mingrammer Follow
I love computer science, automation and mathematics. github.com/mingrammer
Mar 20, 2017 · 4 min read

Understanding the asterisk(*) of Python

> *I'm not a native speaker. Please understand.*

Python has plentiful types of operations compared to other languages.

Especially, the **Asterisk(*)** that is one of the most used operators in Python allows us to enable various operations more than just multiplying the two numbers. In this post, we'll look at the various operations that can be done with this **Asterisk(*)** to write Python more pythonically.

There are 4 cases for using the asterisk in Python.

1. For multiplication and power operations.
2. For repeatedly extending the list-type containers.
3. For using the variadic arguments. (so-called "unpacking")
4. For unpacking the containers.

Let's look at each case.

When used in multiplication and power operations

You may already know of this case. Python supports the built-in *power* operations as well as *multiplication*.

```
>>> 2 * 3
6
>>> 2 ** 3
8
>>> 1.414 * 1.414
1.9993959999999997
```

For repeatedly extending the list-type containers

Python also supports that multiply the *list-type container (includes tuple)* and *int* for extending container data by given number times.

```
# Initialize the zero-valued list with 100 length
zeros_list = [0] * 100

# Declare the zero-valued tuple with 100 length
zeros_tuple = (0,) * 100

# Extending the "vector_list" by 3 times
vector_list = [[1, 2, 3]]
for i, vector in enumerate(vector_list * 3):
    print("{0} scalar product of vector: {1}".format(
```

For using the variadic arguments

We often need variadic arguments (or parameters) for some functions. For example, we need it if we don't know number of passing arguments or when we should process something with arbitrary passing arguments for some reasons.

There are 2 kinds of arguments in Python, one is **positional arguments** and other is **keyword arguments**, the former are specified according to their position and latter are the arguments with keyword which is the name of the argument.

Before looking at the *variadic positional/keyword arguments,* we'll talk about the positional arguments and keyword arguments simply.

```
# A function that shows the results of running competitio
def save_ranking(first, second, third=None, fourth=None):
    rank = {}
    rank[1], rank[2] = first, second
    rank[3] = third if third is not None else 'Nobody'
    rank[4] = fourth if fourth is not None else 'Nobody'
    print(rank)

# Pass the 2 positional arguments
save_ranking('ming', 'alice')
```

Above function has 2 *positional arguments*: `first` , `second` and 2 *keyword arguments*: `third` , `fourth` . For positional arguments, it is not possible to omit it, and you must pass all positional arguments to the correct location for each number of arguments declared. However, for keyword arguments, you can set a default value of it when declaring a function, and if you omit the argument, the corresponding default value is entered as the value of the argument. That is, the keyword arguments can be omitted.

Thus, what you can see here is that keyword arguments can be omitted, so they can not be declared before positional arguments. So, the following code will raises exceptions:

```
def save_ranking(first, second=None, third, fourth=Non
    ...
```

But, in the third case, you can see that there are 3 *positional arguments* and 1 *keyword argument*. Yes, for keyword arguments, if the passed position is the same to declared position, the keyword can be excluded and passed as positional arguments. That is, in above, the `mike` will be passed to `third` key automatically.

So far we've talked about the basic of arguments. By the way, one problem can be met here. The function can not handle the arbitrary numbers of runners because the function has fixed numbers of arguments. So we need the **variadic arguments** for it. Both *positional arguments* and *keyword arguments* can be used as *variadic arguments*. Let's see following examples.

When use only positional arguments

```
def save_ranking(*args):
    print(args)
save_ranking('ming', 'alice', 'tom', 'wilson', 'roy')
# ['ming', 'alice', 'tom', 'wilson', 'roy']
```

When use only keyword arguments

```
def save_ranking(**kwargs):
    print(kwargs)
save_ranking(first='ming', second='alice', fourth='wil
# {'first': 'ming', 'second': 'alice', 'fourth': 'wils
```

When use both positional arguments and keyword arguments

```
def save_ranking(*args, **kwargs):
    print(args)
    print(kwargs)
save_ranking('ming', 'alice', 'tom', fourth='wilson',
# {'fourth': 'wilson', 'fifth': 'roy'}
```

In above, `*args` means accepting the arbitrary numbers of *positional arguments* and `**kwargs` means accepting the arbitrary numbers of *keyword arguments*. In here, `*args` , `**kwargs` are called **unpacking**.

As you can see above, we are passing the arguments which can hold arbitrary numbers of positional or keyword values. The arguments passed as positional are stored in a *list* called `args` , and the arguments passed as keyword are stored in a *dict* called `kwargs` .

As refered before, the *keyword arguments* can not be declared before *positional arguments*, so following code should raises exceptions:

```
def save_ranking(**kwargs, *args):
    ...
```

The *variadic argument* is very often used feature, it could be seen on many open source projects. Usually, many open sources use typically used argument names such as `*args` or `**kwargs` as variadic arguments name. But, of course, you can also use the own name for it like `*required` or `**optional` . (However, if your project is open source and there is no special meaning at variadic arguments, it is good to follow conventions of using `*args` and `**kwarg`)

For unpacking the containers

The * can also be used for unpacking the containers. Its principles is similar to *"For using the variadic arguments"* in above. The easiest example is that we have data in the form of a *list*, *tuple* or *dict*, and a function take variable arguments:

```
from functools import reduce

primes = [2, 3, 5, 7, 11, 13]

def product(*numbers):
    p = reduce(lambda x, y: x * y, numbers)
    return p

product(*primes)
```

Because the `product()` take the variable arguments, we need to unpack the our list data and pass it to that function. In this case, if we pass the `primes` as `*primes` , every elements of the `primes` list will be unpacked, then stored in list called `numbers` . If pass that list `primes` to the function without unpacking, the `numbers` will has only one `primes` list **not** all elements of `primes` .

For tuple, it could be done exactly same to list, and for dict, just use ** instead of *.

```python
headers = {
    'Accept': 'text/plain',
    'Content-Length': 348,
    'Host': 'http://mingrammer.com'
}

def pre_process(**headers):
    content_length = headers['Content-Length']
    print('content length: ', content_length)

    host = headers['Host']
    if 'https' not in host:
        raise ValueError('You must use SSL for http c
```

And there is also one more type of unpacking, it is not for function but just unpack the list or tuple data to other variables dynamically.

```python
numbers = [1, 2, 3, 4, 5, 6]

# The left side of unpacking should be list or tuple.
*a, = numbers
# a = [1, 2, 3, 4, 5, 6]

*a, b = numbers
# a = [1, 2, 3, 4, 5]
# b = 6

a, *b, = numbers
# a = 1
```

Here, the `*a` and `*b` will do packing the remaining values again except the single unpacked values which are assigned other normal variables after unpacking the list or tuple. It is same concepts to packing for variadic arguments.

Conclusion

So far we've covered the **Asterisk(*)** of Python. It was interesting to be able to do various operations with one operator, and most of the those above are the basics for writing *Pythonic* code. Especially, the *"For using the variadic arguments"* is very important thing, but the python beginners often confused about this concept, so if you are a beginner of python, I would like you to know it better.

Next, I'll cover more interesting things about Python. Thank you.