Things to know about super

author: Michele Simionato

date: August 2008

This document is the sum of four blog post appeared on Artima and converted into PDF form for readers' convenience:

- http://www.artima.com/weblogs/viewpost.jsp?thread=236275
- http://www.artima.com/weblogs/viewpost.jsp?thread=236278
- http://www.artima.com/weblogs/viewpost.jsp?thread=237121
- http://www.artima.com/weblogs/viewpost.jsp?thread=281127

Contents

Foreword

I begun programming with Python in 2002, just after the release of Python 2.2. That release was a major overhaul of the language: new-style classes were introduced, the way inheritance worked changed and the builtin `super` was introduced. Therefore, you may correctly say that I have worked with `super` right from the beginning; still, I never liked it and over the years I have discovered more and more of its dark corners.

In 2004 I decided to write a comprehensive paper documenting `super` pitfalls and traps, with the goal of publishing it on the Python web site, just as I had

published my essay on multiple inheritance and the Method Resolution Order. With time the paper grew longer and longer but I never had the feeling that I had covered everything I needed to say: moreover I have a full time job, so I never had the time to fully revise the paper as a whole. As a consequence, four years have passed and the paper is still in draft status. This is a pity, since it documents issues that people encounter and that regularly come out on the Python newsgroups and forums.

Keeping the draft sitting on my hard disk is doing a disservice to the community. Still, I lack to time to finish it properly. To come out from the impasse, I decided to split the long paper in a series of short blog posts, which I do have the time to review properly. Moreover people are free to post comments and corrections in case I am making mistakes (speaking about `super` this is always possible). Once I finish the series, I may integrate the corrections, put it together again and possibly publish it as whole on the Python website. In other words, in order to finish the task, I am trying the strategies of *divide et conquer* and *release early, release often*. We will see how it goes.

Introduction

`super` is a Python built-in, first introduced in Python 2.2 and slightly improved and fixed in later versions, which is often misunderstood by the average Python programmer. One of the reasons for that is the poor documentation of `super`: at the time of this writing (August 2008) the documentation is incomplete and in some parts misleading and even wrong. For instance, the standard documentation (even for the new 2.6 version http://docs.python.org/dev/library/functions.html#super) still says:

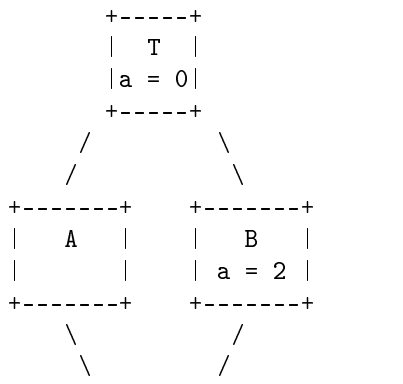
```
super(type[, object-or-type])
  Return the superclass of type. If the second argument is omitted the
  super object returned is unbound. If the second argument is an object,
  isinstance(obj, type) must be true. If the second argument is a type,
  issubclass(type2, type) must be true. super() only works for new-style
  classes.
```

[UPDATE: the final version of Python 2.6 has a better documentation for `super`, as a direct consequence of this post ;)]. The first sentence is just plain wrong: `super` does not return the superclass. There is no such a thing as *the superclass* in a Multiple Inheritance (MI) world. Also, the sentence about *unbound* is misleading, since it may easily lead the programmer to think about bound and unbound methods, whereas it has nothing to do with that concept. IMNSHO `super` is one of the most tricky and surprising Python constructs, and we absolutely need a document to shed light on its secrets. The present paper is a first step in this direction: it aims to tell you the *truth* about `super`. At least the amount of truth I have discovered with my experimentations, which is certainly not the whole truth ;)

A fair warning is in order here: this document is aimed at expert Pythonistas. It assumes you are familiar with new-style classes and the Method Resolution Order (MRO); moreover a good understanding of descriptors would be extremely useful. Some parts also require good familiarity with metaclasses. All in all, this paper is not for the faint of heart ;)

There is no superclass in a MI world

Readers familiar will single inheritance languages, such as Java or Smalltalk, will have a clear concept of superclass in mind. This concept, however, has *no useful meaning* in Python or in other multiple inheritance languages. I became convinced of this fact after a discussion with Bjorn Pettersen and Alex Martelli on comp.lang.python in May 2003 (at that time I was mistakenly thinking that one could define a superclass concept in Python). Consider this example from that discussion:




```
>>> class T(object):
...     a = 0

>>> class A(T):
...     pass

>>> class B(T):
...     a = 2

>>> class C(A,B):
...     pass

>>> c = C()
```

What is the superclass of `C`? There are two direct superclasses (i.e. bases) of `C`: `A` and `B`. `A` comes before `B`, so one would naturally think that the superclass of `C` is `A`. However, `A` inherits its attribute `a` from `T` with value `a=0`: if `super(C,c)` was returning the superclass of `C`, then `super(C,c).a` would return 0. This is NOT what happens. Instead, `super(C,c).a` walks trought the method resolution order of the class of `c` (i.e. `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` and `super(C,c).a` correctly returns the value 2, not 0:

```
>>> super(C,c).a
2
```

You may call `A` the superclass of `C`, but this is not a useful concept since the methods are resolved by looking at the classes in the MRO of `C`, and not by looking at the classes in the MRO of `A` (which in this case is `[A,T, object]` and does not contain `B`). The whole MRO is needed, not just the first superclass.

