

## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结

# 函数的参数



廖雪峰

资深软件开发工程师，业余马拉松选手。

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。

Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

## 位置参数

我们先写一个计算 $x^2$ 的函数：

```
def power(x):  
    return x * x
```

对于 `power(x)` 函数，参数 `x` 就是一个位置参数。

当我们调用 `power` 函数时，必须传入有且仅有的一个参数 `x`：

```
>>> power(5)  
25  
>>> power(15)  
225
```

现在，如果我们要计算 $x^3$ 怎么办？可以再定义一个 `power3` 函数，但是如果我们要计算 $x^4$ 、 $x^5$ .....怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把 `power(x)` 修改为 `power(x, n)`，用来计算 $x^n$ ，说干就干：

```
def power(x, n):  
    s = 1  
    while n > 0:  
        n = n - 1  
        s = s * x  
    return s
```

对于这个修改后的 `power(x, n)` 函数，可以计算任意 $n$ 次方：

```
>>> power(5, 2)  
25  
>>> power(5, 3)  
125
```

修改后的 `power(x, n)` 函数有两个参数：`x` 和 `n`，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数 `x` 和 `n`。

## 默认参数

新的 `power(x, n)` 函数定义没有问题，但是，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

```
>>> power(5)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: power() missing 1 required positional argument: 'n'
```

Python的错误信息很明确：调用函数 `power()` 缺少了一个位置参数 `n`。

这个时候，默认参数就排上用场了。由于我们经常计算 $x^2$ ，所以，完全可以把第二个参数 $n$ 的默认值设定为2：

```
def power(x, n=2):  
    s = 1
```

## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结

```
while n > 0:
    n = n - 1
    s = s * x
return s
```

这样，当我们调用 `power(5)` 时，相当于调用 `power(5, 2)`：

```
>>> power(5)
25
>>> power(5, 2)
25
```

而对于 `n > 2` 的其他情况，就必须明确地传入n，比如 `power(5, 3)`。

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；  
二是如何设置默认参数。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，我们写个一年级小学生注册的函数，需要传入 `name` 和 `gender` 两个参数：

```
def enroll(name, gender):
    print('name:', name)
    print('gender:', gender)
```

这样，调用 `enroll()` 函数只需要传入两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

```
def enroll(name, gender, age=6, city='Beijing'):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)
    print('city:', city)
```

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
age: 6
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

```
enroll('Bob', 'M', 7)
enroll('Adam', 'M', city='Tianjin')
```

可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用 `enroll('Bob', 'M', 7)`，意思是，除了 `name`，`gender` 这两个参数外，最后1个参数应用在参数 `age` 上，`city` 参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用 `enroll('Adam', 'M', city='Tianjin')`，意思是，`city` 参数用传进去的值，其他默认参数继续使用默认值。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个 `END` 再返回：

## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结

```
def add_end(L=[]):
    L.append('END')
    return L
```

当你正常调用时，结果似乎不错：

```
>>> add_end([1, 2, 3])
[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

```
>>> add_end()
['END']
```

但是，再次调用 `add_end()` 时，结果就不对了：

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

很多初学者很疑惑，默认参数是 `[]`，但是函数似乎每次都“记住了”上次添加了 `'END'` 后的list。

原因解释如下：

Python函数在定义的时候，默认参数 `L` 的值就被计算出来了，即 `[]`，因为默认参数 `L` 也是一个变量，它指向对象 `[]`，每次调用该函数，如果改变了 `L` 的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的 `[]` 了。

#### △ 特别注意

定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用 `None` 这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

```
>>> add_end()
['END']
>>> add_end()
['END']
```

为什么要设计 `str`、`None` 这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

## 可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

我们以数学题为例，给定一组数字 `a`, `b`, `c`....., 请计算  $a^2 + b^2 + c^2 + \dots$ 。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把 `a`, `b`, `c`.....作为一个list或tuple传进来，这样，函数可以定义如下：

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结

但是调用的时候，需要先组装出一个list或tuple：

```
>>> calc([1, 2, 3])
14
>>> calc((1, 3, 5, 7))
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

```
>>> calc(1, 2, 3)
14
>>> calc(1, 3, 5, 7)
84
```

所以，我们把函数的参数改为可变参数：

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

定义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个 `*` 号。在函数内部，参数 `numbers` 接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

```
>>> calc(1, 2)
5
>>> calc()
0
```

如果已经有一个list或者tuple，要调用一个可变参数怎么办？可以这样做：

```
>>> nums = [1, 2, 3]
>>> calc(nums[0], nums[1], nums[2])
14
```

这种写法当然是可行的，问题是太繁琐，所以Python允许你在list或tuple前面加一个 `*` 号，把list或tuple的元素变成可变参数传进去：

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
14
```

`*nums` 表示把 `nums` 这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

## 关键字参数

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。请看示例：

```
def person(name, age, **kw):
    print('name:', name, 'age:', age, 'other:', kw)
```

函数 `person` 除了必选参数 `name` 和 `age` 外，还接受关键字参数 `kw`。在调用该函数时，可以只传入必选参数：

```
>>> person('Michael', 30)
name: Michael age: 30 other: {}
```

也可以传入任意个数的关键字参数：

```
>>> person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}
>>> person('Adam', 45, gender='M', job='Engineer')
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

关键字参数有什么用？它可以扩展函数的功能。比如，在 `person` 函数里，我们保证能接收到 `name` 和 `age` 这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结

和可变参数类似，也可以先组装出一个dict，然后，把该dict转换为关键字参数传进去：

```
>>> extra = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, city=extra['city'], job=extra['job'])
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

当然，上面复杂的调用可以用简化的写法：

