# Deep Thoughts by Raymond Hettinger

Ruminations on Computers, Programming and Life

## Python's super() considered super!

If you aren't wowed by Python's super() builtin, chances are you don't really know what it is capable of doing or how to use it effectively.

Much has been written about super() and much of that writing has been a failure. This article seeks to improve on the situation by:

- providing practical use cases
- giving a clear mental model of how it works
- showing the tradecraft for getting it to work every time
- concrete advice for building classes that use super()
- favoring real examples over abstract ABCD diamond diagrams.

The examples for this post are available in both Python 2 syntax and Python 3 syntax.

Using Python 3 syntax, let's start with a basic use case, a subclass for extending a method from one of the builtin classes:

```
class LoggingDict(dict):
    def __setitem__(self, key, value):
        logging.info('Setting %r to %r' % (key, value))
        super().__setitem__(key, value)
```

This class has all the same capabilities as its parent, *dict*, but it extends the __setitem__ method to make log entries whenever a key is updated. After making a log entry, the method uses super() to delegate the work for actually updating the dictionary with the key/value pair.

Before super() was introduced, we would have hardwired the call with *dict.__setitem__(self, key, value)*. However, super() is better because it is a computed indirect reference.

One benefit of indirection is that we don't have to specify the delegate class by name. If you edit the source code to switch the base class to some other mapping, the super() reference will automatically follow. You have a single source of truth:

```
class LoggingDict(SomeOtherMapping):            # new base class
    def __setitem__(self, key, value):
        logging.info('Setting %r to %r' % (key, value))
        super().__setitem__(key, value)         # no change needed
```

In addition to isolating changes, there is another major benefit to computed indirection, one that may not be familiar to people coming from static languages. Since the indirection is computed at runtime, we have the freedom to influence the calculation so that the indirection will point to some other class.

The calculation depends on both the class where super is called and on the instance's tree of ancestors. The first component, the class where super is called, is determined by the source code for that class. In our example, super() is called in the *LoggingDict.__setitem__* method. That component is fixed. The second and more interesting component is variable (we can create new subclasses with a rich tree of ancestors).

Let's use this to our advantage to construct a logging ordered dictionary without modifying our existing classes:

```
class LoggingOD(LoggingDict, collections.OrderedDict):
    pass
```

The ancestor tree for our new class is: *LoggingOD*, *LoggingDict*, *OrderedDict*, *dict*, *object*. For our purposes, the important result is that *OrderedDict* was inserted after *LoggingDict* and before *dict*! This means that the super() call in *LoggingDict.__setitem__* now dispatches the key/value update to *OrderedDict* instead of *dict*.

Think about that for a moment. We did not alter the source code for *LoggingDict*. Instead we built a subclass whose only logic is to compose two existing classes and control their search order.

---

**Search Order**

What I've been calling the search order or ancestor tree is officially known as the Method Resolution Order or MRO. It's easy to view the MRO by printing the __mro__ attribute:

```
>>> pprint(LoggingOD.__mro__)
(<class '__main__.LoggingOD'>,
 <class '__main__.LoggingDict'>,
 <class 'collections.OrderedDict'>,
 <class 'dict'>,
 <class 'object'>)
```

If our goal is to create a subclass with an MRO to our liking, we need to know how it is calculated. The basics are simple. The sequence includes the class, its base classes, and the base classes of those bases and so on until reaching *object* which is the root class of all classes. The sequence is ordered so that a class always appears before its parents, and if there are multiple parents, they keep the same order as the tuple of base classes.

The MRO shown above is the one order that follows from those constraints:

- LoggingOD precedes its parents, LoggingDict and OrderedDict
- LoggingDict precedes OrderedDict because LoggingOD.__bases__ is (LoggingDict, OrderedDict)
- LoggingDict precedes its parent which is dict
- OrderedDict precedes its parent which is dict
- dict precedes its parent which is object

The process of solving those constraints is known as linearization. There are a number of good papers on the subject, but to create subclasses with an MRO to our liking, we only need to know the two constraints: children precede their parents and the order of appearance in *__bases__* is respected.

---

**Practical Advice**

super() is in the business of delegating method calls to some class in the instance's ancestor tree. For reorderable method calls to work, the classes need to be designed cooperatively. This presents three easily solved practical issues:

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

1) Let's first look at strategies for getting the caller's arguments to match the signature of the called method. This is a little more challenging than traditional method calls where the callee is known in advance. With super(), the callee is not known at the time a class is written (because a subclass written later may introduce new classes into the MRO).

