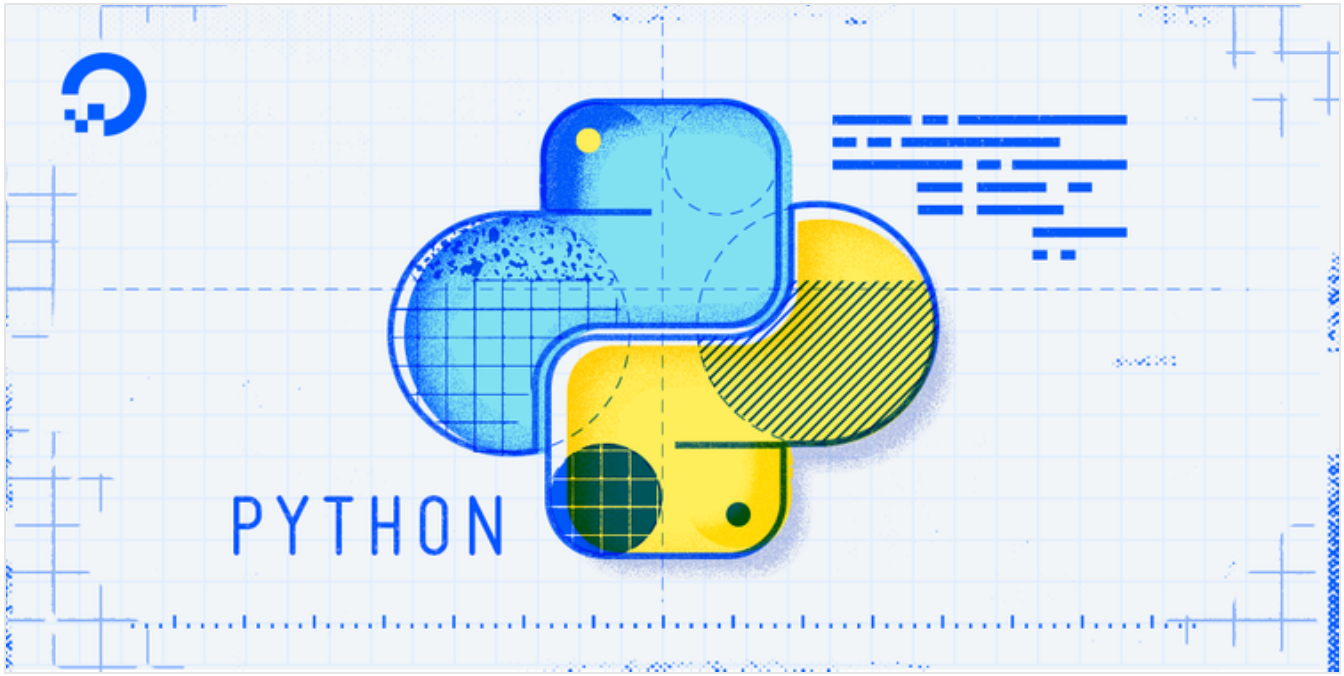
Contents




# How To Use *args and **kwargs in Python 3

Updated November 20, 2017 377.4k PYTHON DEVELOPMENT

By Lisa Tagliaferri
Become an author

## Introduction

interacting with the code. We can pass a variable number of arguments to a function by using `*args` and `**kwargs` in our code.

## Understanding *args

In Python, the single-asterisk form of `*args` can be used as a parameter to send a non-keyworded variable-length argument list to functions. It is worth noting that the asterisk (`*`) is the important element here, as the word `args` is the established conventional idiom, though it is not enforced by the language.

Let's look at a typical function that uses two arguments:

lets_multiply.py

```
def multiply(x, y):
    print (x * y)
```

In the code above, we built the function with `x` and `y` as arguments, and then when we call the function, we need to use numbers to correspond with `x` and `y`. In this case, we will pass the integer `5` in for `x` and the integer `4` in for `y`:

lets_multiply.py

```
def multiply(x, y):
    print (x * y)

multiply(5, 4)
```

Now, we can run the above code:

```
$ python lets_multiply.py
```

We'll receive the following output, showing that the integers 5 and 4 were multiplied as per the `multiply(x,y)` function:

What if, later on, we decide that we would like to multiply three numbers rather than just two? If we try to add an additional number to the function, as shown below, we'll receive an error.

lets_multiply.py

```
def multiply(x, y):
    print (x * y)

multiply(5, 4, 3)
```

```
Output
TypeError: multiply() takes 2 positional arguments but 3 were given
```

So, if you suspect that you may need to use more arguments later on, you can make use of `*args` as your parameter instead.

