2.3 Extended function argument syntax

When we talk about function arguments, we should recall the following facts:

- some functions can be invoked without arguments;
- functions may require a specific number of arguments with no exclusions; we have to pass a required number of arguments in an imposed order to follow function definition;
- functions might have already defined default values for some parameters, so we do not have to pass all arguments as missing arguments, complete with defaults; parameters with default values are presented as keyword parameters;
- we can pass arguments in any order if we are assigning keywords to all argument values, otherwise positional ones are the first ones on the arguments list.

Now let's get familiar with the functions that can accept any arbitrary number of positional arguments and keyword arguments.

So far we've been using a few such functions, but we haven't focused much on that fact.

The most basic example is a **print()** function:

```
print()
print(3)
print(1, 20, 10)
print('--', '++')
```

All presented invocations of the **print()** function end correctly, and the argument values are passed to the standard output.

Another interesting example can be presented with a list function:

```
a_list = list()
b_list = list(10, 20, 43, 54, 23, 23, 34, 23, 2)

print(a_list)
print(b_list)
```

Similarly to the **print()** function, the **list()** function invocation accepts an arbitrary number of arguments. In fact, there are many, many other functions that accept arbitrary numbers of parameters, and you can spot them easily when reading third-party code.

How is it possible for Python functions to deal with a variable number of arguments? Can we prepare such functions ourselves?

The answer is yes, but it may look strange at first glance: **args and **kwargs*. Don't worry, we'll tame them in a short moment.

These two special identifiers (named *args and **kwargs) should be put as the last two parameters in a function definition. Their names could be changed because it is just a convention to name them 'args' and 'kwargs', but it's more important to sustain the order of the parameters and leading asterisks.

Those two special parameters are responsible for handling any number of additional arguments (placed next after the expected arguments) passed to a called function:

- ***args** – refers to a tuple of all additional, not explicitly expected positional arguments, so arguments passed without keywords and passed next after the expected arguments. In other words, *args collects all unmatched positional arguments;

- ****kwargs** (keyword arguments) – refers to a dictionary of all unexpected arguments that were passed in the form of keyword=value pairs. Likewise, **kwargs collects all unmatched keyword arguments.

In Python, asterisks are used to denote that args and kwargs parameters are not ordinary parameters and should be unpacked, as they carry multiple items.

If you've ever programmed in the C or C++ languages, then you should remember that the asterisk character has another meaning (it denotes a pointer) which could be misleading for you.

Let's run the Python code presented in the right pane and study the output:

```
def combiner(a, b, *args, **kwargs):
    print(a, type(a))
    print(b, type(b))
    print(args, type(args))
    print(kwargs, type(kwargs))


combiner(10, '20', 40, 60, 30, argument1=50, argument2='66')
```

output

```
10 <class 'int'>
20 <class 'str'>
(40, 60, 30) <class 'tuple'>
{'argument1': 50, 'argument2': '66'} <class 'dict'>
```

As you can see, the function's definition is expecting two arguments, **a** and **b**, and the definition is prepared to handle any number of additional arguments. Moreover, all other positional arguments are available in a tuple (as you may remember, a tuple is a sequence type, because the order matters in this case). Similarly, all unexpected keyword parameters are available in a dictionary type parameter.

Now if you take a look at the built-in print() function definition, it becomes clear how this function can accept any number of arguments, and why there is an asterisk before one of the parameters:

```
def print(self, *args, sep=' ', end='\n', file=None):
```

Extended function argument syntax – forwarding arguments to other functions

When you want to forward arguments received by your very smart and universal function (defined with *args and **kwargs, of course) to another handy and smart function, you should do that in the following way:

```
def combiner(a, b, *args, **kwargs):
    super_combiner(*args, **kwargs)

def super_combiner(*my_args, **my_kwargs):
    print('my_args:', my_args)
    print('my_kwargs', my_kwargs)

combiner(10, '20', 40, 60, 30, argument1=50, argument2='66')
```

output

```
my_args: (40, 60, 30)
my_kwargs {'argument1': 50, 'argument2': '66'}
```

Pay attention how the **super_combiner()** function is called – it's called with genuine arguments so those arguments can be handled in the same way as they are handled by the combiner function.

If we remove the asterisks from the function call, then both tuple and dictionary would be captured by **my_args**, as it is supposed to handle all positional arguments (none of them is keyworded).

```
def combiner(a, b, *args, **kwargs):
    super_combiner(*args, **kwargs)

def super_combiner(*my_args, **my_kwargs):
    print('my_args:', my_args)
    print('my_kwargs', my_kwargs)

combiner(10, '20', 40, 60, 30, argument1=50, argument2='66')
```

output

```
my_args: ((40, 60, 30), {'argument1': 50, 'argument2': '66'})
my_kwargs {}
```

```
def combiner(a, b, *args, **kwargs):
    print(a, type(a))
    print(b, type(b))
    print(args, type(args))
    print(kwargs, type(kwargs))


combiner(10, '20', 40, 60, 30, argument1=50, argument2='66')
```

output

```
10 <class 'int'>
20 <class 'str'>
(40, 60, 30) <class 'tuple'>
{'argument1': 50, 'argument2': '66'} <class 'dict'>
```

The last example in this section shows how to combine *args, a key word, and **kwargs in one definition:

```
def combiner(a, b, *args, c=20, **kwargs):
    super_combiner(c, *args, **kwargs)
def super_combiner(my_c, *my_args, **my_kwargs):
    print('my_args:', my_args)
    print('my_c:', my_c)
    print('my_kwargs', my_kwargs)
combiner(1, '1', 1, 1, c=2, argument1=1, argument2='1')
```

output

```
my_args: (1, 1)
my_c: 2
my_kwargs {'argument1': 1, 'argument2': '1'}
```

As you can see, Python offers complex parameter handling:

- positional arguments (a,b) are distinguished from all other positional arguments (args)
- the keyword 'c' is distinguished from all other keyworded parameters.

Now that we know the purpose of special parameters, we can move on to decorators that make intensive use of those parameters.