Theory: Method overriding

26 minutes 0 / 5 problems solved

One important concept of object-oriented programming is **overriding**. Overriding is the ability of a class to change the implementation of the methods inherited from its ancestor classes.

This feature is extremely useful as it allows us to explore inheritance to its full potential. We can not only reuse existing code and method implementations but also upgrade and advance them if needed.

Overriding is a concept applicable only to class hierarchies: without inheritance, we cannot talk about method overriding. Let's consider an example of a class hierarchy:

```
class Parent:
    def do_something(self):
        print("Did something")


class Child(Parent):
    def do_something(self):
        print("Did something else")



parent = Parent()

child = Child()


parent.do_something()  # Did something

child.do_something()  # Did something else
```

Here, the method `do_something` is overridden in the class `Child`. If we hadn't overridden it, the method would have the same implementation as in the class `Parent`. The code `child.do_something()` would then print `Did something`.

§1. super()

Python has a special function for calling the method of the parent class inside the methods of the child class: the `super()` function. It returns a proxy, a temporary object of the parent class, and allows us to call a method of the parent class using this proxy. Let's take a look at the following example:

```
class Parent:
    def __init__(self, name):
        self.name = name
        print("Called Parent __init__")


class Child(Parent):
    def __init__(self, name):
        super().__init__(name)

        print("Called Child __init__")
```

We've overridden the `__init__()` method in the child class but inside it we've called the `__init__()` of the parent class. If we create an object of the class `Child`, we will get the following output:

```
jack = Child("Jack")
# Called Parent __init__
# Called Child __init__
```

Current topic:

Method overriding

Topic depends on:

- ✕ Polymorphism
- ✕ Inheritance

Topic is required for:

- Multiple inheritance
- Abstract classes
- User-defined exceptions
- Rendering templates

Table of contents:

In Python 3 the method `super()` doesn't have any required parameters. In earlier versions, however, you had to specify the class from which the method would search for a superclass. In our example, instead of `super().__init__(name)` we would write `super(Child, self).__init__(name)`. Both lines of code mean the same thing: that we want to find the superclass of the class `Child` and then call its `__init__` method. In Python 3 these are equivalent, so you don't have to explicitly write the type. However, it may be useful if you want to access the method of the "grandparent" class: the parent class of the parent class.

§2. super() with single inheritance

The method `super()` is mostly used in cases of multiple inheritance: when a class inherits from two or more classes. There it is most convenient and useful but you'll have a chance to learn about that in the next topics. This method can also be of use with single inheritance which is what we'll cover now.

Suppose we have the following classes:

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


class Cat(Animal):
    def __init__(self, name):
        self.name = name

```

In the subclass `Cat`, we've overridden the `__init__()` method. Now the objects of the class `Cat` do not have the `species` attribute. We would like for objects of the `Cat` class to have this attribute, but adding it as a parameter of the `__init__` seems a bit excessive. We could, of course, simply create this attribute inside the initializer, but there is a more elegant (and more Pythonic) solution. This solution, as expected, is the `super()` method:

```
class Animal:
    def __init__(self, species):
        self.species = species
        print("Animal __init__")


class Cat(Animal):
    def __init__(self, name):
        super().__init__("cat")

        self.name = name

        print("Cat __init__")
```

Let's create a cat and see how this has worked:

```
fluffy = Cat("Fluffy")
# Animal __init__
# Cat __init__

print(fluffy.species, fluffy.name)  # cat Fluffy
```

Both `__init__()` methods have done their job and our cat has both the `species` and the `name` attributes.

You may wonder why we had to do it this way. Why did we have to call the parent implementation of the method when we could manage without it? Well, the example above is a very simple one. In real-life projects, classes, their methods and the relationships between them are much more sophisticated.

Overriding does provide us with an opportunity to enhance the methods of the parent class but it doesn't mean that we should discard the original implementations. Sometimes, you may not have full access to the original implementation and you may not know everything that happens there. If you

just override it, there may be unexpected consequences. So, it is recommended to always call the parent implementation. This way, you get the best of both worlds: you have the original implementation and your enhancements.

Just be careful and thoughtful when overriding method and using the `super()` function and you'll do great!

Report a typo

94 users liked this theory. **2** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion