关于参数（下）

可以接收一系列值的位置参数

如果你在定义参数的时候，在一个_位置参数_（Positional Arguments）前面标注了星号，`*`，那么，这个位置参数可以接收一系列值，在函数内部可以对这一系列值用 `for ... in ...` 循环进行逐一的处理。

带一个星号的参数，英文名称是 “Arbitrary Positional Arguments”，姑且翻译为 “随意的位置参数”。

还有带两个星号的参数，一会儿会讲到，英文名称是 “Arbitrary Keyword Arguments”，姑且翻译为 “随意的关键字参数”。

> 有些中文书籍把 “Arbitrary Positional Arguments” 翻译成 “可变位置参数”。事实上，在这样的地方，无论怎样的中文翻译都是令人觉得非常吃力的。前面的这个翻译还好了，我还见过把 “Arbitrary Keyword Arguments” 翻译成 “武断的关键字参数” 的 —— 我觉得这样的翻译肯定会使读者产生说不明道不白的疑惑。
>
> 所以，**入门之后就尽量只用英文**是个好策略。虽然刚开始有点吃力，但后面会很省心，很长寿 —— 是呀，少浪费时间、少浪费生命，其实就相当于更长寿了呀！

```
def say_hi(*names):
    for name in names:
        print(f'Hi, {name}!')
say_hi()
say_hi('ann')
say_hi('mike', 'john', 'zeo')
```

```
Hi, ann!
Hi, mike!
Hi, john!
Hi, zeo!
```

`say_hi()` 这一行没有任何输出。因为你在调用函数的时候，没有给它传递任何值，于是，在函数内部代码执行的时候，`name in names` 的值是 `False`，所以，`for` 循环内部的代码没有被执行。

在函数内部，是把 `names` 这个参数当作容器处理的 —— 否则也没办法用 `for ... in ...` 来处理。而在调用函数的时候，我们是可以将一个容器传递给函数的 Arbitrary Positional Arugments 的 —— 做法是，在调用函数的时候，在参数前面加上星号 `*`：

```
def say_hi(*names):
    for name in names:
        print(f'Hi, {name}!')

names = ('mike', 'john', 'zeo')
say_hi(*names)
```

```
Hi, mike!
Hi, john!
Hi, zeo!
```

实际上，因为以上的 `say_hi(*names)` 函数内部就是把接收到的参数当作容器处理的，于是，在调用这个函数的时候，向它传递任何容器都会被同样处理：

```
def say_hi(*names):
    for name in names:
        print(f'Hi, {name}!')

a_string = 'Python'
say_hi(*a_string)

a_range = range(10)
say_hi(*a_range)

a_list = list(range(10, 0, -1))
say_hi(*a_list)

a_dictionary = {'ann':2321, 'mike':8712, 'joe': 7610}
say_hi(*a_dictionary)
```

```
Hi, P!
Hi, y!
Hi, t!
Hi, h!
Hi, o!
Hi, n!
Hi, 0!
Hi, 1!
Hi, 2!
Hi, 3!
Hi, 4!
Hi, 5!
Hi, 6!
Hi, 7!
Hi, 8!
Hi, 9!
Hi, 10!
Hi, 9!
Hi, 8!
Hi, 7!
Hi, 6!
Hi, 5!
```

```
Hi, 4!
Hi, 3!
Hi, 2!
Hi, 1!
Hi, ann!
Hi, mike!
Hi, joe!
```

*在定义可以接收一系列值的位置参数时，建议在函数内部为该变量命名时总是用**复数***，因为函数内部，总是需要 for 循环去迭代元组中的元素，这样的时候，名称的复数形式对代码的可读性很有帮助 —— 注意以上程序第二行。以中文为母语的人，在这个细节上常常感觉“不堪重负” —— 因为中文的名词没有复数 —— 但必须习惯。（同样的道理，若是用拼音命名变量，就肯定是为将来挖坑……）

注意：一个函数中，可以接收一系列值的位置参数只能有一个；并且若是还有其它位置参数存在，那就必须把这个可以接收一系列值的位置参数排在所有其它位置参数之后。

```
def say_hi(greeting, *names):
    for name in names:
        print(f'{greeting}, {name.capitalize()}!')

say_hi('Hello', 'mike', 'john', 'zeo')
```

```
Hello, Mike!
Hello, John!
Hello, Zeo!
```

为函数的某些参数设定默认值

可以在定义函数的时候，为某些参数设定默认值，这些有默认值的参数，又被称作关键字参数（Keyword Arguments）。从这个函数的“用户”角度来看，这些设定了默认值的参数，就成了“可选参数”。

```
def say_hi(greeting, *names, capitalized=False):
    for name in names:
        if capitalized:
            name = name.capitalize()
        print(f'{greeting}, {name}!')

say_hi('Hello', 'mike', 'john', 'zeo')
say_hi('Hello', 'mike', 'john', 'zeo', capitalized=True)
```

```
Hello, mike!
Hello, john!
Hello, zeo!
```

```
Hello, Mike!
Hello, John!
Hello, Zeo!
```

可以接收一系列值的关键字参数

之前我们看到，可以设定一个位置参数（Positional Argument），接收一系列的值，被称作 “Arbitrary Positional Argument”；

同样地，我们也可以设定一个可以接收很多值的关键字参数（Arbitrary Keyword Argument）。

```
def say_hi(**names_greetings):
    for name, greeting in names_greetings.items():
        print(f'{greeting}, {name}!')

say_hi(mike='Hello', ann='Oh, my darling', john='Hi')
```

```
Hello, mike!
Oh, my darling, ann!
Hi, john!
```

既然在函数内部，我们在处理接收到的 Arbitrary Keyword Argument 时，用的是对字典的迭代方式，那么，在调用函数的时候，也可以直接使用字典的形式：

```
def say_hi(**names_greetings):
    for name, greeting in names_greetings.items():
        print(f'{greeting}, {name}!')

a_dictionary = {'mike':'Hello', 'ann':'Oh, my darling', 'john':'Hi'}
say_hi(**a_dictionary)

say_hi(**{'mike':'Hello', 'ann':'Oh, my darling', 'john':'Hi'})
```

```
Hello, mike!
Oh, my darling, ann!
Hi, john!
Hello, mike!
Oh, my darling, ann!
Hi, john!
```

至于在函数内部，你用什么样的迭代方式去处理这个字典，是你自己的选择：

```
def say_hi_2(**names_greetings):
    for name in names_greetings:
        print(f'{names_greetings[name]}, {name}!')
say_hi_2(mike='Hello', ann='Oh, my darling', john='Hi')
```

```
Hello, mike!
Oh, my darling, ann!
Hi, john!
```

函数定义时各种参数的排列顺序

在定义函数的时候，各种不同类型的参数应该按什么顺序摆放呢？对于之前写过的 `say_hi()` 这个函数，

```
def say_hi(greeting, *names, capitalized=False):
    for name in names:
        if capitalized:
            name = name.capitalize()
        print(f'{greeting}, {name}!')

say_hi('Hi', 'mike', 'john', 'zeo')
say_hi('Welcome', 'mike', 'john', 'zeo', capitalized=True)
```

```
Hi, mike!
Hi, john!
Hi, zeo!
Welcome, Mike!
Welcome, John!
Welcome, Zeo!
```

如果，你想给其中的 `greeting` 参数也设定个默认值怎么办？写成这样好像可以：

```
def say_hi(greeting='Hello', *names, capitalized=False):
    for name in names:
        if capitalized:
            name = name.capitalize()
        print(f'{greeting}, {name}!')

say_hi('Hi', 'mike', 'john', 'zeo')
say_hi('Welcome', 'mike', 'john', 'zeo', capitalized=True)
```

```
Hi, mike!
Hi, john!
Hi, zeo!

Welcome, Mike!
Welcome, John!
Welcome, Zeo!
```

但 greeting 这个参数虽然有默认值，可这个函数在被调用的时候，还是必须要给出这个参数，否则输出结果出乎你的想象：

```
def say_hi(greeting='Hello', *names, capitalized=False):
    for name in names:
        if capitalized:
            name = name.capitalize()
        print(f'{greeting}, {name}!')

say_hi('mike', 'john', 'zeo')
```

```
mike, john!
mike, zeo!
```

设定了默认值的 greeting，竟然不像你想象的那样是“可选参数”！所以，你得这样写：

```
def say_hi(*names, greeting='Hello', capitalized=False):
    for name in names:
        if capitalized:
            name = name.capitalize()
        print(f'{greeting}, {name}!')

say_hi('mike', 'john', 'zeo')
say_hi('mike', 'john', 'zeo', greeting='Hi')
```

```
Hello, mike!
Hello, john!
Hello, zeo!
Hi, mike!
Hi, john!
Hi, zeo!
```

这是因为函数被调用时，面对许多参数，Python 需要按照既定的规则（即，顺序）判定每个参数究竟是哪一类型的参数：

> **Order of Arguments**
>
> 1. Positional
> 2. Arbitrary Positional
> 3. Keyword
> 4. Arbitrary Keyword

所以，即便你在定义里写成

```
def say_hi(greeting='Hello', *names, capitalized=False):
    ...
```

在调用该函数的时候，无论你写的是

```
say_hi('Hi', 'mike', 'john', 'zeo')
```

还是

```
say_hi('mike', 'john', 'zeo')
```

Python 都会认为接收到的第一个值是 Positional Argument —— 因为在定义中，`greeting` 被放到了 Arbitrary Positional Arguments 之前。