Python args and kwargs: Demystified

by Davide Mastromatteo 🗨 34 Comments 🏷 intermediate   python

Mark as Completed   🔖   🐦 **Tweet**   f **Share**   ✉ **Email**

Table of Contents

> ▶ 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: **Python args and kwargs: Demystified**

Sometimes, when you look at a function definition in Python, you might see that it takes two strange arguments: `*args` and `**kwargs`. If you've ever wondered what these peculiar variables are, or why your IDE defines them in `main()`, then this article is for you. You'll learn how to use args and kwargs in Python to add more flexibility to your functions.

By the end of the article, you'll know:

- What *args and **kwargs actually mean
- How to use *args and **kwargs in function definitions
- How to use a single asterisk (*) to unpack iterables
- How to use two asterisks (**) to unpack dictionaries

This article assumes that you already know how to [define Python functions](#) and work with [lists and dictionaries](#).

Passing Multiple Arguments to a Function

*args and ****kwargs** allow you to pass multiple arguments or keyword arguments to a function. Consider the following example. This is a simple function that takes two arguments and returns their sum:

```python
def my_sum(a, b):
    return a + b
```

This function works fine, but it's limited to only two arguments. What if you need to sum a varying number of arguments, where the specific number of arguments passed is only determined at runtime? Wouldn't it be great to create a function that could sum *all* the integers passed to it, no matter how many there are?

Using the Python args Variable in Function Definitions

There are a few ways you can pass a varying number of arguments to a function. The first way is often the most intuitive for people that have experience with collections. You simply pass a list or a [set](#) of all the arguments to your function. So for `my_sum()`, you could pass a list of all the integers you need to add:

```python
# sum_integers_list.py
def my_sum(my_integers):
    result = 0
    for x in my_integers:
        result += x
    return result

list_of_integers = [1, 2, 3]
print(my_sum(list_of_integers))
```

This implementation works, but whenever you call this function you'll also need to create a list of arguments to pass to it. This can be inconvenient, especially if you don't know up front all the values that should go into the list.

This is where `*args` can be really useful, because it allows you to pass a varying number of positional arguments. Take the following example:

```python
# sum_integers_args.py
def my_sum(*args):
    result = 0
    # Iterating over the Python args tuple
    for x in args:
        result += x
    return result

print(my_sum(1, 2, 3))
```

In this example, you're no longer passing a list to `my_sum()`. Instead, you're passing three different positional arguments. `my_sum()` takes all the parameters that are provided in the input and packs them all into a single iterable object named `args`.

Note that `args` **is just a name.** You're not required to use the name `args`. You can choose any name that you prefer, such as `integers`:

```python
# sum_integers_args_2.py
def my_sum(*integers):
    result = 0
    for x in integers:
        result += x
    return result

print(my_sum(1, 2, 3))
```

The function still works, even if you pass the iterable object as `integers` instead of `args`. All that matters here is that you use the **unpacking operator** (*).

Bear in mind that the iterable object you'll get using the unpacking operator * is [not a list but a tuple](#). A `tuple` is similar to a `list` in that they both support slicing and iteration. However, tuples are very different in at least one aspect: lists are [mutable](#), while tuples are not. To test this, run the following code. This script tries to change a value of a list:

```python
# change_list.py
my_list = [1, 2, 3]
my_list[0] = 9
print(my_list)
```

The value located at the very first index of the list should be updated to `9`. If you execute this script, you will see that the list indeed gets modified:

```shell
$ python change_list.py
[9, 2, 3]
```

The first value is no longer `0`, but the updated value `9`. Now, try to do the same with a tuple:

```python
# change_tuple.py
my_tuple = (1, 2, 3)
my_tuple[0] = 9
print(my_tuple)
```

Here, you see the same values, except they're held together as a tuple. If you try to execute this script, you will see that the Python interpreter returns an [error](#):

```shell
$ python change_tuple.py
Traceback (most recent call last):
  File "change_tuple.py", line 3, in <module>
    my_tuple[0] = 9
TypeError: 'tuple' object does not support item assignment
```

This is because a tuple is an immutable object, and its values cannot be changed after assignment. Keep this in mind when you're working with tuples and `*args`.