One approach is to stick with a fixed signature using positional arguments. This works well with methods like __setitem__ which have a fixed signature of two arguments, a key and a value. This technique is shown in the *LoggingDict* example where __setitem__ has the same signature in both *LoggingDict* and *dict*.

A more flexible approach is to have every method in the ancestor tree cooperatively designed to accept keyword arguments and a keyword-arguments dictionary, to remove any arguments that it needs, and to forward the remaining arguments using **kwds, eventually leaving the dictionary empty for the final call in the chain.

Each level strips-off the keyword arguments that it needs so that the final empty dict can be sent to a method that expects no arguments at all (for example, *object.__init__* expects zero arguments):

```
class Shape:
    def __init__(self, shapename, **kwds):
        self.shapename = shapename
        super().__init__(**kwds)

class ColoredShape(Shape):
    def __init__(self, color, **kwds):
        self.color = color
        super().__init__(**kwds)

cs = ColoredShape(color='red', shapename='circle')
```

2) Having looked at strategies for getting the caller/callee argument patterns to match, let's now look at how to make sure the target method exists.

The above example shows the simplest case. We know that *object* has an __init__ method and that *object* is always the last class in the MRO chain, so any sequence of calls to *super().__init__* is guaranteed to end with a call to *object.__init__* method. In other words, we're guaranteed that the target of the super() call is guaranteed to exist and won't fail with an *AttributeError*.

For cases where *object* doesn't have the method of interest (a draw() method for example), we need to write a root class that is guaranteed to be called before *object*. The responsibility of the root class is simply to eat the method call without making a forwarding call using super().

*Root.draw* can also employ defensive programming using an assertion to ensure it isn't masking some other draw() method later in the chain. This could happen if a subclass erroneously incorporates a class that has a draw() method but doesn't inherit from *Root*.:

```
class Root:
    def draw(self):
        # the delegation chain stops here
        assert not hasattr(super(), 'draw')

class Shape(Root):
    def __init__(self, shapename, **kwds):
        self.shapename = shapename
        super().__init__(**kwds)
    def draw(self):
        print('Drawing.  Setting shape to:', self.shapename)
        super().draw()

class ColoredShape(Shape):
    def __init__(self, color, **kwds):
        self.color = color
        super().__init__(**kwds)
    def draw(self):
        print('Drawing.  Setting color to:', self.color)
        super().draw()

cs = ColoredShape(color='blue', shapename='square')
cs.draw()
```

If subclasses want to inject other classes into the MRO, those other classes also need to inherit from *Root* so that no path for calling draw() can reach *object* without having been stopped by *Root.draw*. This should be clearly documented so that someone writing new cooperating classes will know to subclass from *Root*. This restriction is not much different than Python's own requirement that all new exceptions must inherit from *BaseException*.

3) The techniques shown above assure that super() calls a method that is known to exist and that the signature will be correct; however, we're still relying on super() being called at each step so that the chain of delegation continues unbroken. This is easy to achieve if we're designing the classes cooperatively – just add a super() call to every method in the chain.

The three techniques listed above provide the means to design cooperative classes that can be composed or reordered by subclasses.

---

**How to Incorporate a Non-cooperative Class**

Occasionally, a subclass may want to use cooperative multiple inheritance techniques with a third-party class that wasn't designed for it (perhaps its method of interest doesn't use super() or perhaps the class doesn't inherit from the root class). This situation is easily remedied by creating an adapter class that plays by the rules.

For example, the following *Moveable* class does not make super() calls, and it has an __init__() signature that is incompatible with *object.__init__*, and it does not inherit from *Root*:

```
class Moveable:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def draw(self):
        print('Drawing at position:', self.x, self.y)
```

If we want to use this class with our cooperatively designed *ColoredShape* hierarchy, we need to make an adapter with the requisite super() calls:

```
class MoveableAdapter(Root):
    def __init__(self, x, y, **kwds):
        self.movable = Moveable(x, y)
        super().__init__(**kwds)
    def draw(self):
        self.movable.draw()
        super().draw()

class MovableColoredShape(ColoredShape, MoveableAdapter):
    pass

MovableColoredShape(color='red', shapename='triangle',
                    x=10, y=20).draw()
```

---

**Complete Example – Just for Fun**

In Python 2.7 and 3.2, the collections module has both a *Counter* class and an *OrderedDict* class. Those classes are easily composed to make an *OrderedCounter*:

```
from collections import Counter, OrderedDict

class OrderedCounter(Counter, OrderedDict):
     'Counter that remembers the order elements are first seen'
     def __repr__(self):
         return '%s(%r)' % (self.__class__.__name__,
                            OrderedDict(self))
     def __reduce__(self):
         return self.__class__, (OrderedDict(self),)

oc = OrderedCounter('abracadabra')
```

---

**Notes and References**

* When subclassing a builtin such as dict(), it is often necessary to override or extend multiple methods at a time. In the above examples, the __setitem__ extension isn't used by other methods such as *dict.update*, so it may be necessary to extend those also. This requirement isn't unique to super(); rather, it arises whenever builtins are subclassed.

