

Real Python

Supercharge Your Classes With Python super()

by Kyle Stratis 53 Comments best-practices intermediate python

Mark as Completed



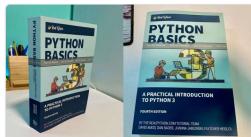
Tweet

Share

Email

Table of Contents

- [An Overview of Python's super\(\) Function](#)
- [super\(\) in Single Inheritance](#)
- [What Can super\(\) Do for You?](#)
- [A super\(\) Deep Dive](#)
- [super\(\) in Multiple Inheritance](#)
 - [Multiple Inheritance Overview](#)
 - [Method Resolution Order](#)
 - [Multiple Inheritance Alternatives](#)
- [A super\(\) Recap](#)



Python Basics: A Practical Introduction to Python 3 Paperback Now Available →

Remove ads

[Watch Now](#) This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Supercharge Your Classes With Python super\(\)](#).

While Python isn't purely an object-oriented language, it's flexible enough and powerful enough to allow you to build your applications using the object-oriented paradigm. One of the ways in which Python achieves this is by supporting **inheritance**, which it does with `super()`.

In this tutorial, you'll learn about the following:

- The concept of inheritance in Python
- Multiple inheritance in Python
- How the `super()` function works

[Improve Your Python](#)

- How the `super()` function in single inheritance works
- How the `super()` function in multiple inheritance works

Free Bonus: [5 Thoughts On Python Mastery](#), a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

An Overview of Python's `super()` Function

If you have experience with object-oriented programming, you've probably heard of the `super()` function.

If not, don't fear! While the [official documentation](#) can be a bit overwhelming, it's just another way to inherit methods in a superclass from the subclass.

`super()` alone returns a temporary object.

Why would you want to do any of this? Well, it's useful for building classes that extend the functionality of existing ones.

Calling the previously built methods with `super()` allows you to swap out superclasses.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)



Monitor Your Python Applications

Trace through your python services and third party APIs
Visualize and understand your microservices-based architecture

[TRY FREE](#)

[Remove ads](#)

super() in Single Inheritance

If you're unfamiliar with object-oriented programming concepts, **inheritance** might be an unfamiliar term. Inheritance is a concept in object-oriented programming in which a class derives (or **inherits**) attributes and behaviors from another class without needing to implement them again.

For me at least, it's easier to understand these concepts when looking at code, so let's write classes describing some shapes:

Python

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

class Square:
    def __init__(self, length):
        self.length = length

    def area(self):
        return self.length * self.length

    def perimeter(self):
        return 4 * self.length
```

Here, there are two similar classes: `Rectangle` and `Square`.

You can use them as below:

Python

>>>

```
>>> square = Square(4)
>>> square.area()
16
>>> rectangle = Rectangle(2,4)
>>> rectangle.area()
8
```

In this example, you have two shapes that are related to each other: a square is a special kind of rectangle. The code, however, doesn't reflect that relationship and thus has code that is essentially repeated.

By using inheritance, you can reduce the relationship between rectangles and squares.

Python

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

# Here we declare that the Square class inherits from the Rectangle class
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)
```

Improve Your Python



...with a fresh **Python Trick** code snippet every couple of days:

Email Address

[Send Python Tricks »](#)

Here, you've used `super()` to call the `__init__()` of the `Rectangle` class, allowing you to use it in the `Square` class without repeating code. Below, the core functionality remains after making changes:

Python

>>>

```
>>> square = Square(4)
>>> square.area()
16
```

In this example, `Rectangle` is the superclass, and `Square` is the subclass.

Because the `Square` and `Rectangle` `__init__()` methods are so similar, you can simply call the superclass's `__init__()` method (`Rectangle.__init__()`) from that of `Square` by using `super()`. This sets the `.length` and `.width` attributes even though you just had to supply a single `length` parameter to the `Square` constructor.

When you run this, even though your `Square` class doesn't explicitly implement it, the call to `.area()` will use the `.area()` method in the superclass and print 16. The `Square` class **inherited** `.area()` from the `Rectangle` class.

Note: To learn more about inheritance and object-oriented concepts in Python, be sure to check out [Inheritance and Composition: A Python OOP Guide](#) and [Object-Oriented Programming \(OOP\) in Python 3](#).

What Can `super()` Do for You?

So what can `super()` do for you in single inheritance?

Like in other object-oriented languages, it allows you to call methods of the superclass in your subclass. The primary use case of this is to extend the functionality of the inherited method.