We can essentially create the same function and code that we showed in the first example, by removing `x` and `y` as function parameters, and instead replacing them with `*args`:

lets_multiply.py

```
def multiply(*args):
    z = 1
    for num in args:
        z *= num
    print(z)

multiply(4, 5)
multiply(10, 9)
multiply(2, 3, 4)
multiply(3, 5, 10, 6)
```

When we run this code, we'll receive the product for each of these function calls:

```
Output
20
```

Because we used `*args` to send a variable-length argument list to our function, we were able to pass in as many arguments as we wished into the function calls.

With `*args` you can create more flexible code that accepts a varied amount of non-keyworded arguments within your function.

## Understanding **kwargs

The double asterisk form of `**kwargs` is used to pass a keyworded, variable-length argument dictionary to a function. Again, the two asterisks (`**`) are the important element here, as the word `kwargs` is conventionally used, though not enforced by the language.

Like `*args`, `**kwargs` can take however many arguments you would like to supply to it. However, `**kwargs` differs from `*args` in that you will need to assign keywords.

First, let's simply print out the `**kwargs` arguments that we pass to a function. We'll create a short function to do this:

print_kwargs.py

```
def print_kwargs(**kwargs):
        print(kwargs)
```

Next, we'll call the function with some keyworded arguments passed into the function:

print_kwargs.py

```
def print_kwargs(**kwargs):
        print(kwargs)

print_kwargs(kwargs_1="Shark", kwargs_2=4.5, kwargs_3=True)
```

Let's run the program above and look at the output:

Depending on the version of Python 3 you are currently using, the dictionary data type may be unordered. In Python 3.6 and above, you'll receive the key-value pairs in order, but in earlier versions, the pairs will be output in a random order.

What is important to note is that a dictionary called `kwargs` is created and we can work with it just like we can work with other dictionaries.

Let's create another short program to show how we can make use of `**kwargs`. Here we'll create a function to greet a dictionary of names. First, we'll start with a dictionary of two names:

print_values.py

```
def print_values(**kwargs):
    for key, value in kwargs.items():
        print("The value of {} is {}".format(key, value))

print_values(my_name="Sammy", your_name="Casey")
```

We can now run the program and look at the output:

```
$ python print_values.py
```

```
Output
The value of your_name is Casey
The value of my_name is Sammy
```

Again, because dictionaries may be unordered, your output may be with the name `Casey` first or with the name `Sammy` first.

Let's now pass additional arguments to the function to show that `**kwargs` will accept however many arguments you would like to include:

```
print_values(
            name_1="Alex",
            name_2="Gray",
            name_3="Harper",
            name_4="Phoenix",
            name_5="Remy",
            name_6="Val"
        )
```

When we run the program at this point, we'll receive the following output, which may again be unordered:

```
Output
The value of name_2 is Gray
The value of name_6 is Val
The value of name_4 is Phoenix
The value of name_5 is Remy
The value of name_3 is Harper
The value of name_1 is Alex
```

Using `**kwargs` provides us with flexibility to use keyword arguments in our program. When we use `**kwargs` as a parameter, we don't need to know how many arguments we would eventually like to pass to a function.

## Ordering Arguments

When ordering arguments within a function or function call, arguments need to occur in a particular order:

1. Formal positional arguments
2. `*args`
3. Keyword arguments
4. `**kwargs`

```
def example(arg_1, arg_2, *args, **kwargs):
...
```

And, when working with positional parameters along with named keyword parameters in addition to `*args` and `**kwargs`, your function would look like this:

```
def example2(arg_1, arg_2, *args, kw_1="shark", kw_2="blobfish", **kwargs):
...
```

It is important to keep the order of arguments in mind when creating functions so that you do not receive a syntax error in your Python code.

## Using *args and **kwargs in Function Calls

We can also use `*args` and `**kwargs` to pass arguments into functions.