So, using the word *superclass* in the standard docs is misleading and should be avoided altogether.

Bound and unbound (super) methods

Having established that `super` cannot return the mythical superclass, we may ask ourselves what the hell it is returning ;) The truth is that `super` returns proxy objects.

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, **kw)`

corresponds more or less to

`right-method-in-the-MRO-applied-to(instance-or-subclass, *args, **kw)`

There is a caveat at this point: the second argument can be an instance of the first argument, or a subclass of it. In the first case we expect a *bound* method to be returned and in the second case and *unbound* method to be returned. This is true in recent versions of Python: for instance, in this example

```
>>> class B(object):
...     def __repr__(self):
...         return "<instance of %s>" % self.__class__.__name__

>>> class C(B):
...     pass

>>> class D(C):
...     pass

>>> d = D()
```

you get

```
>>> print super(C, d).__repr__
<bound method D.__repr__ of <instance of D>>
```

and

```
>>> print super(C, D).__repr__
<unbound method D.__repr__>
```

However, if you are still using Python 2.2 (there are unlucky people forced to use old versions) your should be aware that `super` had a bug and `super(<class>, <subclass>).method` returned a *bound* method, not an unbound one:

```
>> print super(C, D).__repr__ # in Python 2.2
<bound method D.__repr__ of <class '__main__.D'>>
```

That means that in Python 2.2 you get:

```
>> print super(C, D).__repr__() # in Python 2.2
<instance of type>
```

`D`, seen as an instance of the (meta)class `type`, is being passed as first argument to `__repr__`. This has been fixed in Python 2.3+, where you correctly get a `TypeError`:

```
>>> print super(C, D).__repr__() # the same as B.__repr__()
Traceback (most recent call last):
  ...
TypeError: unbound method __repr__() must be called with D instance as first
argument (got nothing instead)
```

The point is subtle, but usually one does not see problems since typically `super` is invoked on instances, not on subclasses, and in this case it works correctly in all Python versions:

```
>>> print super(C, d).__repr__()
<instance of D>
```

When I was using Python 2.2, due to the bug just discussed, and due to the `super` docstring

```
>>> print super.__doc__
super(type) -> unbound super object
super(type, obj) -> bound super object; requires isinstance(obj, type)
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)
```

I got the impression that in order to get unbound methods I needed to use the unbound `super` object. This is actually untrue. To understand how bound/unbound methods work we need to talk about descriptors.

`super` and descriptors

Descriptors (more properly I should speak of the descriptor protocol) were introduced in Python 2.2 by Guido van Rossum. Their primary motivation was technical, since they were needed to implement the new-style object system. Descriptors were also used to introduce new standard concepts in Python, such as classmethods, staticmethods and properties. Moreover, according to the traditional transparency policy of Python, descriptors were exposed to the application programmer, giving him/her the freedom to write custom descriptors. Any serious Python programmer should have a look at descriptors: luckily they are now very well documented (which was not the case when I first studied them

:-/) thanks to the beautiful essay of Raimond Hettinger. You should read it before continuing this article, since it explains all the details. However, for the sake of our discussion of `super`, it is enough to say that a *descriptor class* is just a regular new-style class which implements a `.__get__` method with signature `__get__(self, obj, objtyp=None)`. A *descriptor object* is just an instance of a descriptor class.

Descriptor objects are intended to be used as attributes (hence their complete name attribute descriptors). Suppose that `descr` is a given descriptor object used as attribute of a given class C. Then the syntax `C.descr` is actually interpreted by Python as a call to `descr.__get__(None, C)`, whereas the same syntax for an instance of C corresponds to a call to `descr.__get__(c, type(c))`.

Since the combination of descriptors and super is so tricky, the core developers got it wrong in different versions of Python. For instance, in Python 2.2 the only way to get the unboud method `__repr__` is via the descriptor API:

```
>> super(C, d).__repr__.__get__(None, D) # Python 2.2
<unbound method D.__repr__>
```

You may check that it works correctly:

```
>> print _(d)
<instance of D>
```

In Python 2.3 one can get the unbond method by using the `super(cls, subcls)` syntax, but the syntax `super(C, d).__repr__.__get__(None, D)` also works; in Python 2.4+ instead the same syntax returns a *bound* method, not an unbound one:

```
>>> super(C, d).__repr__.__get__(None, D) # in Python 2.4+
<bound method D.__repr__ of <instance of D>>
```

The core developers changed the behavior again, making my life difficult while I was writing this paper :-/ I cannot trace the history of the bugs of `super` here, but if you are using an old version of Python and you find something weird with `super`, I advice you to have a look at the Python bug tracker before thinking you are doing something wrong. In this case, to be correct, the change is not in `super`, but in the descriptor implementation. In Python 2.2-2.3 you could get an unbound method from a bound one as follows:

```
>> d.__repr__.__get__(None, D) # in Python 2.2-2.3
<unbound method D.__repr__>
```

In Python 2.4 that does not work anymore:

```
>>> d.__repr__.__get__(None, D) # in Python 2.4+
<bound method D.__repr__ of <instance of D>>
```

Still, you can get the unbound method by passing for the underlying function first:

```
>>> d.__repr__.im_func.__get__(None, D) # in Python 2.4+
<unbound method D.__repr__>
```

When working with `super`, virtually everybody uses the two-argument syntax `super(type, object-or-type)` which returns a *bound super object* (bound to the second argument, an instance or a subclass of the first argument). However, `super` also supports a single-argument syntax `super(type)` - fortunately very little used - which returns an *unbound super object*. Here I argue that unbounds super objects are a wart of the language and should be removed or deprecated (and Guido agrees).

