Theory: Multiple inheritance

29 minutes 0 / 0 problems solved

Skip this topic Start practicing

566 users solved this topic. Latest completion was 1 day ago.

Current topic:

Multiple inheritance

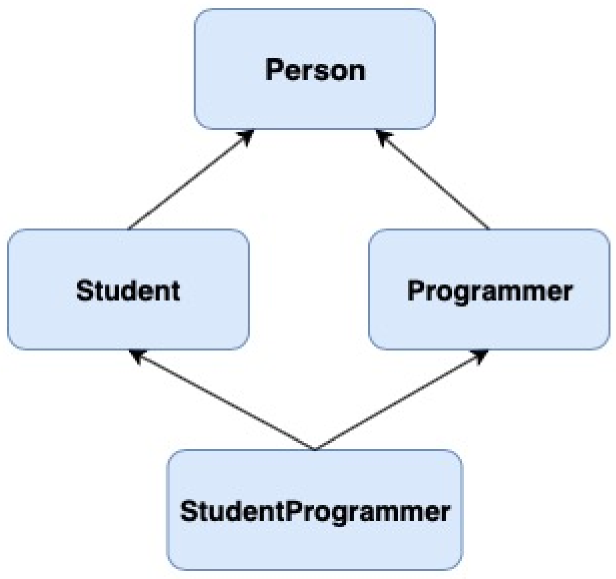
By now, you are familiar with the mechanism of inheritance. Now it's time to go deeper and gain insight into multiple inheritance.

Multiple inheritance is when a class has two or more parent classes.

In the code, multiple inheritance looks similar to the single inheritance. Only now, in brackets after the child class, you need to write all parent classes instead of just one:

```
class ParentClass1:
    ...


class ParentClass2:
    ...


class ParentClass3:

    ...



# Class definition with multiple inheritance

class ChildClass(ParentClass1, ParentClass2, ParentClass3):

    ...
```

Let's have a look at a particular class hierarchy. In the scheme, arrows point from the child class to the parent class.



Class hierarchy with multiple inheritance.

As you can see, there are a basic parent class `Person` and classes `Student` and `Programmer` that inherit from it. The class `StudentProgrammer`, in its turn, inherits from both `Student` and `Programmer` classes which makes it a case of multiple inheritance. This way, we can say that `StudentProgrammer` has two parent classes, `Student` and `Programmer` while `Person` can be regarded as a "grandparent" class.

Here's how the basic code for this hierarchy looks:

Table of contents:

```
class Person:
    ...


class Student(Person):
    ...


class Programmer(Person):

    ...




class StudentProgrammer(Student, Programmer):

    ...
```

Feedback & Comments

§1. The diamond problem

As you remember, classes inherit methods and attributes from their parents. If inheritance is simple, everything is clear and straightforward. However, when we deal with multiple inheritance, things are bound to get a little bit more complicated.

One of the most famous "complications" is called a **diamond problem** (or, rather dramatically, "Deadly Diamond of Death"). The diamond problem is an ambiguity that arises in the case of multiple inheritance. The class hierarchy we've described above is a perfect example of the structure that may cause this problem.

So, we have a class hierarchy with one superclass, two classes that inherit from it, and a class that has those child classes as parents. As you can see from the hierarchy scheme above, the whole structure is shaped like a diamond which is where the name of the issue comes from (not the Rihanna song, unfortunately).

Let's add some methods to classes to see where the problems lie.

```
class Person:
    def print_message(self):
        print("Message from Person")


class Student(Person):
    def print_message(self):
        print("Message from Student")



class Programmer(Person):

    def print_message(self):

        print("Message from Programmer")




class StudentProgrammer(Student, Programmer):

    ...
```

Class `Person` has a method `print_message` that classes `Student` and `Programmer` override to print their own messages. The class `StudentProgrammer` doesn't override this method.

The question is, then: if we create an instance of the class `StudentProgrammer` and call `print_message` method, which message will be printed?

This is the crux of the diamond problem: how to choose an implementation when we have several alternatives.

§2. MRO

Different programming languages use different techniques for dealing with the diamond problem. Basically, what we need to do is to somehow transform the diamond shape (or any complicated hierarchy) into a single line so that we know in which order to look for the necessary method. Python uses the C3 Linearization algorithm that calculates the **Method Resolution Order (MRO).**

MRO tells us how the particular class hierarchy looks in a linear form and how we should navigate this hierarchy. Two basic rules are that child classes precede parent classes and parent classes are placed in the order they were listed in.

Each class has a `__mro__` attribute (inherited from `object`) that contains the parent classes in the MRO. Let's print this attribute of the `StudentProgrammer` class and see what we'll get:

```
print(StudentProgrammer.__mro__)

# (<class '__main__.StudentProgrammer'>, <class '__main__.Student'>, <class '__mai
n__.Programmer'>, <class '__main__.Person'>, <class 'object'>)
```

You can see that according to MRO, the immediate parent of the class `StudentProgrammer` is `Student`. It means that if we call `print_method`, the version from the class `Student` will be implemented.

```
jack = StudentProgrammer()
jack.print_message()  # Message from Student
```

Note that the MRO looks like this because, in the definition of the class `StudentProgrammer`, the class `Student` precedes `Programmer`. If the situation was reversed, the output of the code snippet above would be `Message from Programmer`.

§3. super() with multiple inheritance

By now, you already know how the `super()` function is used in single inheritance. However, it truly shines when we have to deal with multiple inheritance, especially the diamond problem. The `super()` function uses MRO to call the method and get an attribute of the immediate parent class. You don't need to analyze the hierarchy and figure out the parent class yourself, the `super()` function will do it for you.

Let's modify our classes by adding the `super()` calls to the `print_message` methods.

```
class Person:
    def print_message(self):
        print("Message from Person")


class Student(Person):
    def print_message(self):
        print("Message from Student")
        super().print_message()


class Programmer(Person):
    def print_message(self):
        print("Message from Programmer")
        super().print_message()


class StudentProgrammer(Student, Programmer):
    def print_message(self):
        super().print_message()
```

Each class (except `Person`) now calls the method of the parent class after printing its own message. Now if we call this method for `StudentProgrammer` class we'll see the following:

```
jack = StudentProgrammer()
jack.print_message()
# Message from Student
# Message from Programmer
# Message from Person
```

The messages were printed in the MRO of the class `StudentProgrammer` without any repetitions. This is the beauty and the benefit of the `super()` function: if you've designed your classes well, you don't need to worry about the order.

§4. Summary

In this topic, we've looked at multiple inheritance in Python: a situation when a class has more than one parent. While it can be very useful, multiple inheritance can also lead to some problems, for example, the diamond problem.

Python uses method resolution order, MRO, to deal with ambiguity. Every class has an attribute that contains its MRO. The `super()` function, which is used for accessing methods and attributes of the parent class, makes use of the MRO to determine which implementation to call.

We encourage you to experiment with different class hierarchies and the `super()` function. This will allow you to get the hang of multiple inheritance, deal with hidden dangers and learn how to construct complex hierarchies in an efficient way.

Report a typo

55 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (3) Hints (0) Useful links (0) Show discussion