In the example below, you will create a class `Cube` that inherits from `Square` and extends the functionality of `.area()` (inherited from the `Rectangle` class through `Square`) to calculate the surface area and volume of a `Cube` instance:

Python

```
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)
```

[Improve Your Python](#)

```

super().__init__(length, length)

class Cube(Square):
    def surface_area(self):
        face_area = super().area()
        return face_area * 6

    def volume(self):
        face_area = super().area()
        return face_area * self.length

```

Now that you've built the classes, let's look at the surface area and volume of a cube with a side length of 3.

Python

```

>>> cube = Cube(3)
>>> cube.surface_area()
54
>>> cube.volume()
27

```

```

1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}

```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Caution: Note that in our example above, we call the method on the proxy object itself.

[Send Python Tricks »](#)

Here you have implemented two methods for the `Cube` class: `.surface_area()` and `.volume()`. Both of these calculations rely on calculating the area of a single face, so rather than reimplementing the area calculation, you use `super()` to extend the area calculation.

Also notice that the `Cube` class definition does not have an `__init__()`. Because `Cube` inherits from `Square` and `__init__()` doesn't really do anything differently for `Cube` than it already does for `Square`, you can skip defining it, and the `__init__()` of the superclass (`Square`) will be called automatically.

`super()` returns a delegate object to a parent class, so you call the method you want directly on it: `super().area()`.

Not only does this save us from having to rewrite the area calculations, but it also allows us to change the internal `.area()` logic in a single location. This is especially handy when you have a number of subclasses inheriting from one superclass.



[Learn Python »](#)

[!\[\]\(899d8b7697d64725bf017d3296cfcf1b_img.jpg\) Remove ads](#)

A `super()` Deep Dive

Before heading into multiple inheritance, let's take a quick detour into the mechanics of `super()`.

While the examples above (and below) call `super()` without any parameters, `super()` can also take two parameters: the first is the subclass, and the second parameter is an object that is an instance of that subclass.

First, let's see two examples showing what manipulating the first `variable` can do, using the classes already shown:

Python

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

class Square(Rectangle):
    def __init__(self, length):
        super(Square, self).__init__(length)
```

In Python 3, the `super(Square, self)` call the subclass `Square`, while the second pair `super()` with other classes as well:

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Python

```
class Cube(Square):
    def surface_area(self):
        face_area = super(Square, self).area()
        return face_area * 6

    def volume(self):
        face_area = super(Square, self).area()
        return face_area * self.length
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

In this example, you are setting `Square` as the subclass argument to `super()`, instead of `cube`. This causes `super()` to start searching for a matching method (in this case, `.area()`) at one level above `Square` in the instance hierarchy, in this case `Rectangle`.

In this specific example, the behavior doesn't change. But imagine that `Square` also implemented an `.area()` function that you wanted to make sure `Cube` did not use. Calling `super()` in this way allows you to do that.

Caution: While we are doing a lot of fiddling with the parameters to `super()` in order to explore how it works under the hood, I'd caution against doing this regularly.

The parameterless call to `super()` is recommended and sufficient for most use cases, and needing to change the search hierarchy regularly could be indicative of a larger design issue.

What about the second parameter? Remember, this is an object that is an instance of the class used as the first parameter. For an example, `isinstance(Cube, Square)` must return `True`.

By including an instantiated object, `super()` returns a **bound method**: a method that is bound to the object, which gives the method the object's context such as any instance attributes. If this parameter is not included, the method returned is just a function, unassociated with an object's context.

For more information about bound methods, unbound methods, and functions, read the Python documentation [on its descriptor system](#).

Note: Technically, `super()` doesn't return a method. It returns a **proxy object**. This is an object that delegates calls to the correct class methods without making an additional object in order to do so.

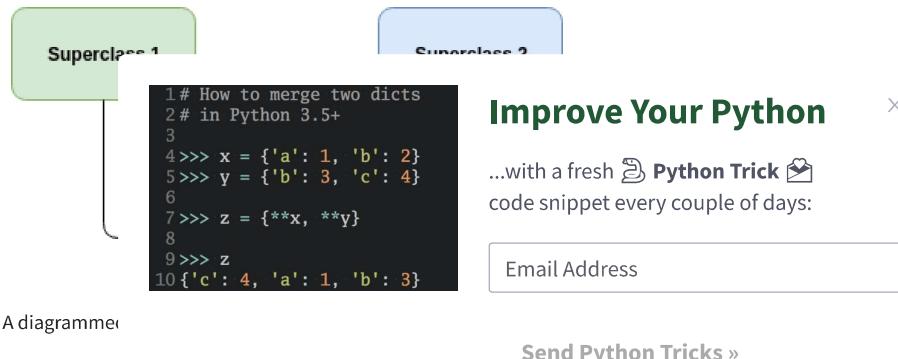
super() in Multiple Inheritance

Now that you've worked through an overview and some examples of `super()` and single inheritance, you will be introduced to an overview and some examples that will demonstrate how multiple inheritance works and how `super()` enables that functionality.