Using the Python kwargs Variable in Function Definitions

Okay, now you've understood what `*args` is for, but what about `**kwargs`? `**kwargs` works just like `*args`, but instead of accepting positional arguments it accepts keyword (or **named**) arguments. Take the following example:

```python
# concatenate.py
def concatenate(**kwargs):
    result = ""
    # Iterating over the Python kwargs dictionary
    for arg in kwargs.values():
        result += arg
    return result


print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

When you execute the script above, `concatenate()` will iterate through the Python kwargs [dictionary](#) and concatenate all the values it finds:

```shell
$ python concatenate.py
RealPythonIsGreat!
```

Like `args`, `kwargs` is just a name that can be changed to whatever you want. Again, what is important here is the use of the **unpacking operator** (**).

So, the previous example could be written like this:

```python
# concatenate_2.py
def concatenate(**words):
    result = ""
    for arg in words.values():
        result += arg
    return result


print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

Note that in the example above the iterable object is a standard `dict`. If you [iterate over the dictionary](#) and want to return its values, like in the example shown, then you must use `.values()`.

In fact, if you forget to use this method, you will find yourself iterating through the **keys** of your Python kwargs dictionary instead, like in the following example:

```python
# concatenate_keys.py
def concatenate(**kwargs):
    result = ""
    # Iterating over the keys of the Python kwargs dictionary
    for arg in kwargs:
        result += arg
    return result


print(concatenate(a="Real", b="Python", c="Is", d="Great", e="!"))
```

Now, if you try to execute this example, you'll notice the following output:

```shell
$ python concatenate_keys.py
abcde
```

As you can see, if you don't specify `.values()`, your function will iterate over the keys of your Python kwargs dictionary, returning the wrong result.

Ordering Arguments in a Function

Now that you have learned what `*args` and `**kwargs` are for, you are ready to start writing functions that take a varying number of input arguments. But what if you want to create a function that takes a changeable number of both positional *and* named arguments?

In this case, you have to bear in mind that **order counts**. Just as non-default arguments have to precede default arguments, so `*args` must come before `**kwargs`.

To recap, the correct order for your parameters is:

1. Standard arguments
2. `*args` arguments
3. `**kwargs` arguments

For example, this function definition is correct:

```python
# correct_function_definition.py
def my_function(a, b, *args, **kwargs):
    pass
```

The `*args` variable is appropriately listed before `**kwargs`. But what if you try to modify the order of the arguments? For example, consider the following function:

```python
# wrong_function_definition.py
def my_function(a, b, **kwargs, *args):
    pass
```

Now, `**kwargs` comes before `*args` in the function definition. If you try to run this example, you'll receive an error from the interpreter:

```shell
$ python wrong_function_definition.py
  File "wrong_function_definition.py", line 2
    def my_function(a, b, **kwargs, *args):
                                    ^
SyntaxError: invalid syntax
```

In this case, since `*args` comes after `**kwargs`, the Python interpreter throws a [SyntaxError](#).

Unpacking With the Asterisk Operators: * & **

You are now able to use `*args` and `**kwargs` to define Python functions that take a varying number of input arguments. Let's go a little deeper to understand something more about the **unpacking operators**.

The single and double asterisk unpacking operators were introduced in Python 2. As of the 3.5 release, they have become even more powerful, thanks to [PEP 448](#). In short, the unpacking operators are operators that unpack the values from iterable objects in Python. The single asterisk operator * can be used on any iterable that Python provides, while the double asterisk operator ** can only be used on dictionaries.

Let's start with an example:

```python
# print_list.py
my_list = [1, 2, 3]
print(my_list)
```

This code defines a list and then prints it to the standard output:

```
Shell
$ python print_list.py
[1, 2, 3]
```

Note how the list is printed, along with the corresponding brackets and commas.

Now, try to prepend the unpacking operator * to the name of your list:

```
Python
# print_unpacked_list.py
my_list = [1, 2, 3]
print(*my_list)
```

Here, the * operator tells `print()` to unpack the list first.

In this case, the output is no longer the list itself, but rather *the content* of the list:

```
Shell
$ python print_unpacked_list.py
1 2 3
```

Can you see the difference between this execution and the one from `print_list.py`? Instead of a list, `print()` has taken three separate arguments as the input.

Another thing you'll notice is that in `print_unpacked_list.py`, you used the unpacking operator * to call a function, instead of in a function definition. In this case, `print()` takes all the items of a list as though they were single arguments.

