Super(type, obj)

Type: is the start point in MRO (method resolution order, like hierarchy) from where super () searches parent classes

Obj: is the object or subtype of the **type,** it is where the returned method binds to

EX:

```
class A(object):
    def __init__(self):
        self.word = "This is A !"

    def print(self):
        print(self.word)


class B(A):
    def __init__(self):
        self.word = "This is B"

    def print(self):
        print(self.word)


class C(B):
    def __init__(self):
        self.word = "This is C"

    def print(self):
        super(C, c).print()



a = A()
b = B()
c = C()
c.print()
```

Result:

>>> This is C

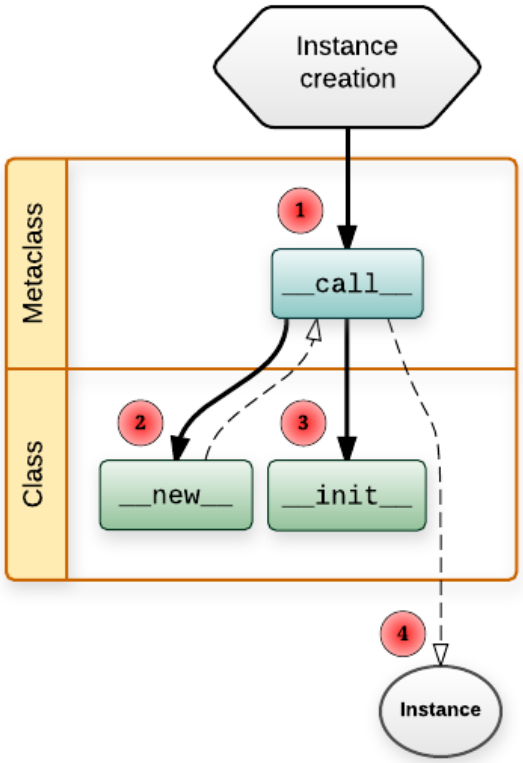
What happens here is that, super () will traverse the Class C parent classes (Class A and Class B) until an attribute print() is found, and bind this print() method to object c (an instance of Class C). In this case, it binds print() in b to c, we can see in this way: c pass c' self to b.print(self), that is why printing out "This is C"

If we change Class c into:

```
class C(B):
    def __init__(self):
        self.word = "This is C"

    def print(self):
        super(B, b).print()
```

Result would be : >>> This is B

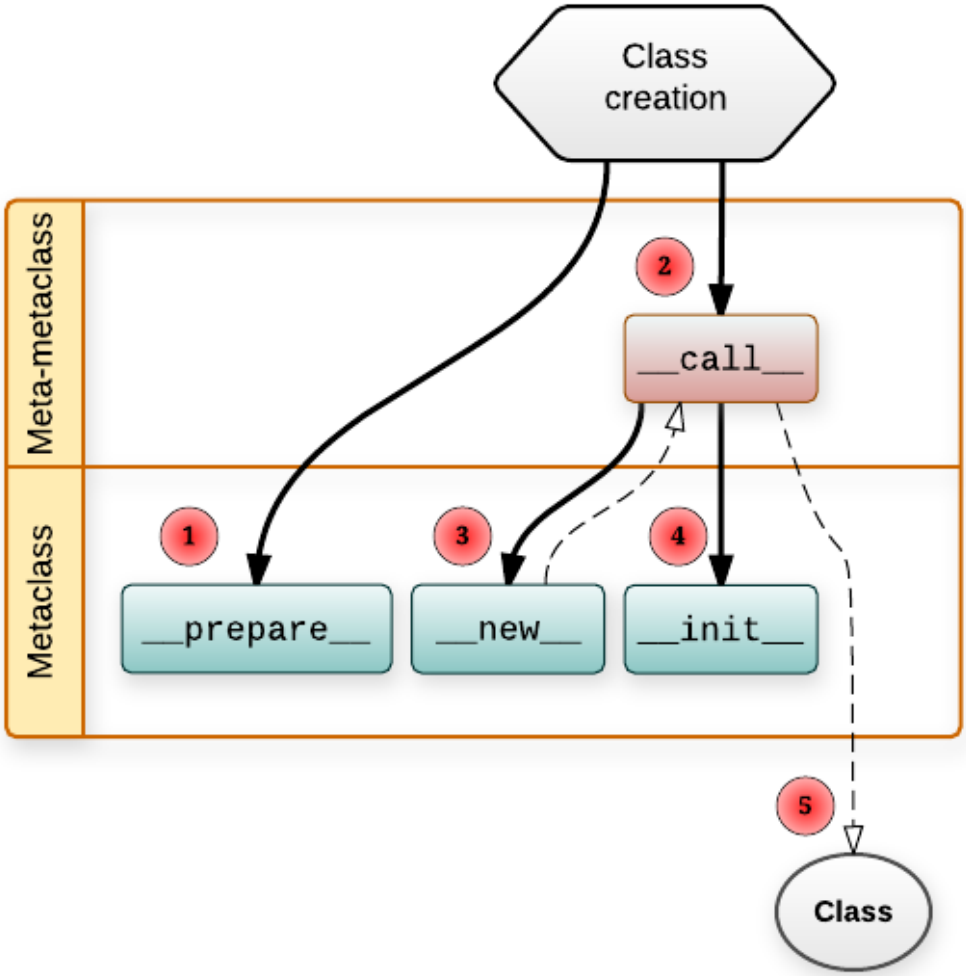
Because, it found print() in A, and bind the print() to b, and use self from b.

