Python's Super is nifty, but you can't use it

(Previously: Python's Super Considered Harmful)

When I first heard about 'super' in python, I thought it sounded like a great idea, and should be adopted by everyone immediately! After all, it fixes inheritance diamonds, makes you not have to encode the superclass name into every call, and lets you use the same structure to call superclass methods as methods on another class (`x.foo(1)` vs. `SuperClass.foo(self, 1)`) However, after poking around with it a bit, I now feel that people should generally avoid it. It really is the "right thing", but it just doesn't seem to work in Python.

One big problem with 'super' is that it sounds like it will cause the superclass's copy of the method to be called. This is simply not the case, it causes the next method in the MRO to be called.

That misconception causes people to make two common mistakes.

1. People omit calls to super(...).__init__ if the only superclass is 'object', as, after all, object.__init__ doesn't do anything! However, this is very incorrect. Doing so will cause other classes' __init__ methods to not be called.
2. People think they know what arguments their method will get, and what arguments they should pass along to super. This is also incorrect.

In addition to the above problems, there is a huge backwards-compatibility problem in that using super() does not mesh well at all with calling the superclass's implementation directly, which has been the proper thing to do thus far.

It just so happens that the semantics of Python's super() are essentially identical to Dylan's 'next-method', which works rather well. So why doesn't Python's implementation work well in practice? There are unfortunately three major differences between Python and Dylan that make Python work significantly less well:

- The *only* reasonable way to call the next method in Dylan is to use 'next-method'. You could not reasonably call a specific implementation explicitly. Thus, there is no problem of some people using next-method and others not.
- Dylan has generic functions, not classes. All the functions for a generic are required to have compatible signatures, while functions on classes are not. As a approximation, a function has a "compatible signature" if

it takes the same number of arguments and the same keyword arguments as the generic it is defined on. (See http://www.gwydiondylan.org/books/dpg/db_179.html#marker-9-529 for the actual rules)
- Dylan's version of __init__ and __new__ only allow keyword arguments, and the generic function accepts all keyword arguments. This considerably reduces the problem with incompatible method signatures.

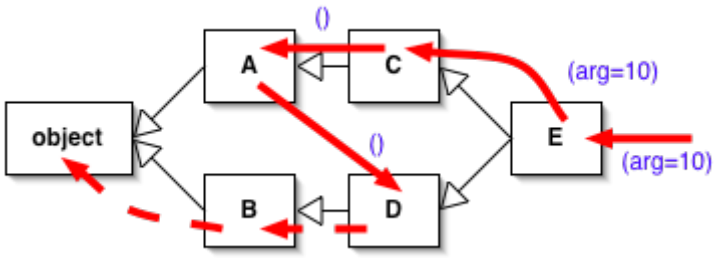
Here are some situations where a non-obvious thing happens in python. Note that the __init__ method is not special -- the same thing happens with any method, I just use __init__ because it is the method that most often needs to be overridden in many classes in the hierarchy.

Argument passing, argh!

Okay, so you have a class hierarchy like in Example 1-1 and you want to convert it to use super. The obvious way to do it, as shown in Example 1-2 does not work. It dies with:

```
MRO: ['E', 'C', 'A', 'D', 'B', 'object']
E arg= 10
C arg= 10
A
Traceback (most recent call last):
  File "example1-2.py", line 27, in ?
    E(10)
  File "example1-2.py", line 24, in __init__
    super(E, self).__init__(arg)
  File "example1-2.py", line 14, in __init__
    super(C, self).__init__()
  File "example1-2.py", line 4, in __init__
    super(A, self).__init__()
TypeError: __init__() takes exactly 2 arguments (1 given)
```

What is it talking about, how can this be? Well, it's pretty simple really -- the next __init__ method after A's in the MRO is D's. D.__init__ takes an argument, which A.__init__ is not passing! Uh oh.



So, here you see the confusion about super again—you must remember that super *does not* call your superclass. You must be prepared to call any other class's method in the hierarchy and be prepared to be called from any other

class's method.

Thus, the general rule is: always pass all arguments you received on to the super function, and, if classes can take differing arguments, always accept *args and **kwargs. This is shown in Example 1-3.

Just as a side note, I thought I'd mention that in Dylan, just "next-method" by itself with no argument-list does the same thing as Python's "super(MyClass, self).currentmethod(alltheargsideclared, *args, **kwargs)". Much simpler, much harder to mess up.

Mixing super and non-super using methods

Subclasses must use super if their superclasses do

A class 'C' derives from two classes ('A' and 'B') that use super to call their super's __init__. However, the author of the subclass does not know about super and simply calls A.__init__ followed by B.__init__.

Example 2-1

Output:

```
MRO is:  ['C', 'A', 'B', 'object']
Methods called: C A B B
```

B.__init__ gets called twice! D'oh! For correct behavior, if any class uses super, all its subclasses must also.

Superclasses must use super if their subclasses do (sometimes)

The inverse is also true. In this example, I demonstrate that inheriting from a class that does not use super causes problems. In particular, any classes after it in the superclass list do not have their methods called.

Example 2-2

Output:

```
MRO: ['F1', 'A', 'B', 'N', 'object']
Calls: F1 A B N
MRO: ['F2', 'A', 'N', 'B', 'object']
Calls: F2 A N
MRO: ['F3', 'N', 'A', 'B', 'object']
Calls: F3 N
MRO: ['F4', 'A', 'B', 'N', 'M', 'object']
Calls: F3 A B N
```

So, F1 appears to work, but F2 and F3 (where N is not the last element), do not, as A and/or B's __init__ does not get called. Thus it would appear that

you simply need to put the classes that do not use super last. This is not the case, as F4 shows.