Multiple Inheritance Overview

There is another use case in which `super()` really shines, and this one isn't as common as the single inheritance scenario. In addition to single inheritance, Python supports multiple inheritance, in which a subclass can inherit from multiple superclasses that don't necessarily inherit from each other (also known as **sibling classes**).

I'm a very visual person, and I find diagrams are incredibly helpful to understand concepts like this. The image below shows a very simple multiple inheritance scenario, where one class inherits from two unrelated (sibling) superclasses:



To better illustrate multiple inheritance let's look at some code. We'll start with a simple right pyramid (a pyramid with a square base) built on a triangle and a square:

Python

```
class Triangle:
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

class RightPyramid(Triangle, Square):
    def __init__(self, base, slant_height):
        self.base = base
        self.slant_height = slant_height

    def area(self):
        base_area = super().area()
        perimeter = super().perimeter()
        return 0.5 * perimeter * self.slant_height + base_area
```

Note: The term **slant height** may be unfamiliar, especially if it's been a while since you've taken a geometry class or worked on any pyramids.

The slant height is the height from the center of the base of an object (like a pyramid) up its face to the peak of that object. You can read more about slant heights at [WolframMathWorld](#).

This example declares a `Triangle` class and a `RightPyramid` class that inherits from both `Square` and `Triangle`.

You'll see another `.area()` method that uses `super()` just like in single inheritance, with the aim of it reaching the `.perimeter()` and `.area()` methods defined all the way up in the `Rectangle` class.

Note: You may notice that the code above isn't using any inherited properties from the `Triangle` class yet. Later examples will fully take advantage of inheritance from both `Triangle` and `Square`.

The problem, though, is that both superclasses (`Triangle` and `Square`) define a `.area()`. Take a second and think about what might happen when you call `.area()` on `RightPyramid`, and then try calling it like below:

```
>> pyramid = RightPyramid(2, 4)
>> pyramid.area()
Traceback (most recent call last):
  File "shapes.py", line 63, in <module>
    print(pyramid.area())
  File "shapes.py", line 47, in area
    base_area = super().area()
  File "shapes.py", line 38, in area
    return 0.5 * self.base * self.height
AttributeError: 'RightPyramid' object has no attribute 'height'
```

Did you guess that Python will try to call the **resolution order**.

Note: How did we notice that `Triangle` is listed before `Rectagle` in the last line of the traceback (before the `AttributeError`)?

Python

```
return 0.5 * self.base * self.height
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

You may recognize this from geometry class as the formula for the area of a triangle. Otherwise, if you're like me, you might have scrolled up to the `Triangle` and `Rectagle` class definitions and seen this same code in `Triangle.area()`.



[Learn Python With a Community of Experts »](#)

[Remove ads](#)

Method Resolution Order

The method resolution order (or **MRO**) tells Python how to search for inherited methods. This comes in handy when you're using `super()` because the MRO tells you exactly where Python will look for a method you're calling with `super()` and in what order.

Every class has an `__mro__` attribute that allows us to inspect the order, so let's do that:

Python

>>>

```
>>> RightPyramid.__mro__
(<class '__main__.RightPyramid'>, <class '__main__.Triangle'>,
 <class '__main__.Square'>, <class '__main__.Rectangle'>,
 <class 'object'>)
```

This tells us that methods will be searched first in `RightPyramid`, then in `Triangle`, then in `Square`, then `Rectangle`, and then, if nothing is found, in `object`, from which all classes originate.

The problem here is that the interpreter is searching for `.area()` in `Triangle` before `Square` and `Rectangle`, and upon finding `.area()` in `Triangle`, Python calls it instead of the one you want. Because `Triangle.area()` expects there to be a `.height` and a `.base` attribute, Python throws an `AttributeError`.

Luckily, you have some control over how the MRO is constructed. Just by changing the signature of the `RightPyramid` class, you can search in the order you want, and the methods will resolve correctly:

Python

```
class RightPyramid(Square, Triangle):
    def __init__(self, base, slant_height):
        self.base = base
        self.slant_height = slant_height
        super().__init__(self.base)
```

[Improve Your Python](#)

