Understanding Python super() with __init__() methods [duplicate]

Asked 13 years, 9 months ago Modified 8 months ago Viewed 2.4m times

3091

This question already has answers here:

What does 'super' do in Python? - difference between super().__init__() and explicit superclass __init__() (11 answers)

Closed 7 years ago.

Why is `super()` used?

Is there a difference between using `Base.__init__` and `super().__init__` ?

```
class Base(object):
    def __init__(self):
        print "Base created"

class ChildA(Base):
    def __init__(self):
        Base.__init__(self)

class ChildB(Base):
    def __init__(self):
        super(ChildB, self).__init__()

ChildA()
ChildB()
```

python class oop inheritance super

Share Edit Follow

edited Apr 1 at 11:47
Mateen Ulhaq
22.7k 16 88 128

asked Feb 23, 2009 at 0:30
Mizipzor
49.9k 22 94 138

19 this is a very simple intro to classes worth going through: realpython.com/python-super/…. It's easier to digest than the answers given that are for most of us I assume too detailed in the implementation of python. It also has examples to make it concrete. – Charlie Parker Nov 5, 2021 at 18:07

I still don't get it. I want to define a `class Event(tuple)` which creates tuples (timestamp, description) and where the timestamp should default to the current time. Thus, something like `e = Event(description="stored the current time")` should give an instance of the subclass `Event` of the tuple `(1653520485,"stored...")`. But in `__init__()` I cannot modify `self` as I would do for a subclass of `dict`. So I thought I could use `super().__init__` to set the components of the tuple `self`. Can I ? – Max May 25 at 23:16

7 Answers

Sorted by: Highest score (default)

2236

`super()` lets you avoid referring to the base class explicitly, which can be nice. But the main advantage comes with multiple inheritance, where all sorts of fun stuff can happen. See the standard docs on super if you haven't already.

Note that the syntax changed in Python 3.0: you can just say `super().__init__()` instead of `super(ChildB, self).__init__()` which IMO is quite a bit nicer. The standard docs also refer to a guide to using `super()` which is quite explanatory.

Share Edit Follow

edited Nov 21, 2019 at 16:04
Neuron
4,730 4 35 54

answered Feb 23, 2009 at 0:37
Kiv
31.1k 6 42 59

11 @rimiro The syntax of super() is `super([type [, object]])` This will return the superclass of `type`. So in this case the superclass of `ChildB` will be returned. If the second argument is omitted, the super object returned is unbound. If the second argument is an object, then `isinstance(object, type)` must be true. – Omnik Aug 30, 2018 at 15:48

9 If you are here and still confused, please read the answer by Aaron Hall you will leave this page much happier: stackoverflow.com/a/27134600/1886357 – eric Feb 18, 2019 at 23:53

9 can you actually explain what the code does? I don't want to click to 1 million more places to find the answer to this. – Charlie Parker Jun 4, 2019 at 23:00

1149

I'm trying to understand `super()`

The reason we use `super` is so that child classes that may be using cooperative multiple inheritance will call the correct next parent class function in the Method Resolution Order (MRO).

In Python 3, we can call it like this:

```
class ChildB(Base):
    def __init__(self):
        super().__init__()
```

In Python 2, we were required to call `super` like this with the defining class's name and `self`, but we'll avoid this from now on because it's redundant, slower (due to the name lookups), and more verbose (so update your Python if you haven't already!):

```
        super(ChildB, self).__init__()
```

Without super, you are limited in your ability to use multiple inheritance because you hard-wire the next parent's call:

```
        Base.__init__(self) # Avoid this.
```

I further explain below.

"What difference is there actually in this code?:"

```
class ChildA(Base):
    def __init__(self):
        Base.__init__(self)

class ChildB(Base):
    def __init__(self):
        super().__init__()
```

The primary difference in this code is that in `ChildB` you get a layer of indirection in the `__init__` with `super`, which uses the class in which it is defined to determine the next class's `__init__` to look up in the MRO.

I illustrate this difference in an answer at the canonical question, How to use 'super' in Python?, which demonstrates **dependency injection** and **cooperative multiple inheritance**.