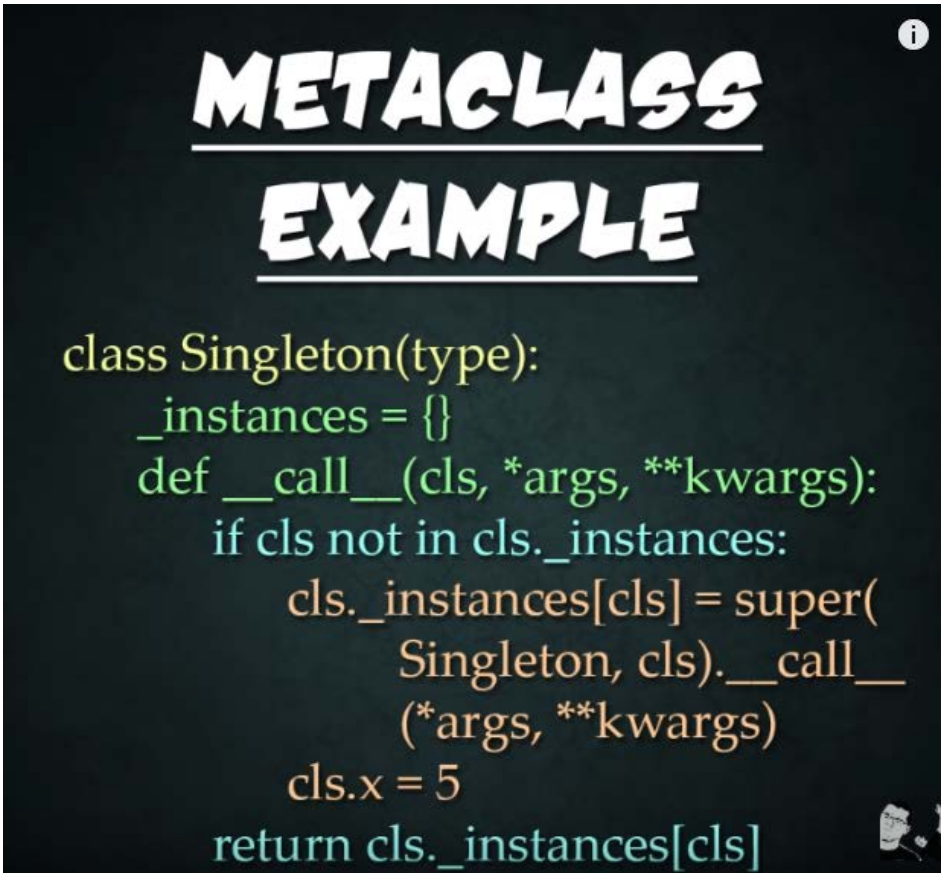


Singleton Metaclass

```
class MySingleton(type):  # A class can be used as a template for instances A type can be used as a template for other class
    _instances = {}

    def __call__(cls, *args, **kwargs):
        print(" I am inside the MySingleton __call__")
        if cls not in cls._instances:
            cls._instances[cls] = super(MySingleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]


class MyClass(metaclass=MySingleton):
    def __new__(cls, *args, **kwargs):...
    '''object.__new__(cls,[...]), will create an instance of cls
    If __new__() returns an instance of cls,  then the new instance's __init__() method will be invoked
    like __init__(self[, ...]), where self is the new instance and the remaining arguments are the same as were passed to __new__().

    If __new__() does not return an instance of cls, then the new instance's __init__() method will not be invoked.'''

    def __init__(self):
        print("start to initialize MyClass")
        self.x = 5


a = MyClass()
'''order is Singleton = type() --> MyClass = Singleton()--> a = MyClass(), that is why Singleton.__call__is called
=> a = MyClass() = Singleton()() = type()()()'''
# b = MyClass()
print(a.x)
# b.x = 10
# print(a.x)
```

```
'''when instantiate an instance of MyClass, MyClass will call its MetaClass __call___, now cls = MyClass
----->  check if cls (MyClass) is created previously
----> No ---> call MySingleton's parent class (type).__call__  ---> call MyClass.__new__ and return an instance --> MyClass.__init__
|
|---> Yes ---> return the instance that was stored in  _instance
pseudo implementation of type.__call__():

class type:
    def __call__(cls, *args, **kwarg):

        # ... a few things could possibly be done to cls here... maybe... or maybe not...

        # then we call cls.__new__() to get a new object
        obj = cls.__new__(cls, *args, **kwargs)

        # ... a few things done to obj here... maybe... or not...

        # then we call obj.__init__()
        obj.__init__(*args, **kwargs)

        # ... maybe a few more things done to obj here

        # then we return obj
        return obj
'''
```




METACLASS
EXAMPLE
class Singleton(type):
_instances = {}
def __call__(cls, *args, **kwargs):
if cls not in cls._instances:
cls._instances[cls] = super(
Singleton, cls).__call__
(*args, **kwargs)
cls.x = 5
return cls._instances[cls]