```

def area(self):
    base_area = super().area()
    perimeter = super().perimeter()
    return 0.5 * perimeter * self.slant_height + base_area

```

Notice that `RightPyramid` initializes partially with the `__init__()` from the `Square` class. This allows `.area()` to use the `.length` on the object, as is designed.

Now, you can build a pyramid, inspect the MRO, and calculate the surface area:

Python

```

>>> pyramid = RightPyramid(2, 4)
>>> RightPyramid.__mro__
(<class '__main__.RightPyramid'>, <class '__main__.Rectangle'>, <class 'object'>)
>>> pyramid.area()
20.0

```

```

1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}

```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Email Address

[Send Python Tricks »](#)

You see that the MRO is now what you'd expect: `RightPyramid`, `Rectangle`, and `object`.

There's still a problem here, though. For the sake of simplicity, I did a few things wrong in this example: the first, and arguably most importantly, was that I had two separate classes with the same method name and signature.

This causes issues with method resolution, because the first instance of `.area()` that is encountered in the MRO list will be called.

When you're using `super()` with multiple inheritance, it's imperative to design your classes to **cooperate**. Part of this is ensuring that your methods are unique so that they get resolved in the MRO, by making sure method signatures are unique—whether by using method names or method parameters.

In this case, to avoid a complete overhaul of your code, you can rename the `Triangle` class's `.area()` method to `.tri_area()`. This way, the area methods can continue using class properties rather than taking external parameters:

Python

```

class Triangle:
    def __init__(self, base, height):
        self.base = base
        self.height = height
        super().__init__()

    def tri_area(self):
        return 0.5 * self.base * self.height

```

Let's also go ahead and use this in the `RightPyramid` class:

Python

```

class RightPyramid(Square, Triangle):
    def __init__(self, base, slant_height):
        self.base = base
        self.slant_height = slant_height
        super().__init__(self.base)

    def area(self):
        base_area = super().area()
        perimeter = super().perimeter()
        return 0.5 * perimeter * self.slant_height + base_area

```

[Improve Your Python](#)

```

def area_2(self):
    base_area = super().area()
    triangle_area = super().tri_area()
    return triangle_area * 4 + base_area

```

The next issue here is that the code doesn't have a delegated `Triangle` object like it does for a `Square` object, so calling `.area_2()` will give us an `AttributeError` since `.base` and `.height` don't have any values.

You need to do two things to fix this:

1. All methods that are called with `super()` means that you will need to add `super().__init__(self)`
2. Redesign all the `__init__()` calls to

[Complete Code Example](#)

```

1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}

```

There are a number of important differences:

- `**kwargs` is modified in some places to instantiate them only with the arguments that make sense for that particular object.
- **Setting up named arguments before `**kwargs`:** You can see this in `RightPyramid.__init__()`. This has the neat effect of popping that key right out of the `**kwargs` dictionary, so that by the time that it ends up at the end of the MRO in the object class, `**kwargs` is empty.

Note: Following the state of `kwargs` can be tricky here, so here's a table of `__init__()` calls in order, showing the class that owns that call, and the contents of `kwargs` during that call:

Class	Named Arguments	kwargs
RightPyramid	base, slant_height	
Square	length	base, height
Rectangle	length, width	base, height
Triangle	base, height	

Now, when you use these updated classes, you have this:

```

Python >>>
>>> pyramid = RightPyramid(base=2, slant_height=4)
>>> pyramid.area()
20.0
>>> pyramid.area_2()
20.0

```

It works! You've used `super()` to successfully navigate a complicated class hierarchy while using both inheritance and composition to create new classes with minimal reimplementations.

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Email Address

[Send Python Tricks »](#)

Multiple Inheritance Alternatives

As you can see, multiple inheritance can be useful but also lead to very complicated situations and code that is hard to read. It's also rare to have objects that neatly inherit everything from more than multiple other objects.

If you see yourself beginning to use multiple inheritance and a complicated class hierarchy, it's worth asking yourself if you can achieve code that is cleaner and easier to understand by using **composition** instead of inheritance. Since this article is focused on inheritance, I will

Luckily, Real Python has published a [dee](#) you an OOP pro in no time.

There's another technique that can help : many of the benefits. This technique is in

A mixin works as a kind of inheritance, but it defines an “includes-a” relationship number of other classes.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

Below, you will see a short example using `volumeMixin` to give specific functionality to our 3D objects – in this case, a volume calculation:

Python

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

class VolumeMixin:
    def volume(self):
        return self.area() * self.height

class Cube(VolumeMixin, Square):
    def __init__(self, length):
        super().__init__(length)
        self.height = length

    def face_area(self):
        return super().area()

    def surface_area(self):
        return super().area() * 6
```

In this example, the code was reworked to include a mixin called `volumeMixin`. The mixin is then used by `Cube` and gives `Cube` the ability to calculate its volume, which is shown below:

Python

>>>

```
>>> cube = Cube(2)
>>> cube.surface_area()
24
>>> cube.volume()
8
```

This mixin can be used the same way in any other class that has an area defined for it and for which the formula `area * height` returns the correct volume.

