Python Multiple Inheritance

Summary: in this tutorial, you'll learn about Python multiple inheritance and how method order resolution works in Python.

Introduction to the Python Multiple inheritance.

When a class (https://www.pythontutorial.net/python-oop/python-class/) inherits from a single class, you have single inheritance (https://www.pythontutorial.net/python-oop/python-inheritance/) . Python allows a class to inherit from multiple classes. If a class inherits from two or more classes, you'll have multiple inheritance.

To extend multiple classes, you specify the parent classes inside the parentheses `()` after the class name of the child class like this:

```
class ChildClass(ParentClass1, ParentClass2, ParentClass3):
    pass
```

The syntax for multiple inheritance is similar to a parameter list in the class definition. Instead of including one parent class inside the parentheses, you include two or more classes, separated by a comma.

Let’s take an example to understand how multiple inheritance works:

Python Multiple Inhertiance

First, define a class `Car` that has the `go()` method:

```
class Car:
    def go(self):
        print('Going')
```

Second, define a class `Flyable` that has the `fly()` method:

```
class Flyable:
    def fly(self):
        print('Flying')
```

Third, define the `FlyingCar` that inherits from both `Car` and `Flyable` classes:

```
class FlyingCar(Flyable, Car):
    pass
```

Since the `FlyingCar` inherits from `Car` and `Flyable` classes, it reuses the methods from those classes. It means you can call the `go()` and `fly()` methods on an instance of the `FlyingCar` class like this:

```
if __name__ == '__main__':
    fc = FlyingCar()
    fc.go()
    fc.fly()
```

Output:

```
Going
Flying
```

Method resolution order (MRO)

When the parent classes have methods with the same name and the child class calls the method, Python uses the method resolution order (MRO) to search for the right method to call. Consider the following example:

First, add the `start()` method to the `Car` , `Flyable` , and `FlyingCar` classes. In the `start()` method of the `FlyingCar` class, call the `start()` method of the `super()` :

```python
class Car:
    def start(self):
        print('Start the Car')

    def go(self):
        print('Going')


class Flyable:
    def start(self):
        print('Start the Flyable object')
    def fly(self):
        print('Flying')


class FlyingCar(Flyable, Car):
    def start(self):
        super().start()
```

Second, create an instance of the `FlyingCar` class and call the `start()` method:

```
if __name__ == '__main__':
    car = FlyingCar()
    car.start()
```

Output:

```
Start the Flyable object
```

As you can see clearly from the output, the `super().start()` calls the `start()` method of the `Flyable` class.

The following shows the `__mro__` of the `FlyingCar` class:

```
print(FlyingCar.__mro__)
```

Output:

```
(<class '__main__.FlyingCar'>, <class '__main__.Flyable'>, <class '__main__.Car'
```

From left to right, you'll see the `FlyingCar` , `Flyable` , `Car` , and `object` .

Note that the `Car` and `Flyable` objects inherit from the `object` class implicitly. When you call the `start()` method from the `FlyingCar` 's object, Python uses the `__mro__` class search path.

Since the `Flyable` class is next to the `FlyingCar` class, the `super().start()` calls the `start()` method of the `FlyingCar` class.

If you flip the order of `Flyable` and `Car` classes in the list, the `__mro__` will change accordingly. For example:

```
# Car, Flyable classes...


class FlyingCar(Car, Flyable):
    def start(self):
        super().start()


if __name__ == '__main__':
    car = FlyingCar()
    car.start()

    print(FlyingCar.__mro__)
```

Output:

```
Start the Car
(<class '__main__.FlyingCar'>, <class '__main__.Car'>, <class '__main__.Flyable'>
```

In this example, the `super().start()` calls the `start()` method of the `Car` class instead, based on their orders in the method order resolution.

Multiple inheritance & super

First, add the `__init__` method to the `Car` class:

```
class Car:
    def __init__(self, door, wheel):
        self.door = door
        self.wheel = wheel

    def start(self):
        print('Start the Car')

    def go(self):
        print('Going')
```

Second, add the `__init__` method to the `Flyable` class:

```
class Flyable:
    def __init__(self, wing):
        self.wing = wing

    def start(self):
        print('Start the Flyable object')

    def fly(self):
        print('Flying')
```

The `__init__` of the `Car` and `Flyable` classes accept a different number of parameters. If the `FlyingCar` class inherits from the `Car` and `Flyable` classes, its `__init__` method needs to call the right `__init__` method specified in the method order resolution `__mro__` of the `FlyingCar` class.

Third, add the `__init__` method to the `FlyingCar` class:

```
class FlyingCar(Flyable, Car):
    def __init__(self, door, wheel, wing):
        super().__init__(wing=wing)
        self.door = door
        self.wheel = wheel

    def start(self):
        super().start()
```

The method order resolution of the `FlyingCar` class is:

```
(<class '__main__.FlyingCar'>, <class '__main__.Flyable'>, <class '__main__.Car'
```

the `super().__init__()` calls the `__init__` of the `FlyingCar` class. Therefore, you need to pass the `wing` argument to the `__init__` method.

Because the `FlyingCar` class cannot access the `__init__` method of the `Car` class, you need to initialize the `door` and `wheel` attributes individually.

Summary

- Python multiple inheritance allows one class to inherit from multiple classes.
- The method order resolution defines the class search path to find the method to call.