* If a class relies on one parent class preceding another (for example, *LoggingOD* depends on *LoggingDict* coming before *OrderedDict* which comes before *dict*), it is easy to add assertions to validate and document the intended method resolution order:

```
position = LoggingOD.__mro__.index
assert position(LoggingDict) < position(OrderedDict)
assert position(OrderedDict) < position(dict)
```

* Good write-ups for linearization algorithms can be found at Python MRO documentation and at Wikipedia entry for C3 Linearization.

* The Dylan programming language has a *next-method* construct that works like Python's super(). See Dylan's class docs for a brief write-up of how it behaves.

* The Python 3 version of super() is used in this post. The full working source code can be found at: Recipe 577720. The Python 2 syntax differs in that the *type* and *object* arguments to super() are explicit rather than implicit. Also, the Python 2 version of super() only works with new-style classes (those that explicitly inherit from *object* or other builtin type). The full working source code using Python 2 syntax is at Recipe 577721.

---

**Acknowledgements**

Serveral Pythonistas did a pre-publication review of this article. Their comments helped improve it quite a bit.

They are: Laura Creighton, Alex Gaynor, Philip Jenvey, Brian Curtin, David Beazley, Chris Angelico, Jim Baker, Ethan Furman, and Michael Foord. Thanks one and all.

**Explore posts in the same categories:** Algorithms, Documentation, Inheritance, Open Source, Python
This entry was posted on May 26, 2011 at 9:15 am and is filed under Algorithms, Documentation, Inheritance, Open Source, Python. You can subscribe via RSS 2.0 feed to this post's comments. You can comment below, or link to this permanent URL from your own site.

## 38 Comments on "Python's super() considered super!"

**Matías Says:**

May 26, 2011 at 11:27 am
This post rocks!. I was aware of super() but this information opens my eyes to a new world of posibilities. Thank you.

**Reply**

**afranck64 Says:**

January 10, 2012 at 7:42 am
I didn't know about this, i was always using the "Parent.__init__" method (Py2.X). I suppose there are many oder new features in Py3

**Reply**

**Bob Says:**

May 26, 2011 at 12:29 pm
This is one of the two annoyances with Python (the other is the GIL).

C++ just does this *so* much better.

**Reply**

**James Says:**

November 14, 2011 at 9:11 am
Surely, in C++, you have to explicitly state the superclass in ambiguous function calls? You can still do that in Python if you want. I really can't imagine how anyone could think that the way C++ deals with multiple inheritance is superior to the way Python deals with it.

**Reply**

**Pykler Says:**

November 17, 2011 at 8:49 pm
Explicit is better than implicit. I mostly always avoid super. You do show cases where it can be useful though.

**iguananaut Says:**

January 11, 2013 at 8:56 am
Re: "Explicit is better than implicit"–Although the semantics of calling super() in Python 3 are not explicit about the superclass an instance object it's still at least well-defined and consistent. Likewise for the MRO. So as long as you understand the rules (which are not really all that complex at the end of the day) it's at least predictable.

**Chris Torek Says:**

May 26, 2011 at 1:04 pm
The fact that Python 2.x requires the class-name (and of course "self") as an argument to super() is problematic, wiping out your first "benefit of indirection". This was always my pet peeve with super(). I know it seems minor, but I always found it tipped the balance between "always use super" and "don't bother with super if you don't need it" over towards "don't bother".

Fortunately, "fixed in Python 3"….

**Reply**

**rhettinger Says:**

May 26, 2011 at 4:48 pm
No one really liked the Python 2 syntax though it did have the advantage of being explicit about the two inputs to the computation (the mro of self and the current position in the mro).

The first advantage listed in the post still applies though. It is only the current class that is referenced explicitly. The parent class is still reached through indirection, so you can change the bases and the super() calls will follow automatically.

Thank you the reply and your insight.

Raymond

**Reply**

**Max Says:**

February 27, 2013 at 5:51 am
But when you change you class name, you must to change this name again in all super's in this class.

