

TUTORIAL

Understanding Class Inheritance in Python 3

Python Development

By [Lisa Tagliaferri](#)

Updated March 9, 2018 490.3k

Introduction

Object-oriented programming creates reusable patterns of code to curtail redundancy in development projects. One way that object-oriented programming achieves recyclable code is through inheritance, when one subclass can leverage code from another base class.

This tutorial will go through some of the major aspects of inheritance in Python, including how parent classes and child classes work, how to override methods and attributes, how to use the `super()` function, and how to make use of multiple inheritance.

SCROLL TO TOP

Inheritance is when a class uses code constructed within another class. If we think of inheritance in terms of biology, we can think of a child inheriting certain traits from their parent. That is, a child can inherit a parent's height or eye color. Children also may share the same last name with their parents.

Classes called **child classes** or **subclasses** inherit methods and variables from **parent classes** or **base classes**.

We can think of a parent class called `Parent` that has class variables for `last_name`, `height`, and `eye_color` that the child class `Child` will inherit from the `Parent`.

Because the `Child` subclass is inheriting from the `Parent` base class, the `Child` class can reuse the code of `Parent`, allowing the programmer to use fewer lines of code and decrease redundancy.

Parent Classes

Parent or base classes create a pattern out of which child or subclasses can be based on. Parent classes allow us to create child classes through inheritance without having to write the same code over again each time. Any class can be made into a parent class, so they are each fully functional classes in their own right, rather than just templates.

Let's say we have a general `Bank_account` parent class that has `Personal_account` and `Business_account` child classes. Many of the methods between personal and business accounts will be similar, such as methods to withdraw and deposit money, so those can belong to the parent class of `Bank_account`. The `Business_account` subclass would have methods specific to it, including perhaps a way to collect business records and forms, as well as an `employee_identification_number` variable.

Similarly, an `Animal` class may have `eating()` and `sleeping()` methods, and a `Snake` subclass may include its own specific `hissing()` and `slithering()` methods.

Let's create a `Fish` parent class that we will later use to construct types of fish as its subclasses. Each of these fish will have first names and last names in addition to characteristics.

We'll create a new file called `fish.py` and start with the `__init__()` constructor method, which we'll populate with `first_name` and `last_name` class variables for each `Fish` object or subclass.

```
class Fish:
    def __init__(self, first_name, last_name="Fish"):
        self.first_name = first_name
        self.last_name = last_name
```

We have initialized our `last_name` variable with the string "Fish" because we know that most fish will have this as their last name.

Let's also add some other methods:

fish.py

```
class Fish:
    def __init__(self, first_name, last_name="Fish"):
        self.first_name = first_name
        self.last_name = last_name

    def swim(self):
        print("The fish is swimming.")

    def swim_backwards(self):
        print("The fish can swim backwards.")
```

We have added the methods `swim()` and `swim_backwards()` to the `Fish` class, so that every subclass will also be able to make use of these methods.

Since most of the fish we'll be creating are considered to be bony fish (as in they have a skeleton made out of bone) rather than cartilaginous fish (as in they have a skeleton made out of cartilage), we can add a few more attributes to the `__init__()` method:

fish.py

```
class Fish:
    def __init__(self, first_name, last_name="Fish",
                  skeleton="bone", eyelids=False):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids

    def swim(self):
        print("The fish is swimming.")

    def swim_backwards(self):
        print("The fish can swim backwards.")
```

thinking about what methods the child classes will be able to make use of once we create those.

Child Classes

Child or subclasses are classes that will inherit from the parent class. That means that each child class will be able to make use of the methods and variables of the parent class.

For example, a `Goldfish` child class that subclasses the `Fish` class will be able to make use of the `swim()` method declared in `Fish` without needing to declare it.

We can think of each child class as being a class of the parent class. That is, if we have a child class called `Rhombus` and a parent class called `Parallelogram`, we can say that a `Rhombus` **is a** `Parallelogram`, just as a `Goldfish` **is a** `Fish`.

The first line of a child class looks a little different than non-child classes as you must pass the parent class into the child class as a parameter:

```
class Trout(Fish):
```

The `Trout` class is a child of the `Fish` class. We know this because of the inclusion of the word `Fish` in parentheses.

