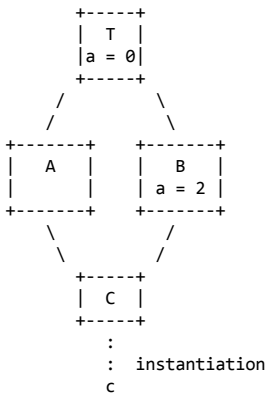
super() - returns MRO and multiple inheritance (MI) **proxy objects**, *not a superclass*.

`super()` is useful for accessing inherited methods *that have been overridden* in a class. The **search order is same as that used by getattr(),** but **current class itself is skipped.** The `__mro__` attribute of the type lists the method resolution search order used by both getattr() and super(). The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

_____There are two typical use cases for super: _____
1. ===== In a class hierarchy with **single inheritance**, super can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of super in other programming languages.

2. ===== To support cooperative **multiple inheritance** in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement "diamond diagrams" with multiple implementations of same method. Good design dictates this method to have same calling signature in every case:
- because the order of calls is determined at runtime,
- because that order adapts to changes in the class hierarchy, and
- because that order can include sibling classes that are unknown prior to runtime.

Super() is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order supporting multiple inheritance.




```
>>> class T(object): a = 0      # without object - it'll be an old-style class
>>> class A(T): pass
>>> class B(T): a = 2
>>> class C(A, B): pass
>>> c = C()
>>> print( super(C, c).a ) -> 2
```

`super(C, c).a` walks through the method resolution order of the class of c (which is C) and retrieves the attribute from the *first class above C which defines it.* In this example the MRO of C is [C, A, B, T, object], so B is the first class above C which defines `.a`

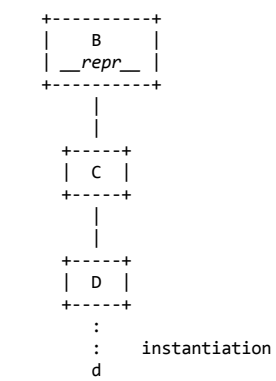
Informally speaking, a proxy is an object with the ability to dispatch to methods of other objects via delegation. Technically, `super` is a class overriding the `__getattribute__` method.

** Instances of super are proxy objects providing access to the methods in the MRO **
The dispatch is done in such a way that
`super(cls, instance-or-subclass).method(*args, **kwargs)`
corresponds more or less to
`right-method-in-the-MRO-applied-to(instance-or-subclass, *args, **kwargs)`

_____ Bound/Unbound methods from super _____

```
>>> print super.__doc__
super(type, obj) -> bound super object; requires isinstance(obj, type)
super(type) -> unbound super object
super(type, type2) -> bound super object; requires issubclass(type2, type)
Typical use to call a cooperative superclass method:
    class C(B):
        def meth(self, arg):
            super(C, self).meth(arg) # same as super().meth(arg)
```

```
_____________________________________________________
B = type('B', (object,), {
    '__repr__': lambda self: "<instance of %s>" % self.__class__.__name__})
>>> b = B();  b -> <instance of B>
```



Zero arguments (magic):
Returns **bound**-super-object and consequently, **bound-methods**,
and equal to `super(cls, cls-instance)`. It will *only* work inside a class: it is a shortcut.

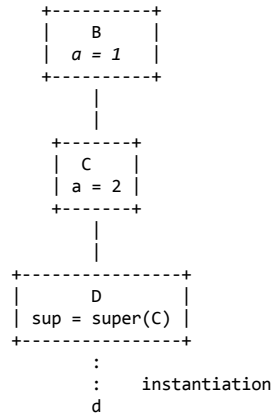
One argument:
Returns **unbound** proxy, which *does not* dispatch to any parent method:
```
>>> super(C).__repr__ -> <method-wrapper '__repr__' of super object at 0x10>
```

Two arguments: *First argument is always the base **class**.*
Second argument can be (a: an instance, returning bound methods), or (b: same class | subclass, returning unbounds) of the first argument:
*a) In case of **instance**, a **BOUND method** to be returned.*
```
>>> super(C, d).__repr__ -> <bound method D.__repr__ of <instance of D>>
```

*b) In case of (same class | **subclass**), an **UNBOUND method** to be returned.*
```
>>> super(C, C).__repr__ -> <unbound method C.__repr__> # same class
>>> super(C, D).__repr__ -> <unbound method D.__repr__> # subclass
----------------------------------------------------------------
>>> super(C, d).__repr__() == super(C, D).__repr__(d) -> True
```

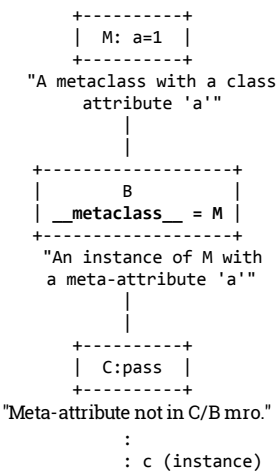
** **Unbound proxy object can not be used to dispatch to the upper methods in the hierarchy.**
In order to dispatch properly over inheritance chain, *unbound object must be turned into bound* one via **descriptor** protocol. *# def __get__(self, instance, cls): ...*