**Max Says:**

February 27, 2013 at 6:00 am
At bottom I found my mistake:
super(self.__class__, self)
We dont need to write class name more then once.

**Blue Havana Says:**

May 26, 2011 at 5:41 pm
For Python 2, instead of using an explicit class, you can use 'super(type(self), self)'. Makes the code more maintainable if you change the name of your class.

**Reply**

**Gavin Panella Says:**

May 27, 2011 at 1:30 am
Unfortunately that doesn't work:

```
>>> class A(object):
… def __init__(self):
… super(type(self), self).__init__()

>>> class B(A):
… def __init__(self):
… super(type(self), self).__init__()

>>> a = A()
>>> b = B() # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
…
RuntimeError: maximum recursion depth exceeded while calling a
Python object
```

When calling A.__init__ from B.__init__, type(self) is B.

**Reply**

**Ned Batchelder Says:**

May 27, 2011 at 7:23 am
@Blue Havana: this is a common "too-tricky" error. If type(self) would have worked there, the creators of super() would have had it take only self, and they would have used type(self) internally.

The class name is needed because self may actually be an instance of a subclass of the class defining the method calling super. super needs to know where in the inheritance tree it is being called from, and that can only be provided by explicitly passing a class.

**Reply**

**Ivan Savov Says:**

May 26, 2011 at 6:10 pm
Very good post!
Thanks for the write up!

I will need to super() in some code quite soon, and now I feel much comfortable getting to work on my re-factoring…

**Reply**

**Benjamin Says:**

May 26, 2011 at 8:14 pm
Nice post! I still wince everytime I think about the horrible compiler hacks that make super() in Python 3 work, though.

**Reply**

**Chris T. Says:**

May 31, 2011 at 9:00 pm
It's not much of a hack: you need only the name of the parent class (so that super() can figure out where in the MRO it is now) and of course the "self" argument (to find the __mro__ list in the first place).

Still, I probably would have been happier overall with an actual keyword, if I had been designing the language in the first place. But it's fine as is. I just hate repeating the name of the class in Python 2.x, since that is another place to make typos or miss a name change when refactoring code.

**merwok Says:**

June 4, 2011 at 8:28 am
Heh, you too 🙂

Nice post Raymond (and coolest title). I was amazed to see that OrderedCounter didn't need to define __init__ at all.

**ezio Says:**

June 1, 2011 at 12:45 am
Thanks.Pretty good!

**Reply**

**Helen Sam Says:**

June 5, 2011 at 6:11 pm
Nice example. Coming from Java programming, I can see the little differences. Will explore Python's super() more.

Thanks for the nice post

**Reply**

**Matt Langston Says:**

June 14, 2011 at 11:47 am
Thank you for taking the time to write this post. It was very helpful!

**Reply**
**super(self, __class__, self) # end of the line for subclassing – yergler.net Says:**

July 4, 2011 at 8:44 pm
[…] reading: Raymond Hettinger's excellent blog post on super provides a great overview of super and shows off the improved Python 3 syntax, which removes the […]

**Reply**
**Waldi Syafei Says:**

September 9, 2011 at 10:22 am
I've been searching for this
Very good post..

**Reply**
**Learn core Python from Stackoverflow | Glorified Geek Says:**

October 2, 2011 at 11:24 am
[…] short yet useful answer explaining what good is super() for. You should also read the article Pythons's super() considered super! along with […]

**Reply**
**Dilawar Says:**

December 12, 2011 at 1:13 pm
Great information. Much better than the standard documentation.

**Reply**
**Mike Gagnon Says:**

December 14, 2011 at 8:49 am
You should have opened with "If you aren't a super() user, chances are…"

**Reply**
**Kov Says:**

January 11, 2012 at 11:09 pm
super is super, this post is even more super..

**Reply**
**Pedle Zelnip Says:**

March 5, 2012 at 2:47 pm
"and every occurrence of the method needs to use super()"

This is the fundamental problem with super() and the mro in Python. Lets say I want to create a class which inherits from a class I've written, and another class from a 3rd party library. Now I'm left with a choice, do I go the super() route for calling parent class's __init__'s (hoping and praying that the 3rd party does the same), or do I call directly the SomeClassName.__init__(self) style (in which case if the 3rd party does the super() route, then my code equally breaks).

It's a fundamentally broken solution.

**Reply**

**rhettinger Says:**

March 5, 2012 at 3:27 pm

The article shows how to create a wrapper for third-party classes that were not designed for cooperative multiple inheritance.