With child classes, we can choose to add more methods, override existing parent methods, or simply accept the default parent methods with the `pass` keyword, which we'll do in this case:

fish.py

```
...  
class Trout(Fish):  
    pass
```

We can now create a `Trout` object without having to define any additional methods.

fish.py

```
...  
class Trout(Fish):  
    pass
```

```
print(terry.skeleton)
print(terry.eyelids)
terry.swim()
terry.swim_backwards()
```

We have created a `Trout` object `terry` that makes use of each of the methods of the `Fish` class even though we did not define those methods in the `Trout` child class. We only needed to pass the value of "Terry" to the `first_name` variable because all of the other variables were initialized.

When we run the program, we'll receive the following output:

Output

```
Terry Fish
bone
False
The fish is swimming.
The fish can swim backwards.
```

Next, let's create another child class that includes its own method. We'll call this class `Clownfish`, and its special method will permit it to live with sea anemone:

fish.py

```
...
class Clownfish(Fish):

    def live_with_anemone(self):
        print("The clownfish is coexisting with sea anemone.")
```

Next, let's create a `Clownfish` object to see how this works:

fish.py

```
...
casey = Clownfish("Casey")
print(casey.first_name + " " + casey.last_name)
casey.swim()
casey.live_with_anemone()
```

When we run the program, we'll receive the following output:

The fish is swimming.

The clownfish is coexisting with sea anemone.

The output shows that the `Clownfish` object `casey` is able to use the `Fish` methods `__init__()` and `swim()` as well as its child class method of `live_with_anemone()`.

If we try to use the `live_with_anemone()` method in a `Trout` object, we'll receive an error:

Output

```
terry.live_with_anemone()
AttributeError: 'Trout' object has no attribute 'live_with_anemone'
```

This is because the method `live_with_anemone()` belongs only to the `Clownfish` child class, and not the `Fish` parent class.

Child classes inherit the methods of the parent class it belongs to, so each child class can make use of those methods within programs.

Overriding Parent Methods

So far, we have looked at the child class `Trout` that made use of the `pass` keyword to inherit all of the parent class `Fish` behaviors, and another child class `Clownfish` that inherited all of the parent class behaviors and also created its own unique method that is specific to the child class. Sometimes, however, we will want to make use of some of the parent class behaviors but not all of them. When we change parent class methods we **override** them.

When constructing parent and child classes, it is important to keep program design in mind so that overriding does not produce unnecessary or redundant code.

We'll create a `Shark` child class of the `Fish` parent class. Because we created the `Fish` class with the idea that we would be creating primarily bony fish, we'll have to make adjustments for the `Shark` class that is instead a cartilaginous fish. In terms of program design, if we had more than one non-bony fish, we would most likely want to make separate classes for each of these two types of fish.

Sharks, unlike bony fish, have skeletons made of cartilage instead of bone. They also have eyelids and are unable to swim backwards. Sharks can, however, maneuver themselves backward

look at this child class:

fish.py

```
...
class Shark(Fish):
    def __init__(self, first_name, last_name="Shark",
                  skeleton="cartilage", eyelids=True):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids

    def swim_backwards(self):
        print("The shark cannot swim backwards, but can sink backwards.")
```

We have overridden the initialized parameters in the `__init__()` method, so that the `last_name` variable is now set equal to the string "Shark", the `skeleton` variable is now set equal to the string "cartilage", and the `eyelids` variable is now set to the Boolean value `True`. Each instance of the class can also override these parameters.

The method `swim_backwards()` now prints a different string than the one in the `Fish` parent class because sharks are not able to swim backwards in the way that bony fish can.

We can now create an instance of the `Shark` child class, which will still make use of the `swim()` method of the `Fish` parent class:

fish.py

```
...
sammy = Shark("Sammy")
print(sammy.first_name + " " + sammy.last_name)
sammy.swim()
sammy.swim_backwards()
print(sammy.eyelids)
print(sammy.skeleton)
```

When we run this code, we'll receive the following output:

Output

Sammy Shark

The fish is swimming.

The shark cannot swim backwards, but can sink backwards.

SCROLL TO TOP

The `Shark` child class successfully overrode the `__init__()` and `swim_backwards()` methods of the `Fish` parent class, while also inheriting the `swim()` method of the parent class.

When there will be a limited number of child classes that are more unique than others, overriding parent class methods can prove to be useful.