First, let's look at an example with `*args`.

some_args.py

```
def some_args(arg_1, arg_2, arg_3):
    print("arg_1:", arg_1)
    print("arg_2:", arg_2)
    print("arg_3:", arg_3)

args = ("Sammy", "Casey", "Alex")
some_args(*args)
```

In the function above, there are three parameters defined as `arg_1`, `arg_`, and `arg_3`. The function will print out each of these arguments. We then create a variable that is set to an iterable (in this case, a tuple), and can pass that variable into the function with the asterisk syntax.

When we run the program with the `python some_args.py` command, we'll receive the following output:

We can also modify the program above to an iterable list data type with a different variable name. Let's also combine the `*args` syntax with a named parameter:

some_args.py

```
def some_args(arg_1, arg_2, arg_3):
    print("arg_1:", arg_1)
    print("arg_2:", arg_2)
    print("arg_3:", arg_3)

my_list = [2, 3]
some_args(1, *my_list)
```

If we run the program above, it will produce the following output:

```
Output
arg_1: 1
arg_2: 2
arg_3: 3
```

Similarly, the keyworded `**kwargs` arguments can be used to call a function. We will set up a variable equal to a dictionary with 3 key-value pairs (we'll use `kwargs` here, but it can be called whatever you want), and pass it to a function with 3 arguments:

some_kwargs.py

```
def some_kwargs(kwarg_1, kwarg_2, kwarg_3):
    print("kwarg_1:", kwarg_1)
    print("kwarg_2:", kwarg_2)
    print("kwarg_3:", kwarg_3)

kwargs = {"kwarg_1": "Val", "kwarg_2": "Harper", "kwarg_3": "Remy"}
some_kwargs(**kwargs)
```

Let's run the program above with the `python some_kwargs.py` command:

```
kwarg_2: Harper
kwarg_3: Remy
```

When calling a function, you can use `*args` and `**kwargs` to pass arguments.

## Conclusion

We can use the special syntax of `*args` and `**kwargs` within a function definition in order to pass a variable number of arguments to the function.

Creating functions that accept `*args` and `**kwargs` are best used in situations where you expect that the number of inputs within the argument list will remain relatively small. The use of `*args` and `**kwargs` is primarily to provide readability and convenience, but should be done with care.

Next in series: How To Construct Classes and Define Objects in Python 3 →

By Lisa Tagliaferri



## Tutorial Series

simplicity and versatility, in terms of extensibility and supported paradigms.

Next in series: How To Construct Classes and Define Objects in Python 3 →

## Related

MEETUP KIT

### Getting Started with Containers and Kubernetes: A DigitalOcean Workshop Kit

This workshop kit is designed to equip a...

TUTORIAL

### How To Display Data from the DigitalOcean API with Django

Django is a web framework written in Python. Django has been used in website...

TUTORIAL

### How To Use Node.js Modules with npm and package.json

The Node.js Package Manager (npm) is the default and most popular...

TUTORIAL

### How To Add Sidekiq and Redis to a Ruby on Rails Application

When developing a Ruby on Rails application, you may find that you have...

## 9 Comments



daviddexter *May 9, 2017*

4 Brilliant refresher.

Report

pegom0896 *August 14, 2017*

1 Thank you for your tutorial. Simple to understand and comprehensive.

Report

gucciferXCIV *November 20, 2017*

0 As of 3.6 dicts stay in order. :)

Report



hon 3 versions.

0 I appreciated this overview. Thank you.

> What is important to note is that a dictionary called **kwargs is created

The dictionary is actually just called `kwargs`, isn't it?

Report

> ltagliaferri MOD *May 17, 2018*
>
> 0 Thanks for pointing that out, I have updated the tutorial to reflect the change.
>
> Report

tariqywsf *July 27, 2019*

0 Thanks , helped me alot

Report

alans *October 29, 2019*

0 Thanks

Report

hioymaci *November 14, 2019*

0 Thank you so much. All Digitalocean tutorials are very clear and easy to understand.

Report







Company

About
Leadership
Blog
Careers
Partners
Referral Program
Press
Legal & Security

Products

Products Overview
Pricing
Droplets
Kubernetes
Managed Databases
Spaces
Marketplace
Load Balancers
Block Storage
Tools & Integrations
API
Documentation
Release Notes

Community

Tutorials
Q&A
Tools and Integrations
Tags
Product Ideas
Meetups
Write for DOnations
Droplets for Demos
Hatch Startup Program
Shop Swag
Research Program
Currents Research
Open Source

Contact

Support
Sales
Report Abuse
System Status