Converting unbound `super(C)` proxy into a **bound (**to an instance `d`**)** one is easy:
```
>>> boundsuper = super(C).__get__(d, C)   # same as super(C, d)
>>> boundsuper.__repr__   ->  <bound method D.__repr__ of <instance of D>>
>>> _()   ->  '<instance of D>'
```



Unbound syntax `super(C)` **or** `super(C, C)` **or** `super(C, D)` **<u>does not</u> return unbound methods.** It is intended to be used as an attribute in other classes. Then, descriptor's magic will automatically convert unbound objects into bound:
```
>>> d.sup.a   ->  1
```
Works because `super()`, by defining a `__get__` method, has the ability to be a plain old descriptor.
Calling `d.sup` results into `super.__get__` being invoked, providing `__get__` with *instance and cls (or owner)* of the **caller**: `super(C).__get__(d, D)`, returning
`<super: <class 'C'>, <D object>>` bound proxy, which then dispatches attribute/method lookups to inheritance chain. It produces exactly the same result as `super(C, d).a` and retrieves everything that was defined in any parent of `C` (in our case, in `B`)
```
>>> D.__dict__['sup'].__get__(d, D).a == super(C, d).a -> True
```

Quick wrap-up:
```
>>> super(D) -> <super: <class 'D'>, NULL>          # unbound super object
>>> super(D, d) -> <super: <class 'D'>, <D object>> # bound (to instance) super object
----------------------------------------------------------------
```



MetaClass.attrs DIFFER FROM Class.attrs:
they are not inherited by instances of instances.
```
>>> print B.a, C.a      -> 1, 1
>>> print super(C, C).a  -> [..] AttributeError: 'super' object has no attribute 'a'
>>> print C().a  -> [..] AttributeError: 'C' object has no attribute 'a' # same with B().a
```

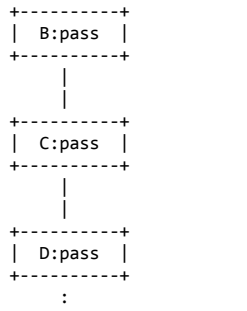
This is a case where `super()` *is doing the right* thing, since `.a` ***is not inherited*** from `B`, but coming from the metaclass `M`, so `.a` is ***not*** **in the MRO** of `C`. A similar thing happens for the `__name__` attribute *(the fact that it is a descriptor and not a plain attribute does not matter)*, so super()'s *correct* behaviour may seem surprising at first.

`__name__` is also a *(Get,Set) Descriptor* defined @ `type` MetaClass and not a plain Class.attribute.
```
> super(C, C()).__doc__ -> "A metaclass with a class attribute 'a'."
> super(C, c).__name__ -> [..] AttributeError: 'super' object has no attribute
'__name__'
```
*** In general, `super()` should be avoided when working with MetaClass.Attrs

```
>>> vars(super).keys() -> ['__thisclass__', '__new__', '__self_class__', '__self__',
'__getattribute__', '__repr__', '__doc__', '__init__', '__get__']
```

Amongst others, *super* objects have some very special attributes, *not presented elsewhere*:

- `__thisclass__` (the first argument passed to *super*)
- `__self__` (the second argument passed to *super*, `or None`), and
- `__self_class__` (`__self__ or None`)




```
__self__ is an instance of __thisclass__ ?     # super(__thisclass__, __self__=None)
    YES:  __self_class__ -> the class of __self__
    NO:   __self_class__ -> __self__

>>> instance_bound = super(C, d) # instance-bound syntax
instance_bound.__thisclass__  -> C
instance_bound.__self__       -> d       # isinstance(d, C) -> True
instance_bound.__self_class__ -> D        # __self__ is an instance of __thisclass__

>>> class_bound = super(C, D)    # class-bound syntax
class_bound.__thisclass__     -> C
class_bound.__self__          -> D       # isinstance(D, C) -> False
class_bound.__self_class__    -> D        # __self__ is NOT an instance of __thisclass__

>>> unbound = super(C)           # unbound syntax
unbound.__thisclass__         -> C
unbound.__self__              -> None    # isinstance(None, C) -> False
unbound.__self_class__        -> None    # __self__ is NOT an instance of __thisclass__
```

Argument passing in multiple inheritance cooperative methods

All cooperative methods should have one, common compatible signature.