You can also use this method to call your own functions, but if your function requires a specific number of arguments, then the iterable you unpack must have the same number of arguments.

To test this behavior, consider this script:

```
Python
# unpacking_call.py
def my_sum(a, b, c):
    print(a + b + c)

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

Here, `my_sum()` explicitly states that a, b, and c are required arguments.

If you run this script, you'll get the sum of the three numbers in `my_list`:

```
Shell
$ python unpacking_call.py
6
```

The 3 elements in `my_list` match up perfectly with the required arguments in `my_sum()`.

Now look at the following script, where `my_list` has 4 arguments instead of 3:

```
Python
# wrong_unpacking_call.py
def my_sum(a, b, c):
    print(a + b + c)

my_list = [1, 2, 3, 4]
my_sum(*my_list)
```

In this example, `my_sum()` still expects just three arguments, but the * operator gets 4 items from the list. If you try to execute this script, you'll see that the Python interpreter is unable to run it:

```
$ python wrong_unpacking_call.py
Traceback (most recent call last):
  File "wrong_unpacking_call.py", line 6, in <module>
    my_sum(*my_list)
TypeError: my_sum() takes 3 positional arguments but 4 were given
```

When you use the * operator to unpack a list and pass arguments to a function, it's exactly as though you're passing every single argument alone. This means that you can use multiple unpacking operators to get values from several lists and pass them all to a single function.

To test this behavior, consider the following example:

Python

```python
# sum_integers_args_3.py
def my_sum(*args):
    result = 0
    for x in args:
        result += x
    return result

list1 = [1, 2, 3]
list2 = [4, 5]
list3 = [6, 7, 8, 9]

print(my_sum(*list1, *list2, *list3))
```

If you run this example, all three lists are unpacked. Each individual item is passed to `my_sum()`, resulting in the following output:

Shell

```
$ python sum_integers_args_3.py
45
```

There are other convenient uses of the unpacking operator. For example, say you need to split a list into three different parts. The output should show the first value, the last value, and all the values in between. With the unpacking operator, you can do this in just one line of code:

Python

```python
# extract_list_body.py
my_list = [1, 2, 3, 4, 5, 6]

a, *b, c = my_list

print(a)
print(b)
print(c)
```

In this example, `my_list` contains 6 items. The first variable is assigned to `a`, the last to `c`, and all other values are packed into a new list `b`. If you run the [script](#), `print()` will show you that your three variables have the values you would expect:

Shell

```
$ python extract_list_body.py
1
[2, 3, 4, 5]
6
```

Another interesting thing you can do with the unpacking operator * is to split the items of any iterable object. This could be very useful if you need to merge two lists, for instance:

```python
# merging_lists.py
my_first_list = [1, 2, 3]
my_second_list = [4, 5, 6]
my_merged_list = [*my_first_list, *my_second_list]

print(my_merged_list)
```

The unpacking operator `*` is prepended to both `my_first_list` and `my_second_list`.

If you run this script, you'll see that the result is a merged list:

```shell
$ python merging_lists.py
[1, 2, 3, 4, 5, 6]
```

You can even merge two different dictionaries by using the unpacking operator `**`:

```python
# merging_dicts.py
my_first_dict = {"A": 1, "B": 2}
my_second_dict = {"C": 3, "D": 4}
my_merged_dict = {**my_first_dict, **my_second_dict}

print(my_merged_dict)
```

Here, the iterables to merge are `my_first_dict` and `my_second_dict`.

Executing this code outputs a merged dictionary:

```shell
$ python merging_dicts.py
{'A': 1, 'B': 2, 'C': 3, 'D': 4}
```

Remember that the `*` operator works on *any* iterable object. It can also be used to unpack a [string](#):

```python
# string_to_list.py
a = [*"RealPython"]
print(a)
```

In Python, strings are iterable objects, so `*` will unpack it and place all individual values in a list `a`:

```shell
$ python string_to_list.py
['R', 'e', 'a', 'l', 'P', 'y', 't', 'h', 'o', 'n']
```

The previous example seems great, but when you work with these operators it's important to keep in mind the seventh rule of [*The Zen of Python*](#) by Tim Peters: *Readability counts*.

To see why, consider the following example:

```python
# mysterious_statement.py
*a, = "RealPython"
print(a)
```

There's the unpacking operator `*`, followed by a variable, a comma, and an assignment. That's a lot packed into one line! In fact, this code is no different from the previous example. It just takes the string `RealPython` and assigns all the items to the new list `a`, thanks to the unpacking operator `*`.

The comma after the a does the trick. When you use the unpacking operator with variable assignment, Python requires that your resulting variable is either a list or a tuple. With the trailing comma, you have defined a tuple with only one named variable, a, which is the list `['R', 'e', 'a', 'l', 'P', 'y', 't', 'h', 'o', 'n']`.

Where's the tuple?	Show/Hide

While this is a neat trick, many Pythonistas would not consider this code to be very readable. As such, it's best to use these kinds of constructions sparingly.

Conclusion

You are now able to use `*args` and `**kwargs` to accept a changeable number of arguments in your functions. You have also learned something more about the unpacking operators.

You've learned:

- What `*args` and `**kwargs` actually mean
- How to use `*args` and `**kwargs` in function definitions
- How to use a single asterisk (*) to unpack iterables
- How to use two asterisks (**) to unpack dictionaries

If you still have questions, don't hesitate to reach out in the comments section below! To learn more about the use of the asterisks in Python, have a look at Trey Hunner's article on the subject.

Mark as Completed	🔖	👍	👎

> ▶ **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: **Python args and kwargs: Demystified**

🐍 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}
```

Email Address

Send Me Python Tricks »

About **Davide Mastromatteo**

Developer and editor of "the Python Corner". Blood donor, Apple user, Python and Swift addicted
NFL, Rugby and Chess lover. Constantly hungry and foolish.

» More about Davide

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:
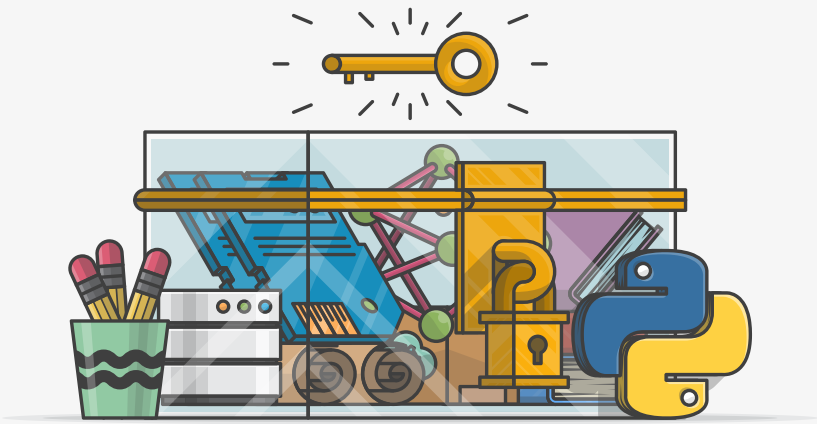
Aldren

Geir Arne

Jaya

Joanna

Mike

Master Real-World Python Skills
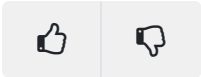## With Unlimited Access to Real Python

**Join us and get access to thousands of tutorials,
hands-on video courses, and a community of expert
Pythonistas:**

Level Up Your Python Skills »

What Do You Think?

Rate this article: 👍   👎

Tweet   Share   Share   Email

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.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next ["Office Hours" Live Q&A Session](#). Happy Pythoning!

Keep Learning

Related Tutorial Categories: `intermediate` `python`

Recommended Video Course: [Python args and kwargs: Demystified](#)

— FREE Email Series —

🐍 Python Tricks 💌

```
 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}
```

Email…

Get Python Tricks »

🔒 No spam. Unsubscribe any time.

All Tutorial Topics

`advanced` `api` `basics` `best-practices` `community` `databases` `data-science` `devops` `django` `docker` `flask` `front-end` `gamedev` `gui` `intermediate` `machine-learning` `projects` `python` `testing` `tools` `web-dev` `web-scraping`

Table of Contents

Mark as Completed 🔖

👍 👎

🐦 Tweet   f Share   ✉ Email

▶ Recommended Video Course

[Python args and kwargs: Demystified](#)

© 2012–2023 Real Python · [Newsletter](#) · [Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) · [Instagram](#) · [Python Tutorials](#) · [Search](#) · [Privacy Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)
❤️ Happy Pythoning!