The `super()` Function

With the `super()` function, you can gain access to inherited methods that have been overwritten in a class object.

When we use the `super()` function, we are calling a parent method into a child method to make use of it. For example, we may want to override one aspect of the parent method with certain functionality, but then call the rest of the original parent method to finish the method.

In a program that grades students, we may want to have a child class for `Weighted_grade` that inherits from the `Grade` parent class. In the child class `Weighted_grade`, we may want to override a `calculate_grade()` method of the parent class in order to include functionality to calculate a weighted grade, but still keep the rest of the functionality of the original class. By invoking the `super()` function we would be able to achieve this.

The `super()` function is most commonly used within the `__init__()` method because that is where you will most likely need to add some uniqueness to the child class and then complete initialization from the parent.

To see how this works, let's modify our `Trout` child class. Since trout are typically freshwater fish, let's add a `water` variable to the `__init__()` method and set it equal to the string `"freshwater"`, but then maintain the rest of the parent class's variables and parameters:

fish.py

```
...
class Trout(Fish):
    def __init__(self, water = "freshwater"):
        self.water = water
        super().__init__(self)
...
```


`__init__()` method of our `Trout` class we have explicitly invoked the `__init__()` method of the `Fish` class.

Because we have overridden the method, we no longer need to pass `first_name` in as a parameter to `Trout`, and if we did pass in a parameter, we would reset `freshwater` instead. We will therefore initialize the `first_name` by calling the variable in our object instance.

Now we can invoke the initialized variables of the parent class and also make use of the unique child variable. Let's use this in an instance of `Trout`:

fish.py

```
...
terry = Trout()

# Initialize first name
terry.first_name = "Terry"

# Use parent __init__() through super()
print(terry.first_name + " " + terry.last_name)
print(terry.eyelids)

# Use child __init__() override
print(terry.water)

# Use parent swim() method
terry.swim()
```

Output

```
Terry Fish
False
freshwater
The fish is swimming.
```

The output shows that the object `terry` of the `Trout` child class is able to make use of both the child-specific `__init__()` variable `water` while also being able to call the `Fish` parent `__init__()` variables of `first_name`, `last_name`, and `eyelids`.

The built-in Python function `super()` allows us to utilize parent class methods even when overriding certain aspects of those methods in our child classes.

class. This can allow programs to reduce redundancy, but it can also introduce a certain amount of complexity as well as ambiguity, so it should be done with thought to overall program design.

To show how multiple inheritance works, let's create a `Coral_reef` child class than inherits from a `Coral` class and a `Sea_anemone` class. We can create a method in each and then use the `pass` keyword in the `Coral_reef` child class:

coral_reef.py

```
class Coral:

    def community(self):
        print("Coral lives in a community.")

class Anemone:

    def protect_clownfish(self):
        print("The anemone is protecting the clownfish.")

class CoralReef(Coral, Anemone):
    pass
```

The `Coral` class has a method called `community()` that prints one line, and the `Anemone` class has a method called `protect_clownfish()` that prints another line. Then we call both classes into the inheritance tuple. This means that `CoralReef` is inheriting from two parent classes.

Let's now instantiate a `CoralReef` object:

coral_reef.py

```
...
great_barrier = CoralReef()
great_barrier.community()
great_barrier.protect_clownfish()
```

The object `great_barrier` is set as a `CoralReef` object, and can use the methods in both parent classes. When we run the program, we'll see the following output:

The output shows that methods from both parent classes were effectively used in the child class.

Multiple inheritance allows us to use the code from more than one parent class in a child class. If the same method is defined in multiple parent methods, the child class will use the method of the first parent declared in its tuple list.

Though it can be used effectively, multiple inheritance should be done with care so that our programs do not become ambiguous and difficult for other programmers to understand.

Conclusion

This tutorial went through constructing parent classes and child classes, overriding parent methods and attributes within child classes, using the `super()` function, and allowing for child classes to inherit from multiple parent classes.

Inheritance in object-oriented coding can allow for adherence to the DRY (don't repeat yourself) principle of software development, allowing for more to be done with less code and repetition. Inheritance also compels programmers to think about how they are designing the programs they are creating to ensure that code is effective and clear.

[Next in series: How To Apply Polymorphism to Classes in Python 3 →](#)

Was this helpful?