The secrets of unbound super objects

Let me begin by clarifying a misconception about bound super objects and unbound super objects. From the names, you may think that if `super(C, c).meth` returns a bound method then `super(C).meth` returns an unbound method: however, this is a *wrong* expectation. Consider for instance the following example:

```
>>> class B1(object):
...     def f(self):
...         return 1
...     def __repr__(self):
...         return '<instance of %s>' % self.__class__.__name__
...
>>> class C1(B1): pass
...
```

The unbound super object `super(C1)` does not dispatch to the method of the superclass:

```
>>> super(C1).f
Traceback (most recent call last):
  ...
AttributeError: 'super' object has no attribute 'f'
```

i.e. `super(C1)` is not a shortcut for the bound super object `super(C1, C1)` which dispatches properly:

```
>>> super(C1, C1).f
<unbound method C1.f>
```

Things are more tricky if you consider methods defined in `super` (remember that `super` is class which defines a few methods, such as `__new__`, `__init__`, `__repr__`, `__getattribute__` and `__get__`) or special attributes inherited from `object`. In our example `super(C1).__repr__` does not give an error,

```
>>> print super(C1).__repr__() # same as repr(super(C1))
<super: <class 'C1'>, NULL>
```

but it is not dispatching to the `__repr__` method in the base class `B1`: instead, it is retrieving the `__repr__` method defined in `super`, i.e. it is giving something completely different.

Very tricky. You *cannot* use unbound `super` object to dispatch to the the upper methods in the hierarchy. If you want to do that, you must use the two-argument syntax `super(cls, cls)`, at least in recent versions of Python.

We said before that Python 2.2 is buggy in this respect, i.e. `super(cls, cls)` returns a *bound* method instead of an *unbound* method:

```
>> print super(C1, C1).__repr__ # buggy behavior in Python 2.2
<bound method C1.__repr__ of <class '__main__.C1'>>
```

Unbound super objects must be turned into bound objects in order to make them to dispatch properly. That can be done via the descriptor protocol. For instance, I can convert `super(C1)` in a super object bound to `c1` in this way:

```
>>> c1 = C1()
>>> boundsuper = super(C1).__get__(c1, C1) # this is the same as super(C1, c1)
```

Now I can access the bound method `c1.f` in this way:

```
>>> print boundsuper.f
<bound method C1.f of <instance of C1>>
```

The unbound syntax is a mess

Having established that the unbound syntax does not return unbound methods one might ask what its purpose is. The answer is that `super(C)` is intended to be used as an attribute in other classes. Then the descriptor magic will automatically convert the unbound syntax in the bound syntax. For instance:

```
>>> class B(object):
...     a = 1
>>> class C(B):
...     pass
>>> class D(C):
...     sup = super(C)
>>> d = D()
>>> d.sup.a
1
```

This works since `d.sup.a` calls `super(C).__get__(d,D).a` which is turned into `super(C, d).a` and retrieves `B.a`.

There is a single use case for the single argument syntax of `super` that I am aware of, but I think it gives more troubles than advantages. The use case is the implementation of *autosuper* made by Guido on his essay about new style classes.

The idea there is to use the unbound super objects as private attributes. For instance, in our example, we could define the private attribute `__sup` in the class `C` as the unbound super object `super(C)`:

```
>>> C._C__sup = super(C)
```

With this definition inside the methods the syntax `self.__sup.meth` can be used as an alternative to `super(C, self).meth`. The advantage is that you avoid to repeat the name of the class in the calling syntax, since that name is hidden in the mangling mechanism of private names. The creation of the `__sup` attributes can be hidden in a metaclass and made automatic. So, all this seems to work: but actually this *not* the case.

Things may wrong in various cases, for instance for classmethods, as in this example:

```
def test__super():
  "These tests work for Python 2.2+"

  class B(object):
      def __repr__(self):
          return '<instance of %s>' % self.__class__.__name__
      def meth(cls):
          print "B.meth(%s)" % cls
      meth = classmethod(meth) # I want this example to work in older Python

  class C(B):
      def meth(cls):
          print "C.meth(%s)" % cls
          cls.__super.meth()
      meth = classmethod(meth)

  C._C__super = super(C)

  class D(C):
      pass

  D._D__super = super(D)

  d = D()

  try:
      d.meth()
  except AttributeError, e:
      print e
  else:
      raise RuntimeError('I was expecting an AttributeError!')
```

The test will print a message `'super' object has no attribute 'meth'`. The issue here is that `self.__sup.meth` works but `cls.__sup.meth` does not, unless the `__sup` descriptor is defined at the metaclass level.

So, using a `__super` unbound super object is not a robust solution (notice that everything would work by substituting `self.__super.meth()` with `super(C,self).meth()` instead). In Python 3.0 all this has been resolved in a much better way.