The rule, then, seems simple: Always put those superclasses that don't use super last, and, if there's more than one, explicitly call all but the first. However, this does not actually work as shown by the following example.

Example 3

Output:

```
MRO: ['G', 'F1', 'A', 'N', 'F2', 'B', 'M', 'object']
Calls: G G1 A N
```

Both F1 and F2 followed the above rule, but, if someone subclasses them, it is impossible for all the non-super-using methods be at the end of the MRO. Thus, breakage.

Thus, in conclusion, if a class uses super, all its superclasses must also.

...But it's okay if they're oldstyle classes

Let's take example 2-2 and 2-3 again, but modify them slightly so that N and M are oldstyle classes (and thus do not derive from object). This turns out to work fine, as long as the oldstyle classes (that do not use super) appear after all newstyle classes in the superclass list of each class. Because they will then always appear after 'object' in the MRO, super will never call them, and you can call them explicitly without worry.

Example 2-2b

Output:

```
MRO: ['F1', 'A', 'B', 'object', 'N']
Calls: F1 A B
MRO: ['F2', 'A', 'N', 'B', 'object']
Calls: F2 A N
MRO: ['F3', 'N', 'A', 'B', 'object']
Calls: F3 N
MRO: ['F4', 'A', 'B', 'object', 'N', 'M']
Calls: F3 A B
```

Example 2-3b

Output:

```
MRO: ['G', 'F1', 'A', 'F2', 'B', 'object', 'N', 'M']
Calls: G G1 A G2 B
```

Conclusion

If you *do* use super, here are some best practices:

- Use it **consistently**, and document that you use it, as it **is** part of the

external interface for your class, like it or not.
- Never call super with anything but the exact arguments you received, unless you **really** know what you're doing.
- When you use it on methods whose acceptable arguments can be altered on a subclass via addition of more optional arguments, always accept *args, **kw, and call super like "`super(MyClass, self).currentmethod(alltheargsideclared, *args, **kwargs)`". If you don't do this, forbid addition of optional arguments in subclasses.
- Never use positional arguments in __init__ or __new__. Always use keyword args, and always call them as keywords, and always pass all keywords on to super.

From my experience in python code, needing to define the same name method in two branches of an inheritance tree is *exceedingly* rare, with the exclusion of __init__. And, super() is useless for that if you need any kind of interoperability with existing python code. If someone has some example code where they actually need this functionality, I'd like to see it.

The super() function is far from easy to use. If you don't believe me: search google for something like "super self py". The first result is the code for PyCon 2004 "New-Style Class Tutorial" by Thomas Wouters. Now, go down to the bottom, where it says (removing irrelevant code):

```
class tristate(int):
    nstates = 3

    # __new__ is the method to override for immutable values.
    # It is a static method. The first argument is the class that is being
    # created (which may be a subclass), and there is no actual instance
    # yet.
    def __new__(cls, state=0):
        # We want to limit the actual value of the type to [0, nstates >
        state %= cls.nstates

        # To actually create the new instance, we call the parent's __new__.
        # Since __new__ is a static method, we have to pass the first
        # argument explicitly.

        # To call the right parent method, we use super(). What super()
        # does, is call the method that _would have been called_ if this
        # method wasn't present. In single-inheritance situations this just
        # means 'int.__new__' in our case, but if a subclass of our class
        # actually inherits from more than just us, it may end up being
        # another method altogether. Using super() in all places assures
        # consistent method-call order.

        # super() can be used in several ways, and the actual return value
        # is a magic object, a proxy object that does the right thing with
        # bound and unbound instance methods, and class methods.
        # Since we only have a class, we

        return super(tristate, cls).__new__(cls, state)
```

Notice two things: 1) it explains in detail how super is supposed to be

useful, and 2) This __new__ only takes a state argument, thus making it difficult to actually use this class in a multiple inheritance situation. However, I must admit, this is no fault of Mr. Wouters. int.__new__ *itself* fails to take extra arguments, and to use keywords for its args, both of which would be required.

```
class NamedInt(int):
    def __new__(cls, num=0, names=None):
        obj=super(NamedInt, cls).__new__(cls, num)
        obj.name=names[obj]
        return obj
    def __str__(self):
       return self.name
    def __repr__(self):
       return self.name

class NamedTristate(tristate, NamedInt):
    pass

NamedTristate(2, "abc")
```

D'oh. `TypeError: __new__() takes at most 2 arguments (3 given)`

I highly suspect nearly all python code that uses super runs into one of the issues documented here.The only situation in which super() can actually be *helpful* is when you have diamond inheritance. And even then, it is often not as helpful as you might have thought. Additionally, it is **harmful** in the situations shown above.

[[[Todo]]]

- Give some examples of why super *really is* necessary sometimes, because all multiple-inheritance trees in python have a diamond structure with 'object' at the base. E.g. overriding __getattribute__. Also that case doesn't have the argument passing problem, because __getattribute__ will always take the same args.
- Make more pretty pictures
- Fontify code and insert examples inline
- Comments welcome, email me.

James Knight --- *foom@fuhm.net*