If Python didn't have `super`

Here's code that's actually closely equivalent to `super` (how it's implemented in C, minus some checking and fallback behavior, and translated to Python):

```
class ChildB(Base):
    def __init__(self):
        mro = type(self).mro()
        check_next = mro.index(ChildB) + 1 # next after *this* class.
        while check_next < len(mro):
            next_class = mro[check_next]
            if '__init__' in next_class.__dict__:
                next_class.__init__(self)
                break
            check_next += 1
```

Written a little more like native Python:

```
class ChildB(Base):
    def __init__(self):
        mro = type(self).mro()
        for next_class in mro[mro.index(ChildB) + 1:]: # slice to end
            if hasattr(next_class, '__init__'):
                next_class.__init__(self)
                break
```

If we didn't have the `super` object, we'd have to write this manual code everywhere (or recreate it!) to ensure that we call the proper next method in the Method Resolution Order!

How does super do this in Python 3 without being told explicitly which class and instance from the method it was called from?

It gets the calling stack frame, and finds the class (implicitly stored as a local free variable, `__class__`, making the calling function a closure over the class) and the first argument to that function, which should be the instance or class that informs it which Method Resolution Order (MRO) to use.

Since it requires that first argument for the MRO, [using `super` with static methods is impossible as they do not have access to the MRO of the class from which they are called](#).

Criticisms of other answers:

> super() lets you avoid referring to the base class explicitly, which can be nice. . But the main advantage comes with multiple inheritance, where all sorts of fun stuff can happen. See the standard docs on super if you haven't already.

It's rather hand-wavey and doesn't tell us much, but the point of `super` is not to avoid writing the parent class. The point is to ensure that the next method in line in the method resolution order (MRO) is called. This becomes important in multiple inheritance.

I'll explain here.

```
class Base(object):
    def __init__(self):
        print("Base init'ed")

class ChildA(Base):
    def __init__(self):
        print("ChildA init'ed")
        Base.__init__(self)
```

```
class ChildB(Base):
    def __init__(self):
        print("ChildB init'ed")
        super().__init__()
```

And let's create a dependency that we want to be called after the Child:

```
class UserDependency(Base):
    def __init__(self):
        print("UserDependency init'ed")
        super().__init__()
```

Now remember, `ChildB` uses super, `ChildA` does not:

```
class UserA(ChildA, UserDependency):
    def __init__(self):
        print("UserA init'ed")
        super().__init__()

class UserB(ChildB, UserDependency):
    def __init__(self):
        print("UserB init'ed")
        super().__init__()
```

And `UserA` does not call the UserDependency method:

```
>>> UserA()
UserA init'ed
ChildA init'ed
Base init'ed
<__main__.UserA object at 0x0000000003403BA8>
```

But `UserB` does in-fact call UserDependency because `ChildB` invokes `super` :

```
>>> UserB()
UserB init'ed
ChildB init'ed
UserDependency init'ed
Base init'ed
<__main__.UserB object at 0x0000000003403438>
```

Criticism for another answer

In no circumstance should you do the following, which another answer suggests, as you'll definitely get errors when you subclass ChildB:

```
super(self.__class__, self).__init__()  # DON'T DO THIS! EVER.
```

(That answer is not clever or particularly interesting, but in spite of direct criticism in the comments and over 17 downvotes, the answerer persisted in suggesting it until a kind editor fixed his problem.)

Explanation: Using `self.__class__` as a substitute for the class name in `super()` will lead to recursion. `super` lets us look up the next parent in the MRO (see the first section of this answer) for child classes. If you tell `super` we're in the child instance's method, it will then lookup the next method in line (probably this one) resulting in recursion, probably causing a logical failure (in the answerer's example, it does) or a `RuntimeError` when the recursion depth is exceeded.

```
>>> class Polygon(object):
...     def __init__(self, id):
...         self.id = id
...
>>> class Rectangle(Polygon):
...     def __init__(self, id, width, height):
...         super(self.__class__, self).__init__(id)
...         self.shape = (width, height)
...
>>> class Square(Rectangle):
...     pass
...
>>> Square('a', 10, 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __init__
TypeError: __init__() missing 2 required positional arguments: 'width' and
'height'
```

Python 3's new `super()` calling method with no arguments fortunately allows us to sidestep this issue.

