheritance in Object Oriented Programming - Multiple Inheritance

Here we have one Child class that inherits the properties of three-parent classes Parent_1, Parent_2, and Parent_3. All the classes have different functions, and all of the functions are called using the object of the Child class.

But suppose a child class inherits two classes having the same function:

```
class Parent_1:
  def f1(self):
    print("Function of parent_1 class.")

class Parent_2:
  def f1(self):
    print("Function of parent_2 class.")

class Child(Parent_1, Parent_2):
  def f2(self):
    print("Function of child class.")
```

Here, the classes Parent_1 and Parent_2 have the same class methods, f1(). Now, when we create a new object of the child class and call f1() from it since the child class is inheriting both parent classes, what do you think should happen?

```
obj = Child()
obj.f1()
```

Output:

Image not found or type unknown

Inheritance in Object Oriented Programming

So in the above example, why was the function f1() of the class Parent_2 not inherited?

In multiple inheritances, the child class first searches for the method in its own class. If not found, then it searches in the parent classes depth_first and left-right order. Since this was an easy example with just two parent classes, we can clearly see that class Parent_1 was inherited first, so the child class will search the method in Parent_1 class before searching in class Parent_2.

But for complicated inheritance oops problems, it gets tough to identify the order. So the actual way of doing this is called **Method Resolution Order (MRO)** in Python. We can find the MRO of any class using the attribute __mro__.

```
Child.__mro__
```

Output:

Image not found or type unknown

Inheritance in Object Oriented Programming - MRO

This tells that the Child class first visited the class Parent_1 and then Parent_2, so the f1() method of Parent_1 will be called.

Let’s take a bit complicated example in Python:

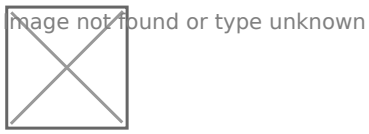
```
class Parent_1:
pass

class Parent_2:
pass

class Parent_3:
pass

class Child_1(Parent_1,Parent_2):
pass

class Child_2(Parent_2,Parent_3):
pass

class Child_3(Child_1,Child_2,Parent_3):
pass
```

Here, the class Child_1 inherits two classes – Parent_1 and Parent_2. The class Child_2 is also inheriting two classes – Parent_2 and Parent_3. Another class, Child_3, is inheriting three classes – Child_1, Child_2, and Parent_3.

Now, just by looking at this inheritance, it is quite hard to determine the Method Resolution Order for class Child_3. So here is the actual use of __mro__.

```
Child_3.__mro__
```

Output:



We can see that, first, the interpreter searches Child_3, then Child_1, followed by Parent_1, Child_2, Parent_2, and Parent_3, respectively.

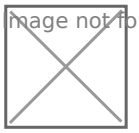
Multi-level Inheritance

For example, a class_1 is inherited by a class_2, and this class_2 also gets inherited by class_3, and this process goes on. This is known as multi-level inheritance oops. Let’s understand with an example:

```
class Parent:
  def f1(self):
    print("Function of parent class.")

class Child_1(Parent):
  def f2(self):
    print("Function of child_1 class.")

class Child_2(Child_1):
  def f3(self):
    print("Function of child_2 class.")

obj_1 = Child_1()
obj_2 = Child_2()

obj_1.f1()
obj_1.f2()

print("\n")
```

```
obj_2.f1()
obj_2.f2()
obj_2.f3()
```

Output:



Here, the class Child_1 inherits the Parent class, and the class Child_2 inherits the class Child_1. In this Child_1 has access to functions f1() and f2() whereas Child_2 has access to functions f1(), f2() and f3(). If we try to access the function f3() using the object of class Class_1, then an error will occur stating:

‘Child_1’ object has no attribute ‘f3’.

```
obj_1.f3()
```

Hierarchical Inheritance

In this, various Child classes inherit a single Parent class. The example given in the introduction of the inheritance is an example of Hierarchical inheritance since classes BMW and Audi inherit class Car.

For simplicity, let’s look at another example:

```
class Parent:
deff1(self):
print("Function of parent class.")

class Child_1(Parent):
deff2(self):
print("Function of child_1 class.")

class Child_2(Parent):
deff3(self):
```

```
print("Function of child_2 class.")

obj_1 = Child_1()
obj_2 = Child_2()

obj_1.f1()
obj_1.f2()

print('\n')
obj_2.f1()
obj_2.f3()
```

Output:

Here two child classes inherit the same parent class. The class Child_1 has access to functions f1() of the parent class and function f2() of itself. Whereas the class Child_2 has access to functions f1() of the parent class and function f3() of itself.

