Method Resolution Order (MRO)

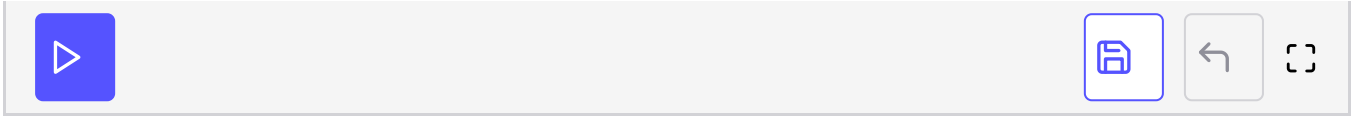
Method Resolution Order (MRO) is just a list of types that the class is derived from. So if you have a class that inherits from two other classes, you might think that it's MRO will be itself and the two parents it inherits from. However the parents also inherit from Python's base class: **object**. Let's take a look at an example that will make this clearer:

```
class X:
    def __init__(self):
        print('X')
        super().__init__()

class Y:
    def __init__(self):
        print('Y')
        super().__init__()

class Z(X, Y):
    pass


z = Z()
print(Z.__mro__)
```



Here we create 3 classes. The first two just print out the name of the class and the last one inherits from the previous two. Then we instantiate the class and also print out its MRO:

```
X
Y
(<class '__main__.Z'>, <class '__main__.X'>, <class '__main__.Y'>, <class 'object'>)
```

As you can see, when you instantiate it, each of the parent classes prints out its name. Then we get the Method Resolution Order, which is ZXY and object. Another good example to look at is to see what happens when you create a

class variable in the base class and then override it later:

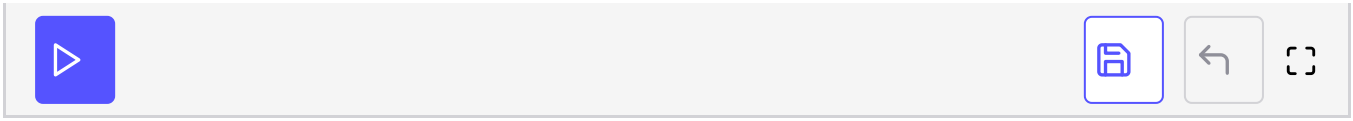
```
class Base:
    var = 5
    def __init__(self):
        pass

class X(Base):
    def __init__(self):
        print('X')
        super().__init__()

class Y(Base):
    var = 10
    def __init__(self):
        print('Y')
        super().__init__()

class Z(X, Y):
    pass


z = Z()
print(Z.__mro__)
print(super(Z, z).var)
```



So in this example, we create a Base class with a class variable set to

5. Then we create subclasses of our Base class: X and Y. Y overrides the Base class's class variable and sets it to 10. Finally we create class Z which inherits from X and Y. When we call super on class Z, which class variable will get printed? Try running this code and you'll get the following result:

```
X
Y
(<class '__main__.Z'>, <class '__main__.X'>, <class '__main__.Y'>, <class '__main__.Base'>, <
10
```

Let's parse this out a bit. Class Z inherits from X and Y. So when we ask it what it's **var** is, the MRO will look at X to see if it is defined. It's not there, so it moves on to Y. Y has it, so that is what gets returned. Try adding a class variable to X and you will see that it overrides Y because it is first in the Method Resolution Order.

There is a wonderful document that a fellow named Michele Simionato

There is a wonderful document that a fellow named Michele Simionato created that describes Python's Method Resolution Order in detail. You can check it out here: https://www.python.org/download/releases/2.3/mro/. It's a long read and you'll probably need to re-read portions of it a few times, but it explains MRO very well. By the way, you might note that this article is labeled as being for Python 2.3, but it still applies even in Python 3, even though the calling of super is a bit different now.

The super method was updated slightly in Python 3. In Python 3, super can figure out what class it is invoked from as well as the first argument of the containing method. It will even work when the first argument is not called **self**! This is what you see when you call **super()** in Python 3. In Python 2, you would need to call **super(ClassName, self)**, but that is simplified in Python 3.

Because of this fact, super knows how to interpret the MRO and it stores this information in the following magic propertie: **__thisclass__** and **__self_class__**. Let's look at an example:

```
class Base():
    def __init__(self):
        s = super()
        print(s.__thisclass__)
        print(s.__self_class__)
        s.__init__()

class SubClass(Base):
    pass

sub = SubClass()
```

Here we create a base class and assign the super call to a variable so we can find out what those magic properties contain. Then we print them out and initialize. Finally we create a SubClass of the Base class and instantiate the SubClass. The result that gets printed to stdout is this:

```
<class '__main__.Base'>
<class '__main__.SubClass'>
```

This is pretty cool, but probably not all that handy unless you're doing a metaclasses or mixin classes.

Wrapping Up

There are lots of interesting examples of super that you will see on the internet. Most of them tend to be a bit mind-bending at first, but then you'll figure them out and either think it's really cool or wonder why you'd do that. Personally I haven't had a need for super in most of my work, but it can be useful for forward compatibility. What this means is that you want your code to work in the future with as few changes as possible. So you might start off with single inheritance today, but a year or two down the road, you might add another base class. If you use super correctly, then this will be a lot easier to add. I have also seen arguments for using super for dependency injection, but I haven't seen any good, concrete examples of this latter use case. It's a good thing to keep in mind though.

The super function can be very handy or it can be really confusing or a little bit of both. Use it wisely and it will serve you well.