Yes

No



[Report an issue](#)

About the authors



Lisa Tagliaferri

Lisa Tagliaferri is Senior Manager of Developer Education at DigitalOcean.

SCROLL TO TOP

Tutorial Series

Object-Oriented Programming in Python 3

Object-oriented programming (OOP) focuses on creating reusable patterns of code, in contrast to procedural programming, which focuses on explicit sequenced instructions. When working on complex programs in particular, object-oriented programming lets you reuse code and write code that is more readable, which in turn makes it more maintainable.

[Next in series: How To Apply Polymorphism to Classes in Python 3 →](#)

How To Code in Python 3

Python is an extremely readable and versatile programming language. Written in a relatively straightforward style with immediate feedback on errors, Python offers simplicity and versatility, in terms of extensibility and supported paradigms.

[Next in series: How To Apply Polymorphism to Classes in Python 3 →](#)

Still looking for an answer?



Ask a question



Search for more help

Comments

18 Comments

[Sign In to Comment](#)[shahzadkhalid](#) December 21, 2017

1

Amazing tutorial. Make me understand the inheritance in OOP, very clearly, for first time . Hats off to Lisa

[Reply](#) [Report](#)[ltagliaferri](#)  December 21, 2017

0

So glad that this has been helpful for you, thanks for your comment!

[Reply](#) [Report](#)[ehmatthes](#) February 23, 2018

0

I think there's an issue in the section about `super()`. In the `__init__()` method for `Trout`, `self` is being passed to the parent `__init__()` method. But this argument is being used as the value for `first_name`. It's not causing an issue because the next line in the example sets `terry.first_name` to "Terry".

If you call `super().__init__()` from a child class, I think you need to include all the required arguments for the parent class' `__init__()` method. I think the fix for this example is to add a parameter for `first_name` in the child class `__init__()` method, and pass that as an argument to `super().__init__()`?

Thank you for this tutorial, this is a really helpful resource!

[Reply](#) [Report](#)[ltagliaferri](#)  March 9, 2018

0

Thanks for your comment. If I understand your question correctly, you can skip the arguments from the parent class, e.g. if you have the `Fish` class as defined in the tutorial, you can define the `Trout(Fish)` class as follows and then run the following without initializing `first_name` while still leveraging `super().__init__(self)`:

fish.py

```
...
class Trout(Fish):
    def __init__(self, water = "freshwater"):
```

[SCROLL TO TOP](#)

```
# create terry object without initializing first_name
terry = Trout()

# Use parent __init__() through super()
print(terry.eyelids)

# Use child __init__() override
print(terry.water)

# Use parent swim() method
terry.swim()
```

You'll receive this output upon running the program:

Output

False

freshwater

The fish is swimming.

[Reply](#) [Report](#)

^ [ehmatthes](#) March 9, 2018
4 Try this:

```
class Trout(Fish):
    def __init__(self, water = "freshwater"):
        self.water = water
        super().__init__(self)

terry = Trout()
print(terry.first_name)
```

Here's the output:

```
<__main__.Trout object at 0x000001E28ED283C8>
```

`Fish.__init__()` requires an argument for `first_name`. But we're passing `self`, which refers to a `Trout` instance. `self` is not a required argument when you **call** an `__init__()` method.

If we call `super().__init__()` without any argument, we can see that `Fish.__i` [SCROLL TO TOP](#)
for an argument for `first_name`:

```
        self.water = water
        super().__init__()

    terry = Trout()
    print(terry.first_name)
```

Here's the output:

```
Traceback (most recent call last):
  File "fish_inheritance.py", line 39, in <module>
    terry = Trout()
  File "fish_inheritance.py", line 37, in __init__
    super().__init__()
TypeError: __init__() missing 1 required positional argument: 'first_name'
```

So I think we either need to give Trout a first name argument and pass it to Fish, or make the first name parameter in Fish optional.

[Reply](#) [Report](#)



Itagliaferri

March 11, 2018

Because the Trout instantiation of terry isn't assigned a name (nor does the parent class Fish assign names to Fish objects), it needs to be given a name, which can be done like this:

```
terry = Trout()
terry.first_name = "Terry"
```

The official Python documentation states:

[The `super()` function returns] a proxy object that delegates method calls to a parent or sibling class of type. This is useful for accessing inherited methods that have been overridden in a class.