If it was me, I would just remove the single argument syntax of `super`, making it illegal. But this would probably break someone code, so I don't think it will ever happen in Python 2.X. I did ask on the Python 3000 mailing list about removing unbound super objects (the title of the thread was *let's get rid of unbound super*) and this was Guido's reply:

> *Thanks for proposing this -- I've been scratching my head wondering what the use of unbound super() would be. :-) I'm fine with killing it -- perhaps someone can do a bit of research to try and find out if*

there are any real-life uses (apart from various auto-super clones)?
--- Guido van Rossum

Unfortunaly as of now unbound super objects are still around in Python 3.0, but you should consider them morally deprecated.

Bugs of unbound super objects in earlier versions of Python

The unbound form of `super` is pretty buggy in Python 2.2 and Python 2.3. For instance, it does not play well with pydoc. Here is what happens with Python 2.3.4 (see also bug report 729103):

```
>>> class B(object): pass
...
>>> class C(B):
...     s=super(B)
...
>>> help(C)
Traceback (most recent call last):
  ...
  ... lots of stuff here
  ...
File "/usr/lib/python2.3/pydoc.py", line 1198, in docother
   chop = maxlen - len(line)
TypeError: unsupported operand type(s) for -: 'type' and 'int'
```

In Python 2.2 you get an AttributeError instead, but still `help` does not work.

Moreover, an incompatibility between the unbound form of `super` and doctest in Python 2.2 and Python 2.3 was reported by Christian Tanzer (902628). If you run the following

```
class C(object):
    pass

 C.s = super(C)

if __name__ == '__main__':
    import doctest, __main__; doctest.testmod(__main__)
```

you will get a

```
TypeError: Tester.run__test__: values in dict must be strings, functions or
classes; <super: <class 'C'>, NULL>
```

Both issues are not directly related to `super`: they are bugs with the `inspect` and `doctest` modules not recognizing descriptors properly. Nevertheless, as usual, they are exposed by `super` which acts as a magnet for subtle bugs. Of course, there may be other bugs I am not aware of; if you know of other issues, just add a comment here.

Appendix

In this appendix I give some test code for people wanting to understand the current implementation of `super`. Starting from Python 2.3+, `super` defines the following attributes:

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

In particular super objects have attributes `__thisclass__` (the first argument passed to `super`) `__self__` (the second argument passed to `super` or `None`) and `__self_class__` (the class of `__self__`, `__self__` or `None`). You may check that the following assertions hold true:

```
def test_super():
    "These tests work for Python 2.3+"

    class B(object):
         pass

    class C(B):
        pass

    class D(C):
       pass

    d = D()

    # instance-bound syntax
    bsup = super(C, d)
    assert bsup.__thisclass__ is C
    assert bsup.__self__ is d
    assert bsup.__self_class__ is D

    # class-bound syntax
    Bsup = super(C, D)
    assert Bsup.__thisclass__ is C
    assert Bsup.__self__ is D
    assert Bsup.__self_class__ is D

    # unbound syntax
    usup = super(C)
    assert usup.__thisclass__ is C
```

```
assert usup.__self__ is None
assert usup.__self_class__ is None
```

The tricky point is the `__self_class__` attribute, which is the class of `__self__` only if `__self__` is an instance of `__thisclass__`, otherwise `__self_class__` coincides with `__self__`. Python 2.2 was buggy because it failed to make that distinction, so it could not distinguish bound and unbound methods correctly.

Working with `super` is tricky, not only because of the quirks and bugs of `super` itself, but also because you are likely to run into some gray area of the Python language itself. In particular, in order to understand how `super` works, you need to understand really well how attribute lookup works, including the tricky cases of special attributes and metaclass attributes. Moreover, even if you know perfectly well how `super` works, interacting with a third party library using (or not using) `super` is still non-obvious. At the end, I am led to believe that the problem is not `super`, but the whole concept of multiple inheritance and cooperative methods in Python.

Special attributes are special

This issue came up at least three or four times in the Python newsgroup, and there are various independent bug reports on sourceforge about it, you may face it too. Bjorn Pettersen was the first one who pointed out the problem to me (see also bug report 729913): the issue is that

`super(MyCls, self).__getitem__(5)`

works, but not

`super(MyCls, self)[5]`.

The problem is general to all special methods, not only to `__getitem__` and it is a consequence of the implementation of attribute lookup for special methods. Clear explanations of what is going on are provided by Michael Hudson as a comment to the bug report: 789262 and by Raymond Hettinger as a comment to the bug report 805304. Shortly put, this is not a problem of `super` per se, the problem is that the special call `x[5]` (using `__getitem__` as example) is converted to `type(x).__getitem__(x,5)` *only if* `__getitem__` is explicitely defined in `type(x)`. If `type(x)` does not define `__getitem__` directly, but only indirectly via delegation (i.e. overriding `__getattribute__`), then the second form works but not the first one.

This restriction will likely stay in Python, so it has to be considered just a documentation bug, since nowhere in the docs it is mentioned that special calling syntaxes (such as the `[]` call, the `iter` call, the `repr` call, etc. etc.) are special and bypass `__getattribute__`. The advice is: just use the more explicit form and everything will work.

`super` does not work with meta-attributes

Even when `super` is right, its behavior may be surprising, unless you are deeply familiar with the intricacies of the Python object model. For instance, `super` does not play well with the `__name__` attribute of classes, even if it works well for the `__doc__` attribute and other regular class attributes. Consider this example:

```
>>> class B(object):
...     "This is class B"
```

```
...
>>> class C(B):
...     pass
...
```

The special (class) attribute `__doc__` is retrieved as you would expect:

```
>>> super(C, C).__doc__ == B.__doc__
True
```

On the other hand, the special attribute `__name__` is not retrieved correctly:

```
>>> super(C, C).__name__ # one would expect it to be 'B'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'super' object has no attribute '__name__'
```

The problem is that `__name__` is not just a plain class attribute: it is actually a *getset descriptor* defined on the metaclass `type` (try to run `help(type.__dict__['__name__'])` and you will see it for yourself). More in general, `super` has problems with meta-attributes, i.e. class attributes of metaclasses.

