Super Inherit Your Python Class

E.Y. · Follow
6 min read · Sep 17, 2020

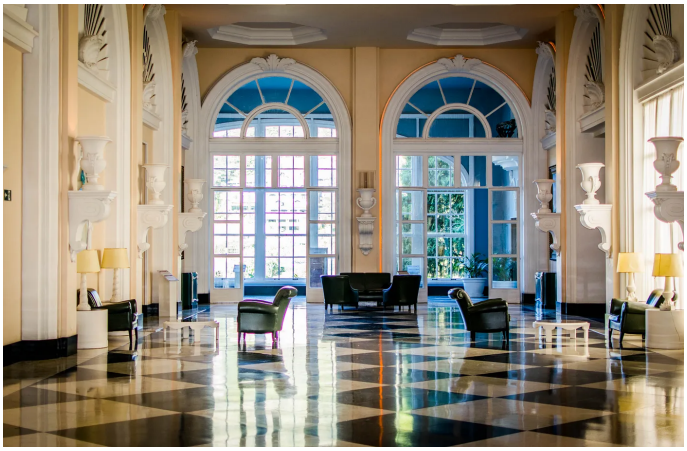

Photo by Carlos de Almeida on Unsplash

Not long ago, I started to use Python in my projects at the spare time. Luckily, as it shares quite a lot of similarities with Ruby, it's not difficult to get onboard. But there're some interesting topics in Python that worth digging deeper, such as what we discuss today, **inheritance using super keyword**.

Normally we write a class with inheritance like below.

```
class Demo():
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

In this way, we can easily create class inheritance if needed. But how so? And what does the keyword `super` do?

Let's start with a simple example:

```
class Quadrilateral:
    def __init__(self, length, height):
        self.length = length
        self.height = height

    def area(self):
        return self.length * self.height

class Square(Quadrilateral):
    def __init__(self, length):
        super().__init__(length, length)
```

Here inside `Square` class `init` method, we can use `super()` to call the `__init__()` of the `Quadrilateral` class, which pass the params to the `Quadrilateral` class `init` method. So is the `area` class get inherited as well through the `super` keyword.

You can also call super without the `init` step like the example below, but it's an anti-pattern.

```
class Cuboid(Quadrilateral):
    def volume(self):
        face_area = super().area()
        return face_area * self.length

>>> cuboid = Cuboid(2,3)
>>> cuboid.volume()
12
```

The reason we can do this is because `Cuboid` inherits from `Quadrilateral` and `.__init__()` doesn't really do anything differently for `Cuboid` than it already does for `Quadrilateral` since the parameter is the same (length, height(width)). The `.__init__()` of the superclass (`Quadrilateral`) will be called automatically.

So what is `super` doing?

The `super()` returns a **proxy object**, a substitute object that can call methods of the **base class** via **delegation**. This is called indirection (ability to reference base object with `super()`)

Since the indirection is computed at the runtime, we can use different base classes at different times .

Most of the time, we don't need to pass in any parameter in the `super` , it will default to the current base class. But we can if we want.

It takes 2 parameters. The first is the class whose parent's scope we're trying to resolve to, and the second argument is the object of interest to indicate which object to apply the scope to.

Consider a class hierarchy `A`, `B`, and `C` where each class is the parent of the one following it, and `a`, `b`, and `c` respective instances of each.

```
super(B, b)
# resolves to the scope of B's parent i.e. A
# and applies that scope to b, as if b was an instance of A

super(C, c)
# resolves to the scope of C's parent i.e. B
# and applies that scope to c

super(B, c)
# resolves to the scope of B's parent i.e. A
# and applies that scope to c
```

Under the hood `super()` returns a **bound method to** bound to the object, which gives the method the object's context such **as any instance attributes.**(c has all instance attributes of A in the last example).

In the example we use earlier, we can change the `super` call to:

```
class Square(Quadrilateral):
    def __init__(self, length):
        super(Square, self).__init__(length, length)
```

In Python 3, the `super(Square, self)` call is equivalent to the parameterless `super()` call, this is telling to lookup the `init` method is `Square` parent class, aka: `Quadrilateral` .

Now let's look at a more complex example, **multiple inheritance.**

