*args and **kwargs All Versions

Examples

Using **kwargs when writing functions

You can define a function that takes an arbitrary number of keyword (named) arguments by using the double star `**` before a parameter name:

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

When calling the method, Python will construct a dictionary of all keyword arguments and make it available in the function body:

```
print_kwargs(a="two", b=3)
# prints: "{a: "two", b=3}"
```

Note that the **kwargs parameter in the function definition must always be the last parameter, and it will only match the arguments that were passed in after the previous ones.

```
def example(a, **kw):
    print kw

example(a=2, b=3, c=4) # => {'b': 3, 'c': 4}
```

Inside the function body, `kwargs` is manipulated in the same way as a dictionary; in order to access individual elements in `kwargs` you just loop through them as you would with a normal dictionary:

```
def print_kwargs(**kwargs):
    for key in kwargs:
        print("key = {0}, value = {1}".format(key, kwargs[key]))
```

Now, calling `print_kwargs(a="two", b=1)` shows the following output:

```
print_kwargs(a = "two", b = 1)
key = a, value = "two"
key = b, value = 1
```

Populating kwarg values with a dictionary

```
def foobar(foo=None, bar=None):
    return "{}{}".format(foo, bar)

values = {"foo": "foo", "bar": "bar"}

foobar(**values) # "foobar"
```

Using *args when writing functions

You can use the star * when writing a function to collect all positional (ie. unnamed) arguments in a tuple:

```
def print_args(farg, *args):
   print("formal arg: %s" % farg)
   for arg in args:
       print("another positional arg: %s" % arg)
```

Calling method:

```
print_args(1, "two", 3)
```

In that call, farg will be assigned as always, and the two others will be fed into the args tuple, in the order they were received.

Keyword-only and Keyword-required arguments

Python 3 allows you to define function arguments which can only be assigned by keyword, even without default values. This is done by using star * to consume additional positional parameters without setting the keyword parameters. All arguments after the * are keyword-only (i.e. non-positional) arguments. Note that if keyword-only arguments aren't given a default, they are still required when calling the function.

```
def print_args(arg1, *args, keyword_required, keyword_only=True):
    print("first positional arg: {}".format(arg1))
    for arg in args:
        print("another positional arg: {}".format(arg))
    print("keyword_required value: {}".format(keyword_required))
    print("keyword_only value: {}".format(keyword_only))

print(1, 2, 3, 4) # TypeError: print_args() missing 1 required keyword-only argument: 'keyword_r
print(1, 2, 3, keyword_required=4)
# first positional arg: 1
# another positional arg: 2
# another positional arg: 3
# keyword_required value: 4
# keyword_only value: True
```

**kwargs and default values

To use default values with **kwargs

```
def fun(**kwargs):
    print kwargs.get('value', 0)

fun()
# print 0
fun(value=1)
# print 1
```

Using **kwargs when calling functions

You can use a dictionary to assign values to the function's parameters; using parameters name as keys in the dictionary and the value of these arguments bound to each key:

```
def test_func(arg1, arg2, arg3): # Usual function with three arguments
   print("arg1: %s" % arg1)
   print("arg2: %s" % arg2)
   print("arg3: %s" % arg3)

# Note that dictionaries are unordered, so we can switch arg2 and arg3. Only the names matter.
kwargs = {"arg3": 3, "arg2": "two"}

# Bind the first argument (ie. arg1) to 1, and use the kwargs dictionary to bind the others
test_var_args_call(1, **kwargs)
```

Using *args when calling functions

The effect of using the `*` operator on an argument when calling a function is that of unpacking the list or a tuple argument

```
def print_args(arg1, arg2):
    print(str(arg1) + str(arg2))

a = [1,2]
b = tuple([3,4])

print_args(*a)
# 12
print_args(*b)
# 34
```

Note that the length of the starred argument need to be equal to the number of the function's arguments.

A common python idiom is to use the unpacking operator `*` with the `zip` function to reverse its effects:

```
a = [1,3,5,7,9]
b = [2,4,6,8,10]

zipped = zip(a,b)
```

```
# [(1,2), (3,4), (5,6), (7,8), (9,10)]

zip(*zipped)
# (1,3,5,7,9), (2,4,6,8,10)
```

Using *args when calling functions

A common use case for *args in a function definition is to delegate processing to either a wrapped or inherited function. A typical example might be in a class's __init__ method

```
class A(object):
    def __init__(self, b, c):
        self.y = b
        self.z = c

class B(A):
    def __init__(self, a, *args, **kwargs):
        super(B, self).__init__(*args, **kwargs)
        self.x = a
```

Here, the a parameter is processed by the child class after all other arguments (positional and keyword) are passed onto - and processed by - the base class.

For instance:

```
b = B(1, 2, 3)
b.x  # 1
b.y  # 2
b.z  # 3
```

What happens here is the class B __init__ function sees the arguments 1, 2, 3 . It knows it needs to take one positional argument (a), so it grabs the first argument passed in (1), so in the scope of the function a == 1 .

Next, it sees that it needs to take an arbitrary number of positional arguments (*args) so it takes the rest of the positional arguments passed in (1, 2) and stuffs them into *args . Now (in the scope of the function) args == [2, 3] .

Then, it calls class A 's __init__ function with *args . Python sees the * in front of args and "unpacks" the list into arguments. In this example, when class B 's __init__ function calls class A 's __init__ function, it will be passed the arguments 2, 3 (i.e. A(2, 3)).

Finally, it sets its own x property to the first positional argument a , which equals 1 .

Syntax

Parameters

Remarks

There a few things to note:

1. The names args and kwargs are used by convention, they are not a part of the language specification. Thus, these are equivalent:

   ```
   def func(*args, **kwargs):
       print(args)
       print(kwargs)
   ```

   ```
   def func(*a, **b):
       print(a)
       print(b)
   ```

2. You may not have more than one args or more than one kwargs parameters (however they are not required)

```
def func(*args1, *args2):
#   File "<stdin>", line 1
#     def test(*args1, *args2):
#                      ^
# SyntaxError: invalid syntax
```

```
def test(**kwargs1, **kwargs2):
#   File "<stdin>", line 1
#     def test(**kwargs1, **kwargs2):
#                       ^
# SyntaxError: invalid syntax
```

3. If any positional argument follow `*args`, they are keyword-only arguments that can only be passed by name. A single star may be used instead of `*args` to force values to be keyword arguments without providing a variadic parameter list. Keyword-only parameter lists are only available in Python 3.

```
def func(a, b, *args, x, y):
    print(a, b, args, x, y)

func(1, 2, 3, 4, x=5, y=6)
#>>> 1, 2, (3, 4), 5, 6
```

```
def func(a, b, *, x, y):
    print(a, b, x, y)

func(1, 2, x=5, y=6)
#>>> 1, 2, 5, 6
```

4. `**kwargs` must come last in the parameter list.

```
def test(**kwargs, *args):
#   File "<stdin>", line 1
#     def test(**kwargs, *args):
#                      ^
# SyntaxError: invalid syntax
```