Meta-attributes differs from regular attributes since they are not transmitted to the instances of the instances. Consider this example:

```
class M(type):
    "A metaclass with a class attribute 'a'."
    a = 1

class B:
    "An instance of M with a meta-attribute 'a'."
    __metaclass__ = M

class C(B):
    "An instance of M with the same meta-attribute 'a'"

if __name__ == "__main__":
    print B.a, C.a # => 1 1
    print super(C,C).a #=> attribute error
```

If you run this, you will get an attribute error. This is a case where `super` is doing the *right* thing, since 'a' is *not* inherited from B, but it comes directly from the metaclass, 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` is working correctly, but still it may seems surprising at first. You can find the rationale for this behaviour in my second article with David Mertz; in the case of `__name__` it is obvious though: you don't want all of your objects to have a name, even if all your classes do.

There are certainly other bugs and pitfalls which I have not mentioned here because I think are not worth mention, or because I have forgot them, or also because I am not aware of them all. So, be careful when you use `super`, especially in earlier versions of Python.

Remember to use super consistently

Some years ago James Knight wrote an essay titled Super considered harmful where he points out a few shortcomings of `super` and he makes an important recommendation: *use super consistently, and document that you use it, as it is part of the external interface for your class, like it or not.* The issue is that a developer inheriting from a hierarchy written by somebody else has to know if the hierarchy uses `super` internally or not. For instance, consider this case, where the library author has used `super` internally:

```
# library_using_super

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

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

If the application programmer knows that the library uses `super` internally, she will use `super` and everything will work just fine; but it she does not know if the library uses `super` she may be tempted to call `A.__init__` and `B.__init__` directly, but this will end up in having `B.__init__` called twice!

```
>>> from library_using_super import A, B

>>> class C(A, B):
...     def __init__(self):
...         print "C",
...         A.__init__(self)
...         B.__init__(self)

>>> c = C()
C A B B
```

On the other hand, if the library does *not* uses `super` internally,

```
# library_not_using_super

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

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

the application programmer cannot use `super` either, otherwise `B.__init__` will not be called:

```
>>> from library_not_using_super import A, B

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

>>> c = C()
C A
```

So, if you use classes coming from a library in a multiple inheritance situation, you must know if the classes were intended to be cooperative (using `super`) or not. Library author should always document their usage of `super`.

Argument passing in cooperative methods can fool you

James Knight devolves a paragraph to the discussion of argument passing in cooperative methods. Basically, if you want to be safe, all your cooperative methods should have a compatible signature. There are various ways of getting a compatible signature, for instance you could accept everything (i.e. your cooperative methods could have signature `*args, **kw`) which is a bit too much for me, or all of your methods could have exactly the same arguments. The issue comes when you have default arguments, since your MRO can change if you change your hierarchy, and argument passing may break down. Here is an example:

```
"An example of argument passing in cooperative methods"

class A(object):
    def __init__(self):
        print 'A'

class B(A):
    def __init__(self, a=None):
        print 'B with a=%s' % a
        super(B, self).__init__(a)

class C(A):
    def __init__(self, a):
        print 'C with a=%s' % a
        super(C, self).__init__()

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

```
>>> from cooperation_ex import D
>>> d = D()
D
B with a=None
```

```
C with a=None
A
```

This works, but it is fragile (you see what will happen if you change `D(B, C)` with `D(C, B)`?) and in general it is always difficult to figure out which arguments will be passed to each method and in which order so it is best just to use the same arguments everywhere (or not to use cooperative methods altogether, if you have no need for cooperation). There is no shortage of examples of trickiness in multiple inheritance hierarchies; for instance I remember a post from comp.lang.python about the fragility of super when changing the base class.

Also, beware of situations in which you have some old style classes mixing with new style classes: the result may depend on the order of the base classes (see examples 2-2b and 2-3b in Super considered harmful).

UPDATE: the introduction of Python 2.6 made the special methods `__new__` and `__init__` even more brittle with respect to cooperative super calls.

Starting from Python 2.6 the special methods `__new__` and `__init__` of `object` do not take any argument, whereas previously the had a generic signature, but all the arguments were ignored. That means that it is very easy to get in trouble if your constructors take arguments. Here is an example:

```
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
```

As you see, this cannot work: when `self` is an instance of `C`, `super(A, self).__init__()` will call `B.__init__` without arguments, resulting in a `TypeError`. In older Python you could avoid that by passing `a` to the super calls, since `object.__init__` could be called with any number of arguments. This problem was recently pointed out by Menno Smits in his blog and there is no way to solve it in Python 2.6, unless you change all of your classes to inherit from a custom `Object` class with an `__init__` accepting all kind of arguments, i.e. basically reverting back to the Python 2.5 situation.

Conclusion: is there life beyond super?

In this series I have argued that `super` is tricky; I think nobody can dispute that. However the existence of dark corners is not a compelling argument against a language construct: after all, they are rare and there is an easy solution to their obscurity, i.e. documenting them. This is what I have being doing all along. On the other hand, one may wonder if all `super` warts aren't hints of some serious problem underlying. It may well be that the problem is not with `super`, nor with cooperative methods: the problem may be with multiple inheritance itself.

I personally liked super, cooperative methods and multiple inheritance for a couple of years, then I started working with Zope and my mind changed completely. Zope 2 did not use super at all but is a mess anyway, so the problem is multiple inheritance itself. Inheritance makes your code heavily coupled and difficult to follow (*spaghetti inheritance*). I have not found a real life problem yet that I could not solve with single inheritance + composition/delegation in a better and more maintainable way than using multiple inheritance. Nowadays I am *very* careful when using multiple inheritance.

People should be educated about the issues; moreover people should be aware that there are alternative to multiple inheritance in other languages. For instance Ruby uses mixins (they are a restricted multiple inheritance without cooperative methods and with a well defined superclass, but they do not solve the issue of name conflicts and the issue with the ordering of the mixin classes); recently some people proposed the concepts of traits (restricted mixin where name conflicts must be solved explicitely and the ordering of the mixins does not matter) which is interesting.

In CLOS multiple inheritance works better since (multi-)methods are defined outside classes and `call-next-method` is well integrated in the language; it is simpler to track down the ancestors of a single method than to wonder about the full class hierarchy. The language SML (which nobody except academics use, but would deserve better recognition) goes boldly in the direction of favoring composition over inheritance and uses functors to this aim.

Recently I have written a trilogy of papers for Stacktrace discussing why multiple inheritance and mixins are a bad idea and suggesting alternatives. I plan to translate the series and to publish here in the future. For the moment you can use the Google Translator. The series starts from here and it is a recommended reading if you ever had troubles with mixins.

Super in Python 3

Most languages supporting inheritance support cooperative inheritance, i.e. there is a language-supported way for children methods to dispatch to their parent method: this is usually done via a `super` keyword. Things are easy when the language support single inheritance only, since each class has a single parent and there is an unique concept of super method. Things are difficult when the language support multiple inheritance: in that case the programmer has to understand the intricacies of the Method Resolution Order.

Why cooperative hierarchies are tricky

This paper is intended to be very practical, so let me start with an example. Consider the following hierarchy (in Python 3):

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

class B(object):
    def __init__(self):
```

```
        print('B.__init__')
        super().__init__()

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

What is the "superclass" of `A`? In other words, when I create an instance of `A`, which method will be called by `super().__init__()`? Notice that I am considering here generic instances of `A`, not only direct instances: in particular, an instance of `C` is also an instance of `A` and instantiating `C` will call `super().__init__()` in `A.__init__` at some point: the tricky point is to understand which method will be called for *indirect* instances of `A`.

In a single inheritance language there is an unique answer both for direct and indirect instances (`object` is the super class of `A` and `object.__init__` is the method called by `super().__init__()`). On the other hand, in a multiple inheritance language there is no easy answer. It is better to say that there is no super class and it is impossible to know which method will be called by `super().__init__()` unless the subclass from wich `super` is called is known. In particular, this is what happens when we instantiate `C`:

```
>>> c = C()
C.__init__
A.__init__
B.__init__
```

As you see the super call in `C` dispatches to `A.__init__` and then the super call there dispatches to `B.__init__` which in turns dispatches to `object.__init__`. The important point is that *the same super call can dispatch to different methods*: when `super().__init__()` is called directly by instantiating `A` it dispatches to `object.__init__` whereas when it is called indirectly by instantiating `C` it dispatches to `B.__init__`. If somebody extends the hierarchy, adds subclasses of `A` and instantiated them, then the super call in `A.__init__` can dispatch to an entirely different method: the super method call depends on the instance I am starting from. The precise algorithm specifying the order in which the methods are called is called the Method Resolution Order algorithm, or MRO for short. It is discussed in detail in an old essay I wrote years ago and interested readers are referred to it (see the references below). Here I will take the easy way and I will ask Python.

Given any class, it is possibly to extract its linearization, i.e. the ordered list of its ancestors plus the class itself: the super call follow such list to decide which is the right method to dispatch to. For instance, if you are considering a direct instance of `A`, `object` is the only class the super call can dispatch to:

```
>>> A.mro()
[<class '__main__.A'>, <class 'object'>]
```

If you are considering a direct instance of `C`, `super` looks at the linearization of `C`:

```
>>> C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class
'object'>]
```

A super call in `C` will look first at `A`, then at `B` and finally at `object`. Finding out the linearization is non-trivial; just to give an example suppose we add to our hierarchy three classes `D`, `E` and `F` in this way:

```
>>> class D: pass
>>> class E(A, D): pass
>>> class F(E, C): pass
>>> for c in F.mro():
...     print(c.__name__)
F
E
C
A
D
B
object
```

As you see, for an instance of `F` a super call in `A.__init__` will dispatch at `D.__init__` and not directly at `B.__init__`!

The problem with incompatible signatures

I have just shown that one cannot tell in advance where the supercall will dispatch, unless one knows the whole hierarchy: this is quite different from the single inheritance situation and it is also very much error prone and brittle. When you design a hierarchy you will expect for instance that `A.__init__` will call `B.__init__`, but adding classes (and such classes may be added by a third party) may change the method chain. In this case `A.__init__` (when invoked by an `F` instance) will call `D.__init__`. This is dangerous: for instance, if the behavior of your code depends on the ordering of the methods you may get in trouble. Things are worse if one of the methods in the cooperative chain does not have a compatible signature, since the chain will break.

This problem is not theoretical and it happens even in very trivial hierarchies. For instance, here is an example of incompatible signatures in the `__init__` method (this problem affects even Python 2.6, not only Python 3.X):