So in this example we can still assign a first name to the Trout object but because we have overridden the `__init__()` method we can't pass the first name in the same way that we can when creating the Fish object because in that class it is given as an argument.

There are different ways you can go about this, though, so depending on your needs you may consider modifying the program.

[Reply](#) [Report](#)

[SCROLL TO TOP](#)

Reply Report

 **QueenSveta** May 25, 2018

 **[deleted]**

0

Reply Report

 **QueenSveta** May 24, 2018



1


Hello Lisa,

I came across your tutorial and I really like it, but I noticed some problems, for example like the one ehmatthes pointed out. I've taken the time to outline what the problems are, and how to correct them here: <https://github.com/svetarosemond/digital-ocean-inheritance-correction>

Take a look and let me know what you think.

Reply Report

 **Itagliaferri**  May 24, 2018

 Hi Sveta, thank you for taking the time to offer some alternatives. I will review the tutorial again for clarity and consistency.

0

Reply Report

 **valexia** June 28, 2018



0

Hi Lisa,

After being filed as spam for putting up some useful links, and then a second time for no apparent reason, I'm going to keep my message short(er):

Your tutorial was really helpful to me, but I do agree with ehmatthes. Overriding init doesn't mean that the parent's init changes– it still maintains the same arguments. The child just has a different init that is called when a new child instance is created. When you call a parent's init, you need to pass any non-'self' and non-default arguments as specified in the parent init's definition; otherwise, an error occurs. The solution that ehmatthes proposed is the conventional/correct way to go about making sure the initialization goes smoothly.

That being said, I thought much of content was beneficial. The fun and intuitive examples were especially appreciated. Thanks for the help!

Best,

V

Reply Report

 **Itagliaferri**  July 2, 2018

 Hi V, sorry about the comments being marked as spam. This is an issue that we're looking into.

1

SCROLL TO TOP

[Reply](#) [Report](#)

 [markhayes111](#) June 30, 2018

 0 Wow am I Impressed

I'm a programming beginner learning Python, trying to get clear on Classes and Objects. I'm also a life-long professional writer and former investment industry quant (an odd pairing to be sure).


This tutorial and the two that precede it are some of the clearest, most direct and lucid examples of expository instructional writing I've ever seen.

Absolutely amazing job! I'm old enough to have seen a lot and learned a lot. Nothing is complicated if it's explained well by an expert instructor skilled at illuminating the inner simplicity that lies hidden underneath apparent outer complexity.

These tutorials achieve that and more. Hat's off and bows, Lisa, in your general direction.

[Reply](#) [Report](#)

 [Itagliaferri](#)  July 2, 2018

 1 Thank you so much for taking the time to comment! I am so glad to hear that you are finding this tutorial series useful, and I hope you are enjoying Python :)

[Reply](#) [Report](#)

 [vitezslavik](#) July 9, 2018

 1 Thanks, Lisa, that's exactly what I was searching for. You are super() :)


[Reply](#) [Report](#)

 [Itagliaferri](#)  July 9, 2018

 0 So glad you have found this useful! And thank you for the pun :)

[Reply](#) [Report](#)

 [nutellacookie](#) September 1, 2018

 0 Thank you so much for this tutorial! I'm a beginner in Python and this made learning so fun and simple, great job!! :) I genuinely picked up more in this tutorial than in tons of others!

Could I also trouble you to explain a piece of code I found elsewhere in your own words?
(I've used the shark example provided in the tutorial to replace the context of the code I found elsewhere so it might be easier)

```
Class Shark(Layer):  
    def __init__(self, **kwargs):  
        super(Shark, self).__init__(**kwargs)
```

SCROLL TO TOP

the use of `**kwargs`

Thank you so much and I appreciate your tutorial nonetheless! :)

[Reply](#) [Report](#)



hagertmb September 19, 2018

Hi.

Thanks for the tutorial, it was well written.

One part that doesn't seem right, is where you create a Trout object

```
terry = Trout()
```

and then set the name:

```
terry.first_name = "Terry"
```

Most programming designs frown on this as if anything happens between those 2 lines, you've created an indeterminate state, where a Trout may or may not have a name.

A better solution is to do this:

```
class Trout(Fish):
```

```
def init(self, args, water = "freshwater"):
```

```
    self.water = water
```

```
    super().init(args) # Notice you don't need to pass self into the parent init!
```

Now you can set the name when you create the object as before:

```
terry = Trout("Terry")
```

Everything else is as before.