Hybrid Inheritance

When there is a combination of more than one form of inheritance, it is known as hybrid inheritance. It will be more clear after this example:

```
class Parent:
  def f1(self):
    print("Function of parent class.")

class Child_1(Parent):
  def f2(self):
    print("Function of child_1 class.")

class Child_2(Parent):
  def f3(self):
    print("Function of child_2 class.")

class Child_3(Child_1, Child_2):
  def f4(self):
```

```
    print("Function of child_3 class.")


obj = Child_3()
obj.f1()
obj.f2()
obj.f3()
obj.f4()
```

Output:

In this example, two classes, ‘Child_1′ and ‘Child_2’, are derived from the base class ‘Parent’ using hierarchical inheritance. Another class, ‘Child_3’, is derived from classes ‘Child_1’ and ‘Child_2’ using multiple inheritances. The class ‘Child_3’ is now derived using hybrid inheritance.

Method Overriding in Inheritance in Python

> We do this so we can have our own methods without modifying the base class. If we modify the base class, then that could lead to other problems if other users are expecting certain functionality from the said base class

The concept of overriding is very important in inheritance oops. It gives the special ability to the child/subclasses to provide specific implementation to a method that is already present in their parent classes.

```
class Parent:
  def f1(self):
    print("Function of Parent class.")


class Child(Parent):
  def f1(self):
    print("Function of Child class.")
```

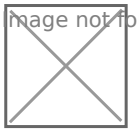
```
obj = Child()
obj.f1()
```

Output:

Here the function f1() of the child class has overridden the function f1() of the parent class. Whenever the object of the child class invokes f1(), the function of the child class gets executed. However, the object of the parent class can invoke the function f1() of the parent class.

```
obj_2 = Parent()
obj_2.f1()
```

Output:



Super() Function in Python

The super() function in Python returns a proxy object that references the parent class using the **super** keyword. This super() keyword is basically useful in accessing the overridden methods of the parent class.

The official documentation of the super() function sites two main uses of super():

In a class hierarchy with single inheritance oops, *super* helps to refer to the parent classes without naming them explicitly, thus making the code more maintainable.

For example:

```
class Parent:
  def f1(self):
    print("Function of Parent class.")
```

```
class Child(Parent):
  def f1(self):
    super().f1()
    print("Function of Child class.")

obj = Child()
obj.f1()
```

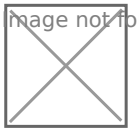
Output:

Here, with the help of super().f1(), the f1() method of the superclass of the child class, i.e., the parent class, has been called without explicitly naming it.

One thing to note here is that the super() class can accept two parameters- the first is the name of the subclass, and the second is an object that is an instance of that subclass. Let’s see how:

```
class Parent:
  def f1(self):
    print("Function of Parent class.")

class Child(Parent):
  def f1(self):
    super( Child, self ).f1()
    print("Function of Child class.")

obj = Child()
obj.f1()
```

Output:



The first parameter refers to the subclass **Child**, while the second parameter refers to the object of Child, which, in this case, is **self**. You can see the output after using super(), and super(Child, self) is the same because, in Python 3, super(Child, self) is equivalent to self().

Now let’s see one more example using the __init__ function.

```
class Parent(object):
  def__init__(self, ParentName):
    print(ParentName, 'is derived from another class.')

class Child(Parent):
  def__init__(self, ChildName):
    print(name,'is a sub-class.')
    super().__init__(ChildName)

obj = Child('Child')
```

Output:

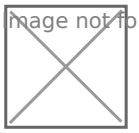
What we have done here is that we called the __init__ function of the parent class (inside the child class) using *super().__init__(ChildName)*. And as the __init__ method of the parent class requires one argument, it has been passed as “ChildName”. So after creating the object of the child class, first, the __init__ function of the child class got executed, and after that, the __init__ function of the parent class.

The second use case is to support multiple cooperative inheritances in a dynamic execution environment.

```
class First():
  def __init__(self):
    print("first")
    super().__init__()

class Second():
  def __init__(self):
    print("second")
    super().__init__()

class Third(Second, First):
  def __init__(self):
    print("third")
```

```
    super().__init__()


obj = Third()
```

Output:



The super() call finds the next method in the MRO at each step, which is why First and Second have to have it, too; otherwise, execution stops at the end of *first().__init__*.

Note that the super-class of both First and Second is *Object.*

Let’s find the MRO of Third() as well.

```
Third.__mro__
```

Output:

The order is Third > Second > First, and the same is the order of our output.

Revision #5
Created 26 February 2024 01:48:00 by victor
Updated 1 April 2024 03:39:40 by victor