```
class Quadrilateral:
    def __init__(self, length, height):
        self.length = length
        self.height = height

def area(self):
        return self.length * self.height

class Rectangle(Quadrilateral):
    def __init__(self, length, height):
        super().__init__(length, height)

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return π * radius* radius

class Cylinder(Circle, Square):
    def __init__(self, radius, height):
        self.radius = radius
        self.height = height

    def area(self):
        base_area1 = super().area() * 2
        base_length =  π * radius * 2
        base_area2 = super.area()

        return base_area2 + base_area1
```

The problem, though, is that both superclasses (`Circle` and `Square`) has a `.area()` method. So when we call the highlighted method above, which method of the parent class we are calling from?

Luckily, we have a **method resolution order (MRO).** Method Resolution Order (MRO) is the order in which methods should be inherited in the presence of multiple inheritance. You can view the MRO by using the `__mro__` attribute.

```
>>> Cylinder.__mro__
(<class '__main__.Cylinder'>, <class '__main__.Circle'>,
 <class '__main__.Square'>, <class '__main__.Quadrilateral'>,
 <class 'object'>)
```

So it's clear that when the `.area()` in `Circle` gets found, Python will call it instantly. Because `Circle.area` only takes one param (radius), Python throws an `AttributeError`.

What we can do is to make sure **the signatures of the method unique both by making sure the method names or method parameters unique.** So we can simply rename the `Circle.area` to be `Circle.cir_area` .

But we still have the problem with the multiple inheritance, as the inheritance chain grow longer, how can we make sure all the super methods get called **have a matching method and method arguments?**

There are several issues to be solved along the way according to super considered super:

- **the caller and callee need to have a matching argument signature**
- **the method being called by super() needs to exist**
- **and every occurrence of the method needs to use super()**

How can we make sure the first requirement is met?

We can use the python **unpacking methods** to get store all the arguments get passed in, getting all the keyword arguments stripped off for each level as required, and forwarding the rest to the next level class. When it reaches the end of the chain, `object` , there'd be no arguments left in the dictionary.

```
class Sneaky:
   def __init__(self, sneaky = false, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.sneaky = sneaky

class Person:
    def __init__(self, human = false,  *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.human = human

class Thief(Sneaky, Person):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

t = Thief(human = true, sneaky = true)
print(t.human)
# True
```

So we can see the in the first `Sneaky` class, `sneaky` will get stripped off, in `Person` , `human` will get stripped off, in `object` , nothing will be in the `**kwargs` .

If you want to make 100% safe just in case there are some params get passed into object. Then add a `Base` class to absorb the rest of the params before it reaches the final `object` . Make sure it inserted in the end of the MRO.

```
class Base(object):
    def __init__(self, *args, **kwargs): pass
```

Now the **second** question is, how can we make sure **the method being called by super() exist?**

Similar to the method above, we can implement a Base class to have this **methodA** if you worry about a subclass incorporates a class that has a **methodA**() method, so that this subclass won't call `super().method()A` on `object` without reaching `Base` class.

```
class Base:
    def methodA(self):
```

```
        # the delegation chain stops here
        assert not hasattr(super(), 'methodA')

class Thief(Base):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

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

Noted that this is very similar to implement the inheritance without using super keyword:

```
class ChildB(Base):
    def __init__(self):
        mro = type(self).mro()
        for next_class in mro[mro.index(ChildB) + 1:]: # slice to end
            if hasattr(next_class, '__init__'):
                next_class.__init__(self)
                break
```

The last problem is **to make sure that super() is called on all the methods along the chain. (normally we will call it on `__init__()`).**

Why is this necessary? So use the same example as above:

```
class Sneaky:
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

class Person:
    def __init__(self, human=True, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.human = human

class Thief(Sneaky, Person):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

t = Thief()
print(t.human)
# True
```

If the `super().__init__` call were removed from `Sneaky.__init__` , then `t.human` would raise `AttributeError: 'Thief' object has no attribute 'human'` , since the inheritance chain stops at `Sneaky` class without ever reaching to `Person` class which has a `human attribute` . Even if `Sneaky` class don't need to get anything from the inheritance (no args taken), it still need to do `super` in order to pass packed `kwargs` to the next class on the chain.

The only way to work in multi-inheritance is to make sure all the classes implement the super cooperatively.