Unknown (wrong usage):

```
class A(object):
    def __init__(self):
        print "A", self
    >   super(A, self).__init__()

class B(object):
    def __init__(self):
        print "B", self
  >>>  super(B, self).__init__()
super(B, C()).__init__
-- (behind-scenes) -----------
class C(A, B):
    def __init__(self):
        print "C", self
        A.__init__(self)
    >   B.__init__(self)

>>> C()
C <__main__.C object at 0x...>
A <__main__.C object at 0x...>
B <__main__.C object at 0x...>
B <__main__.C object at 0x...>
<__main__.C object at 0x...>
```

Proper: super(C, self).__init__() ::::**HACK**
Skip A initialization: Call only B.__init__(self)

Per Class (wrong expectations):

```
class A(object):
    def __init__(self):
        print "A", self

class B(object):
    def __init__(self):
        print "B", self



-- (behind-scenes) -----------
class C(A,B):
    def __init__(self):
        print "C", self
  >     super(C, self).__init__()

>>> c = C()
C <__main__.C object at 0x...>
A <__main__.C object at 0x...>
<__main__.C object at 0x...>
```

* B will not be called.

Library authors should always document their usage of `super`: classes were intended to be cooperative (using `super` behind) or not.

Cooperative, propagation

```
class A(object):
    def __init__(self):
        print "A", self
  >     super(A, self).__init__()

class B(object):
    def __init__(self):
        print "B", self
  >     super(B, self).__init__()

-- (behind-scenes) -----------
class C(A,B):
    def __init__(self):
        print "C", self
  >     super(C, self).__init__()

>>> c = C()
C <__main__.C object at 0x...>
A <__main__.C object at 0x...>
B <__main__.C object at 0x...>
<__main__.C object at 0x...>
```

```
>>> class A(object): def __init__(self, a): super(A, self).__init__()      # object.__init__ cannot take arguments
>>> class B(object): def __init__(self, a): super(B, self).__init__()      # object.__init__ cannot take arguments
>>> class C(A, B):   def __init__(self, a): super(C, self).__init__(a)     # A.__init__ takes one argument
```

```
class Inherited(object):
    def __init__(self):
        print 'Inherited'
```

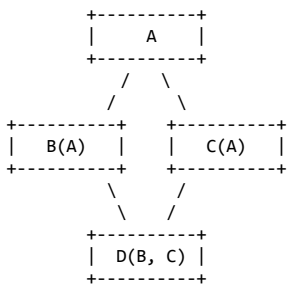
```
class A(Inherited):
    def __init__(self, a=None):
        print 'A with a=%s' % a
        super(A, self).__init__(a)   # calls B.__init__ because C(A, B)
```

```
class C(A, B):
    def __init__(self):
        print 'C'
        super(C, self).__init__()
```

```
class B(Inherited):
    def __init__(self, a):
        print 'B with a=%s' % a
        super(B, self).__init__()
```

```
>>> c = C() -> C | (A with a=None) | (B with a=None) | Inherited
But if C(B, A) not C(A, B) -> [...] TypeError: __init__() takes exactly 2 arguments (1 given)
```

Change all classes to inherit from a custom `xObject` class with an `__init__` accepting all kind of arguments, otherwise there is no way to solve it in Python 2.6+. Inheritance makes code heavily coupled and difficult to follow (*spaghetti inheritance*). There's hardly a single real life problem that could not be solved with **single inheritance + composition/delegation** in a better and more maintainable way, than using *multiple inheritance*.

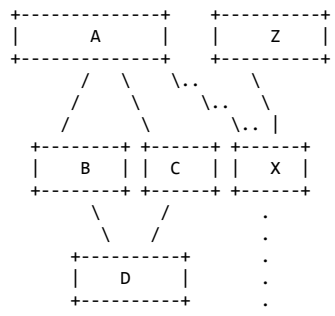



```
class A(object):
    def m(self):
        print 'A'
class D(B, C):
    def m(self):
        super(D, self).m()
        print 'D'
```

```
class B(A):
    def m(self):
        super(B, self).m()
        print 'B'
class C(A):
    def m(self):
        super(C, self).m()
        print 'C'
```

```
>>> D().m() -> A C B D      # looks good so far
```

Now introduce classes Z & X:




```
class Z(object):
    def m(self):
        print 'Z'
class X(A, Z):
    def m(self):
        super(X, self).m()
        print 'X'
```

```
>>> X().m() -> A X"   # Z.m was not called...
```

That is because A is not calling super.
Change class A to call super?..

```
class A(object):
    def m(self):
    >   super(A, self).m()
        print 'A'
```

```
>>> X().m() -> Z A X
>>> D().m() -> [..] Traceback:
super object has no attribute 'm'
```

1. Introduce a placeholder base class Y with a dummy .m() on top of the hierarchy: A(Y), Z(Y)

```
class Y(object):
    def m(self):
        Pass
```

Without changing the hierarchy, the trick is to use a custom super(), which ignores attribute errors:

```
class mysuper(super):
    def __getattribute__(self,name):
        try:
            return super.__getattribute__(self,name)
        except AttributeError: # returns a do-nothing method
            return lambda *args, **kw: None
```