Share Edit Follow

edited Jul 1, 2021 at 18:07

answered Nov 25, 2014 at 19:00

Russia Must Remove Putin ♦
356k 85 395 329

57 I'll still need to work my head around this `super()` function, however, this answer is clearly the best in terms of depth and details. I also appreciate greatly the criticisms inside the answer. It also help to better understand the concept by identifying pitfalls in other answers. Thank you ! – Yohan Obadia May 29, 2017 at 15:09

4 I have been using `tk.Tk.__init__(self)` over `super().__init__()` as I didn't fully understand what super was but this post has been very enlightening. I guess in the case of Tkinter classes `tk.Tk.__init__(self)` and `super().__init__()` are the same thing but it looks like you are saying we should avoid doing something like `Base.__init__(self)` so I may be switching to `super()` even though I am still trying to grasp its complexity. – Mike - SMT Nov 8, 2018 at 15:55

3 this answer is especially comprehensive, and really filled in gaps in my knowledge. hats off to you sir. – physincubus Feb 10, 2019 at 20:26

Your second "native Python" example behaves differently than the first example (and as `super` -based solution). `hasattr(next_class, '__init__')` used may implicitly indicate the presence of an unnecessary method. For example, imagine canonical diamond hierarchy `class A(): ...; class B(A): ...; class C(A): ...; class D(B, C): ...` where `A` and `C` have their own `__init__` methods and `D` has `__init__` such as in the described example. Then calling `D().__init__()` will cause `A.__init__` to call instead of `C.__init__` – facehugger Feb 10, 2020 at 21:33

Maybe `if '__init__' in next_class.__dict__:` is better – facehugger Feb 10, 2020 at 21:34

275

It's been noted that in Python 3.0+ you can use

```
super().__init__()
```

to make your call, which is concise and does not require you to reference the parent OR class names explicitly, which can be handy. I just want to add that for Python 2.7 or under, some people implement a name-insensitive behaviour by writing `self.__class__` instead of the class name, i.e.

```
super(self.__class__, self).__init__()  # DON'T DO THIS!
```

HOWEVER, this breaks calls to `super` for any classes that inherit from your class, where `self.__class__` could return a child class. For example:

```
class Polygon(object):
    def __init__(self, id):
        self.id = id
```

```
class Rectangle(Polygon):
    def __init__(self, id, width, height):
        super(self.__class__, self).__init__(id)
        self.shape = (width, height)

class Square(Rectangle):
    pass
```

Here I have a class `Square`, which is a sub-class of `Rectangle`. Say I don't want to write a separate constructor for `Square` because the constructor for `Rectangle` is good enough, but for whatever reason I want to implement a Square so I can reimplement some other method.

When I create a `Square` using `mSquare = Square('a', 10,10)`, Python calls the constructor for `Rectangle` because I haven't given `Square` its own constructor. However, in the constructor for `Rectangle`, the call `super(self.__class__,self)` is going to return the superclass of `mSquare`, so it calls the constructor for `Rectangle` again. This is how the infinite loop happens, as was mentioned by @S_C. In this case, when I run `super(...).__init__()` I am calling the constructor for `Rectangle` but since I give it no arguments, I will get an error.

Share Edit Follow

edited May 12, 2020 at 14:37

answered Oct 8, 2013 at 20:08
AnjoMan
4,748 4 19 28

45 What this answer suggests, `super(self.__class__, self).__init__()` does not work if you subclass again without providing a new `__init__`. Then you have an infinite recursion. – glglgl Mar 31, 2014 at 7:21

25 This answer is ridiculous. If you're going to abuse super this way, you might as well just hardcode the base class name. It is less wrong than this. The whole point of first argument of super is that it's *not* necessarily the type of self. Please read "super considered super" by rhettinger (or watch some of his videos). – Veky Jul 29, 2016 at 12:54

6 The shortcut demonstrated here for Python 2 has pitfalls that have been mentioned already. Don't use this, or your code will break in ways you can't predict. This "handy shortcut" breaks super, but you may not realize it until you've sunk a whole lot of time into debugging. Use Python 3 if super is too verbose. – Ryan Hiebert Jan 13, 2017 at 16:17

@Tino I don't really agree with your edits. It doesn't make sense to say both that one cannot do something and that one shouldn't do it - it is possible to do what I describe in my post, and for the reasons I lay out it is a bad idea. – AnjoMan Nov 22, 2017 at 2:16

5 What makes no sense is to tell someone they can do something that is trivially demonstrated as incorrect. You can alias `echo` to `python`. Nobody would ever suggest it! – Russia Must Remove Putin ♦ Dec 2, 2017 at 23:18

82

Super has no side effects

```
Base = ChildB

Base()
```

works as expected

```
Base = ChildA

Base()
```

gets into infinite recursion.