[Reply](#) [Report](#)

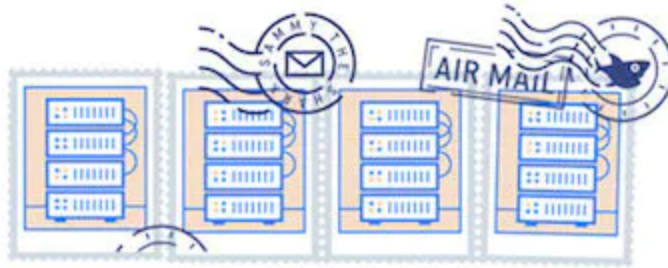


ltagliaferri September 19, 2018

Thanks for the feedback, as mentioned above, this article will be reviewed as part of maintenance.

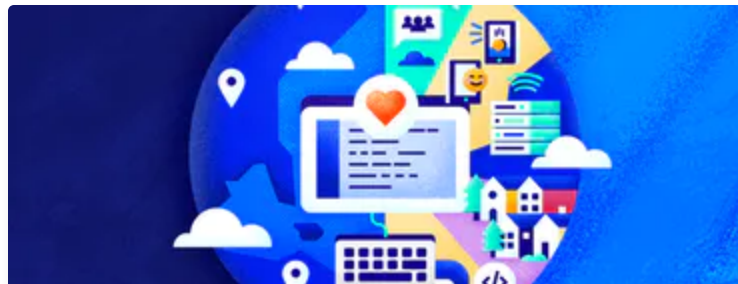
[Reply](#) [Report](#)





GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a
Newsletter.



HUB FOR GOOD

Working on improving health
and education, reducing
inequality, and spurring
economic growth? We'd like to
help.



DEPLOY: NOV 10, 2020

SCROLL TO TOP

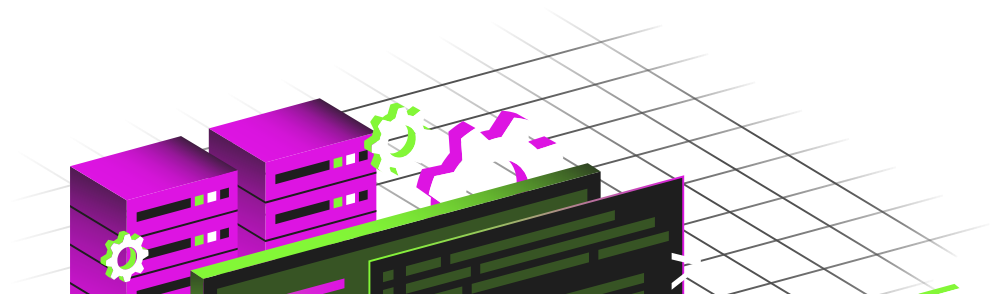
Featured on Community [Kubernetes Course](#) [Learn Python 3](#) [Machine Learning in Python](#) [Getting started with Go](#)
[Intro to Kubernetes](#)

[DigitalOcean Products](#) [Virtual Machines](#) [Managed Databases](#) [Managed Kubernetes](#) [Block Storage](#)
[Object Storage](#) [Marketplace](#) [VPC](#) [Load Balancers](#)

DigitalOcean's first virtual global 24-hour community conference.

Over 80 tech-focused sessions, new product announcements, hourly goody giveaways, RSVP for free and join deploy today!

RSVP today!



© 2020 DigitalOcean, LLC. All rights reserved.

Company

[About](#)
[Leadership](#)
[Blog](#)
[Careers](#)
[Partners](#)
[Referral Program](#)
[Press](#)
[Legal](#)
[Security & Trust Center](#)

Products

[Pricing](#)
[Products Overview](#)
[Droplets](#)
[Kubernetes](#)
[Managed Databases](#)
[Spaces](#)
[Marketplace](#)
[Load Balancers](#) **SCROLL TO TOP**
[Block Storage](#)

Community

Contact

Tutorials

Get Support

Q&A

Trouble Signing In?

Tools and Integrations

Sales

Tags

Report Abuse

Product Ideas

System Status

Write for DigitalOcean

Presentation Grants

Hatch Startup Program

Shop Swag

Research Program

Open Source

Code of Conduct