Calling it a "fundamentally broken solution" is a red-herring. The solution is based on a good deal of academic research (plus real-world experience with the Dylan programming language). The underlying problem is non-trivial — care and forethought are required for freely composeable classes using multiple inheritance while allowing for diamond patterns.

There is a reason that the style is called cooperative multiple inheritance. The classes either need to be **designed** cooperatively or they need to be wrapped to make them cooperative. Anything else equates to wishful thinking — "Oh, I wish that unrelated third-party classes composed together effortlessly and magically happened to do exactly the behavior I want without me every bothering to specify (or think about) what that behavior should be or how it would work."

**Reply**

**Pedle Zelnip Says:**

March 6, 2012 at 9:12 am

My apologies, I missed the How to Incorporate a Non-cooperative Class section on my 1st read somehow. 🙂

"Oh, I wish that unrelated third-party classes composed together effortlessly and magically happened to do exactly the behavior I want without me every bothering to specify (or think about) what that behavior should be or how it would work."

That's a strawman right there. All I'm saying is that there should be a consistent, safe mechanism for calling parent class constructor's. super() isn't it, as it requires knowledge of how a class was defined in order to know how to call it's constructor.

**Reply**

**Ivc Says:**

May 19, 2012 at 7:34 pm

There's no way around having to know the signature of a parent class' constructor in order to call the parent class' constructor (aside from the technique mentioned in the post of accepting arbitrary kwargs, taking out the ones you can deal with, and passing the rest along). This problem isn't unique to Python – Java and C++ convention have you always declare a constructor that can be called with no arguments.

**Reply**

**Snooze Says:**

March 11, 2012 at 3:47 pm

Thank you for the great post.

I get how python handles multiple class inheritance now, but I still don't quite understand what you meant by this paragraph:

"Root.draw can also employ defensive programming using an assertion to ensure it isn't masking some other draw() method later in the chain. This could happen if a subclass erroneously incorporates a class that has a draw() method but doesn't inherit from Root."

I can understand how the assertion would fail if Root inherited from another class, but we know that Root doesn't because we wrote it ourselves.

I don't quite understand how the assertion in Root will be called if the subclass in question doesn't inherit from Root. Maybe you can show how such an inheritance tree would be structured?

**Reply**
**Karl Knechtel Says:**

March 30, 2012 at 5:57 pm
Regarding the (Counter, OrderedDict) example: when the OrderedCounter is initialized or updated, the Counter __init__ (or update) is run because that's the next class in the MRO, right? So how does the OrderedDict behaviour get triggered? Is Counter making a super() call somewhere that gets routed through OrderedDict (instead of passing directly to the builtin dict type as it normally would)? Or just what?

**Reply**
**rhettinger Says:**

September 5, 2012 at 11:26 pm
Yes, you're correct. The Counter.__init__ method will be the first __init__ encountered in the MRO. The code in Counter.__init__ calls super which finds OrderedDict.__init__ as the next in the MRO.

**Reply**
**Chris Says:**

June 24, 2013 at 10:09 pm
Why does Counter.__init__() call super(), but OrderedDict.__init__() does not?

**rhettinger Says:**

July 19, 2013 at 10:08 pm
*OrderedDict* doesn't call *super* to keep that part of the API closed-off from subclassers. That will give us a little more freedom to replace the current implementation with a faster version written in C.

**Ramesh Sahu Says:**

August 8, 2012 at 4:08 am
Thank you very much for taking the time to write this post. It is very informative..

**Reply**
**nielshenrikbruun Says:**

September 1, 2012 at 9:50 pm
Hi
I\'ve started using classes and subclasses etc for some time now.
And in lack of documentation I\'d do some trail and error when starting.

In my code I usually reuse methods in a class to define new methods, ie I define some base methods that might be used to create more complicated functionality in new methods.

In doing so I\'ve found that using self much like super() is used (I wasn\'t aware of super()) did the job for me.

class Example:
def basemethod1(self):
pass
def basemethod2(self):
pass
def advancedmethod1(self):
refers to self.basemethod1() and/or self.basemethod1()

And it also works when subclassing.

The question is whether the above is proper code or not?
Or if I should use super instead?
what are the pros and cons in self vs super()?

Looking forward to your answer

**Reply**
**Python's super() | blog.notmet.net Says:**

October 5, 2012 at 7:17 am
[…] when you're using Python for object oriented code and have some inheritance going on. This blog post describes the best way I've seen it used so […]

**Reply**