Share Edit Follow

edited Feb 15 at 4:33
Art
2,703 4 14 28

answered Nov 27, 2012 at 23:26
S C
901 6 2

6 The statement, "Super has no side effects," doesn't make sense in this context. Super simply guarantees we call the correct next class's method in the method resolution order, whereas the other way hard-codes the next method to be called, which makes cooperative multiple inheritance more difficult. – Russia Must Remove Putin ♦ Oct 1, 2017 at 0:53

3 This answer is fragmentary (code examples only makes sense as a continuation of code from the answer.) – MarkHu Sep 2, 2020 at 17:49

74

Just a heads up... with Python 2.7, and I believe ever since `super()` was introduced in version 2.2, you can only call `super()` if one of the parents inherit from a class that eventually inherits `object` (new-style classes).

Personally, as for python 2.7 code, I'm going to continue using `BaseClassName.__init__(self, args)` until I actually get the advantage of using `super()`.

Share Edit Follow

edited Aug 7, 2012 at 18:05

answered May 25, 2012 at 17:52
rgenito
1,731 13 8

5 very good point. IF you don't clearly mention: class Base(object): then you will get error like that: "TypeError: must be type, not classobj" – andilabs Jul 19, 2013 at 11:38

58

There isn't, really. `super()` looks at the next class in the MRO (method resolution order, accessed with `cls.__mro__`) to call the methods. Just calling the base `__init__` calls the base `__init__`. As it happens, the MRO has exactly one item-- the base. So you're really doing the exact same thing, but in a nicer way with `super()` (particularly if you get into multiple inheritance later).

Share Edit Follow

answered Feb 23, 2009 at 0:34
Devin Jeanpierre
90.7k 4 55 79

2 I see. Could you elaborate a little as to why its nicer to use super() with multiple inheritance? To me, the base.__init__(self) is shorter (cleaner). If I had two baseclasses, it would be two of those lines, or two super() lines. Or did I misunderstand what you meant by "nicer"? – Mizipzor Feb 23, 2009 at 0:40

7 Actually, it would be one super() line. When you have multiple inheritance, the MRO is still flat. So the first super().__init__ call calls the next class's **init**, which then calls the next, and so on. You should really check out some docs on it. – Devin Jeanpierre Feb 23, 2009 at 0:45

The child class MRO contains object too - a class's MRO is visible in the **mro** class variable. – James Brady Feb 23, 2009 at 1:24

1 Also note that classic classes (pre 2.2) don't support super - you have to explicitly refer to base classes. – James Brady Feb 23, 2009 at 1:26

"The child class MRO contains object too - a class's MRO is visible in the **mro** class variable." That is a big oops. Whoops. – Devin Jeanpierre Feb 23, 2009 at 4:14

40

The main difference is that `ChildA.__init__` will unconditionally call `Base.__init__` whereas `ChildB.__init__` will call `__init__` in **whatever class happens to be `ChildB` ancestor in `self`'s line of ancestors** (which may differ from what you expect).

If you add a `ClassC` that uses multiple inheritance:

```
class Mixin(Base):
  def __init__(self):
    print "Mixin stuff"
    super(Mixin, self).__init__()

class ChildC(ChildB, Mixin):  # Mixin is now between ChildB and Base
  pass

ChildC()
```

```
help(ChildC) # shows that the Method Resolution Order is ChildC->ChildB->Mixin-
>Base
```

then `Base` **is no longer the parent of `ChildB`** for `ChildC` instances. Now `super(ChildB, self)` will point to `Mixin` if `self` is a `ChildC` instance.

You have inserted `Mixin` in between `ChildB` and `Base`. And you can take advantage of it with `super()`

So if you are designed your classes so that they can be used in a Cooperative Multiple Inheritance scenario, you use `super` because you don't really know who is going to be the ancestor at runtime.

The super considered super post and pycon 2015 accompanying video explain this pretty well.

Share Edit Follow

edited Aug 27, 2020 at 20:25
HoldOffHunger
17.1k 8 93 124

answered Sep 21, 2015 at 6:41
RubenLaguna
19.5k 12 98 134

1 This. The meaning of `super(ChildB, self)` changes depending on the MRO of the object referred to by `self`, which cannot be known until runtime. In other words, the author of `ChildB` has no way of knowing what `super()` will resolve to in all cases unless they can guarantee that `ChildB` will never be subclassed. – nispio Nov 13, 2015 at 4:17