```
class X(object):
   def __init__(self, a):
       super().__init__()

class Y(object):
   def __init__(self, a):
       super().__init__()

class Z(X, Y):
   def __init__(self, a):
       super().__init__(a)
```

Here instantiating `X` and `Y` works fine, but as soon as you introduce `Z` you get in trouble since `super().__init__(a)` in `Z.__init__` will call `super().__init__()` in `X` which in turns will call `Y.__init__` with no arguments, resulting in a `TypeError`! In older Python versions (from 2.2 to 2.5) such problem can be avoided by leveraging on the fact that `object.__init__` accepts any number of arguments (ignoring them), by replacing `super().__init__()` with `super().__init__(a)`. In Python 2.6+ instead there is no real solution for this problem, except avoiding `super` in the constructor or avoiding multiple inheritance.

In general if you want to support multiple inheritance you should use `super` only when the methods in a cooperative chain have consistent signature: that means that you will not use super in `__init__` and `__new__` since likely your constructors will have custom arguments whereas `object.__init__` and `object.__new__` have no arguments. However, in practice, you may inherits from third party classes which do not obey this rule, or others could derive from your classes without following this rule and breakage may occur. For instance, I have used `super` for years in my `__init__` methods and I never had problems because in older Python versions `object.__init__` accepted any number of arguments: but in Python 3 all that code is fragile under multiple inheritance. I am left with two choices: removing `super` or telling people that those classes are not intended to be used in multiple inheritance situations, i.e. the constructors will break if they do that. Nowadays I tend to favor the second choice.

Luckily, usually multiple inheritance is used with mixin classes, and mixins do not have constructors, so that in practice the problem is mitigated.

The intended usage for super

Even if `super` has its shortcomings, there are meaningful use cases for it, assuming you think multiple inheritance is a legitimate design technique. For instance, if you use metaclasses and you want to support multiple inheritance, you *must* use `super` in the `__new__` and `__init__` methods: there is no problem, since the constructor for metaclasses has a fixed signature *(name, bases, dictionary)*. But metaclasses are extremely rare, so let me give a more meaningful example for an application programmer where a design bases on cooperative multiple inheritances could be reasonable.

Suppose you have a bunch of `Manager` classes which share many common methods and which are intended to manage different resources, such as databases, FTP sites, etc. To be concrete, suppose there are two common methods: `getinfolist` which returns a list of strings describing the managed resorce (containing infos such as the URI, the tables in the database or the files in the site, etc.) and `close` which closes the resource (the database connection or the FTP connection). You can model the hierarchy with a `Manager` abstract base class

```
class Manager(object):
    def close(self):
        pass
    def getinfolist(self):
        return []
```

and two concrete classes `DbManager` and `FtpManager`:

```
class DbManager(Manager):
    def __init__(self, dsn):
        self.conn = DBConn(dsn)
    def close(self):
        super().close()
        self.conn.close()
    def getinfolist(self):
        return super().getinfolist() + ['db info']

class FtpManager(Manager):
    def __init__(self, url):
        self.ftp = FtpSite(url)
    def close(self):
        super().close()
        self.ftp.close()
    def getinfolist(self):
        return super().getinfolist() + ['ftp info']
```

Now suppose you need to manage both a database and an FTP site: then you can define a `MultiManager` as follows:

```
class MultiManager(DbManager, FtpManager):
    def __init__(self, dsn, url):
        DbManager.__init__(dsn)
        FtpManager.__init__(url)
```

Everything works: calling `MultiManager.close` will in turn call `DbManager.close` and `FtpManager.close`. There is no risk of running in trouble with the signature since the `close` and `getinfolist` methods have all the same signature (actually they take no arguments at all). Notice also that I did not use `super` in the constructor. You see that `super` is *essential* in this design: without it, only `DbManager.close` would be called and your FTP connection would leak. The `getinfolist` method works similarly: forgetting `super` would mean losing some information. An alternative not using `super` would require defining an explicit method `close` in the `MultiManager`, calling `DbManager.close` and `FtpManager.close` explicitly, and an explicit method `getinfolist` calling `'DbManager.getinfolist` and `FtpManager.getinfolist`:

```
def close(self):
    DbManager.close(self)
    FtpManager.close(self)

def getinfolist(self):
    return DbManager.getinfolist(self) + FtpManager.getinfolist(self)
```

This would be less elegant but probably clearer and safer so you can always decide not to use `super` if you really hate it. However, if you have `N` common methods, there is some boiler plate to write; moreover, every time you add a `Manager` class you must add it to the `N` common methods, which is ugly. Here `N` is just 2, so not using `super` may work well, but in general it is clear that the cooperative approach is more effective. Actually, I strongly believe (and always

had) that `super` and the MRO are the *right* way to do multiple inheritance: but I also believe that multiple inheritance itself is *wrong*. For instance, in the `MultiManager` example I would not use multiple inheritance but composition and I would probably use a generalization such as the following:

```
class MyMultiManager(Manager):
    def __init__(self, *managers):
        self.managers = managers
    def close(self):
        for mngr in self.managers:
            mngr.close()
    def getinfolist(self):
        return sum(mngr.getinfolist() for mngr in self.managers)
```

There are languages that do not provide inheritance (even single inheritance!) and are perfectly fine, so you should always question if you should use inheritance or not. There are always many options and the design space is rather large. Personally, I always use `super` but I use single-inheritance only, so that my cooperative hierarchies are trivial.

The magic of super in Python 3

Deep down, `super` in Python 3 is the same as in Python 2.X. However, on the surface - at the syntactic level, not at the semantic level - there is a big difference: Python 3 super is smart enough to figure out *the class it is invoked from and the first argument of the containing method*. Actually it is so smart that it works also for inner classes and even if the first argument is not called `self`. In Python 2.X `super` is dumber and you must tell the class and the argument explicitly: for instance our first example must be written

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

By the way, this syntax works both in Python 3 *and* in Python 2, this is why I said that deep down `super` is the same. The new feature in Python 3 is that there is a shortcut notation `super()` for `super(A, self)`. In Python 3 the (bytecode) compiler is smart enough to recognize that the supercall is performed inside the class `A` so that it inserts the reference to `A` automagically; moreover it inserts the reference to the first argument of the current method too. Typically the first argument of the current method is `self`, but it may be `cls` or any identifier: `super` will work fine in any case.

Since `super()` knows the class it is invoked from and the class of the original caller, it can walk the MRO correctly. Such information is stored in the attributes `.__thisclass__` and `.__self_class__` and you may understand how it works from the following example:

```
class Mother(object):
    def __init__(self):
        sup = super()
```

```
        print(sup.__thisclass__)
        print(sup.__self_class__)
        sup.__init__()

class Child(Mother):
    pass
```

```
>>> child = Child()
<class '__main__.Mother'>
<class '__main__.Child'>
```

Here `.__self__class__` is just the class of the first argument (`self`) but this is not always the case. The exception is the case of classmethods and staticmethods taking a class as first argument, such as `__new__`. Specifically, `super(cls, x)` checks if `x` is an instance of `cls` and then sets `.__self_class__` to `x.__class__`; otherwise (and that happens for classmethods and for `__new__`) it checks if `x` is a subclass of `cls` and then sets `.__self_class__` to `x` directly. For instance, in the following example

```
class C0(object):
    @classmethod
    def c(cls):
        print('called classmethod C0.c')

class C1(C0):
    @classmethod
    def c(cls):
        sup = super()
        print('__thisclass__', sup.__thisclass__)
        print('__selfclass__', sup.__self_class__)
        sup.c()

class C2(C1):
    pass
```

the attribute `.__self_class__` is *not* the class of the first argument (which would be `type` the metaclass of all classes) but simply the first argument:

```
>>> C2.c()
__thisclass__ <class '__main__.C1'>
__selfclass__ <class '__main__.C2'>
called classmethod C0.c
```

There is a lot of magic going on in Python 3 `super`, and even more. For instance, this is a syntax that cannot work:

```
def __init__(self):
    print('calling __init__')
    super().__init__()

class C(object):
    __init__ = __init__

if __name__ == '__main__':
    c = C()
```

If you try to run this code you will get a `SystemError: super(): __class__ cell not found` and the reason is obvious: since the `__init__` method is external to the class the compiler cannot infer to which class it will be attached at runtime. On the other hand, if you are completely explicit and you use the full syntax, by writing the external method as

```
def __init__(self):
    print('calling __init__')
    super(C, self).__init__()
```

everything will work because you are explicitly telling than the method will be attached to the class `C`.

I will close this section by noticing a wart of `super` in Python 3, pointed out by Armin Ronacher and others: the fact that `super` should be a keyword but it is not. Therefore horrors like the following are possible:

```
def super():
    print("I am evil, you are NOT calling the supermethod!")

class C(object):
    def __init__(self):
        super().__init__()

if __name__ == '__main__':
    c = C() # prints "I am evil, you are NOT calling the supermethod!"
```

DON'T DO THAT! Here the called `__init__` is the `__init__` method of the object `None`!!

Of course, only an evil programmer would shadow `super` on purpose, but that may happen accidentally. Consider for instance this use case: you are refactoring an old code base written before the existence of `super` and using `from mod import *` (this is ugly but we know that there are code bases written this way), with `mod` defining a function `super` which has nothing to do with the `super` builtin. If in this code you replace `Base.method(self, *args)` with `super().method(*args)` you will introduce a bug. This is not common (it never happened to me), but still it is bug that could not happen if `super` were a keyword.

Moreover, `super` is special and it will not work if you change its name as in this example:

```
# from http://lucumr.pocoo.org/2010/1/7/pros-and-cons-about-python-3
_super = super
class Foo(Bar):
    def foo(self):
        _super().foo()
```

Here the bytecode compiler will not treat specially `_super`, only `super`. It is unfortunate that we missed the opportunity to make `super` a keyword in Python 3, without good reasons (Python 3 was expected to break compatibility anyway).

References

There is plenty of material about super and multiple inheritance. You should probably start from the MRO paper, then read Super considered harmful by James Knight. A lot of the issues with `super`, especially in old versions of Python are covered in Things to know about super. I did spent some time thinking about ways to avoid multiple inheritance; you may be interested in reading my series Mixins considered harmful.