A `super()` Recap

In this tutorial, you learned how to supercharge your classes with `super()`. Your journey started with a review of single inheritance and then showed how to call superclass methods easily with `super()`.

You then learned how multiple inheritance works in Python, and techniques to combine `super()` with multiple inheritance. You also learned about how Python resolves method calls using the method resolution order (MRO), as well as how to inspect and modify the MRO to ensure appropriate methods are called at appropriate times.

For more information about object-oriented programming in Python and using `super()`, check out these resources:

- [Official `super\(\)` documentation](#)
- [Python's `super\(\)` Considered Super](#)
- [Object-Oriented Programming in Py](#)

 This tutorial has a related written tutorial to deepen your unders

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

[Send Me Python Tricks »](#)

About Kyle Stratis



Kyle is a self-taught developer working as a senior data engineer at Vizit Labs. In the past, he has founded DanqEx (formerly Nasdanq: the original meme stock exchange) and Encryptid Gaming.

[» More about Kyle](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:



Aldren



Geir Arne

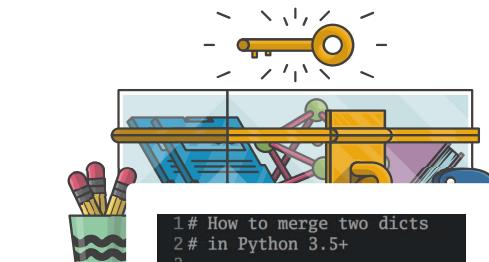


Joanna

[Improve Your Python](#)

Master Real-World Python Skills

With Unlimited Access to Real Python



Join us an
on vi

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh **Python Trick** code snippet every couple of days:

[Send Python Tricks »](#)

What Do You Think?

[Tweet](#) [Share](#) [Email](#)

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: [best-practices](#) [intermediate](#) [python](#)

Recommended Video Course: [Supercharge Your Classes With Python super\(\)](#)

— FREE Email Series —



```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
```

[Improve Your Python](#)

```
8  
9>>> z  
10 {'c': 4, 'a': 1, 'b': 3}
```

Email...

Get Python Tricks »

All Tutorial Topics

[advanced](#) [api](#) [basics](#) [best-practices](#) [community](#) [databases](#) [data-science](#)

[devops](#) [django](#)

[machine-learning](#)

```
1 # How to merge two dicts  
2 # in Python 3.5+  
3  
4 >>> x = {'a': 1, 'b': 2}  
5 >>> y = {'b': 3, 'c': 4}  
6  
7 >>> z = {**x, **y}  
8  
9 >>> z  
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Email Address

Send Python Tricks »

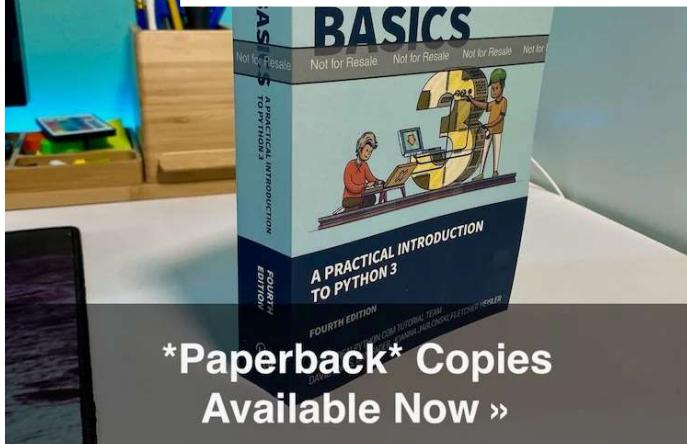


Table of Contents

- [An Overview of Python's super\(\) Function](#)
- [super\(\) in Single Inheritance](#)
- [What Can super\(\) Do for You?](#)
- [A super\(\) Deep Dive](#)
- [super\(\) in Multiple Inheritance](#)
- [A super\(\) Recap](#)

Mark as Completed



 Tweet  Share  Email

 Recommended Video Course

[Supercharge Your Classes With Python super\(\)](#)



```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

X

...with a fresh  **Python Trick** 
code snippet every couple of days:

[Send Python Tricks »](#)