```
>>> extra = {'city': 'Beijing', 'job': 'Engineer'}
>>> person('Jack', 24, **extra)
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
```

`**extra` 表示把 `extra` 这个dict的所有key-value用关键字参数传入到函数的 `**kw` 参数，`kw` 将获得一个dict，注意 `kw` 获得的dict是 `extra` 的一份拷贝，对 `kw` 的改动不会影响到函数外的 `extra`。

## 命名关键字参数

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过 `kw` 检查。

仍以 `person()` 函数为例，我们希望检查是否有 `city` 和 `job` 参数：

```
def person(name, age, **kw):
    if 'city' in kw:
        # 有city参数
        pass
    if 'job' in kw:
        # 有job参数
        pass
    print('name:', name, 'age:', age, 'other:', kw)
```

但是调用者仍可以传入不受限制的关键字参数：

```
>>> person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123456)
```

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收 `city` 和 `job` 作为关键字参数。这种方式定义的函数如下：

```
def person(name, age, *, city, job):
    print(name, age, city, job)
```

和关键字参数 `**kw` 不同，命名关键字参数需要一个特殊分隔符 `*`，`*` 后面的参数被视为命名关键字参数。

调用方式如下：

```
>>> person('Jack', 24, city='Beijing', job='Engineer')
Jack 24 Beijing Engineer
```

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符 `*` 了：

```
def person(name, age, *args, city, job):
    print(name, age, args, city, job)
```

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

```
>>> person('Jack', 24, 'Beijing', 'Engineer')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: person() missing 2 required keyword-only arguments: 'city' and 'job'
```

由于调用时缺少参数名 `city` 和 `job`，Python解释器把前两个参数视为位置参数，后两个参数传给 `*args`，但缺少命名关键字参数导致报错。

命名关键字参数可以有缺省值，从而简化调用：

```
def person(name, age, *, city='Beijing', job):
    print(name, age, city, job)
```

由于命名关键字参数 `city` 具有默认值，调用时，可不传入 `city` 参数：

## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结

```
>>> person('Jack', 24, job='Engineer')
Jack 24 Beijing Engineer
```

使用命名关键字参数时，要特别注意，如果没有可变参数，就必须加一个 `*` 作为特殊分隔符。如果缺少 `*`，Python 解释器将无法识别位置参数和命名关键字参数：

```
def person(name, age, city, job):
    # 缺少 *, city和job被视为位置参数
    pass
```

## 参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数。

比如定义一个函数，包含上述若干种参数：

```
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

```
>>> f1(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> f1(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> f1(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> f1(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
>>> f2(1, 2, d=99, ext=None)
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

最神奇的是通过一个tuple和dict，你也可以调用上述函数：

```
>>> args = (1, 2, 3, 4)
>>> kw = {'d': 99, 'x': '#'}
>>> f1(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}
>>> args = (1, 2, 3)
>>> kw = {'d': 88, 'x': '#'}
>>> f2(*args, **kw)
a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

所以，对于任意函数，都可以通过类似 `func(*args, **kw)` 的形式调用它，无论它的参数是如何定义的。

### 💡 提示

虽然可以组合多达5种参数，但不要同时使用太多的组合，否则函数接口的可理解性很差。

## 练习

以下函数允许计算两个数的乘积，请稍加改造，变成可接收一个或多个数并计算乘积：

```
def mul(x, y):
    return x * y

# 测试
print('mul(5) =', mul(5))
print('mul(5, 6) =', mul(5, 6))
print('mul(5, 6, 7) =', mul(5, 6, 7))
print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9))
if mul(5) != 5:
    print('mul(5)测试失败!')
elif mul(5, 6) != 30:
    print('mul(5, 6)测试失败!')
```



## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结

```
elif mul(5, 6, 7) != 210:
    print('mul(5, 6, 7)测试失败!')
elif mul(5, 6, 7, 9) != 1890:
    print('mul(5, 6, 7, 9)测试失败!')
else:
    try:
        mul()
        print('mul()测试失败!')
    except TypeError:
        print('测试成功!')
```

## 参考源码

[var\\_args.py](#)

[kw\\_args.py](#)

## 小结

Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

默认参数一定要用不可变对象，如果是可变对象，程序运行时会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

`*args` 是可变参数，args接收的是一个tuple；

`**kw` 是关键字参数，kw接收的是一个dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：`func(1, 2, 3)`，又可以先组装list或tuple，再通过 `*args` 传入：`func(*(1, 2, 3))`；

关键字参数既可以直接传入：`func(a=1, b=2)`，又可以先组装dict，再通过 `**kw` 传入：`func(**{'a': 1, 'b': 2})`。

使用 `*args` 和 `**kw` 是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

定义命名的关键字参数在没有可变参数的情况下不要忘了写分隔符 `*`，否则定义的将是位置参数。

《 定义函数

递归函数 》



## Comments

Comments loaded. To post a comment, please [Sign In](#)



仰望星空 @ 2025/12/31 09:16:02

```
def mul(*nums): if not nums: raise TypeError res = 1 for num in nums: res *= num return res
```

## 测试

```
print('mul(5) =', mul(5)) print('mul(5, 6) =', mul(5, 6)) print('mul(5, 6, 7) =', mul(5, 6, 7)) print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9)) if mul(5) != 5: print('mul(5)测试失败!') elif mul(5, 6) != 30: print('mul(5, 6)测试失
```

## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结

败!) elif mul(5, 6, 7) != 210: print('mul(5, 6, 7)测试失败!') elif mul(5, 6, 7, 9) != 1890: print('mul(5, 6, 7, 9)测试失败!') else: try: mul() print('mul()测试失败!') except TypeError: print('测试成功!')



殇+魂 @ 2025/12/27 04:07:37

```
def mul(x, *y): num = 1 list1 = list(y) list1.append(x) for i in list1: num = num * i return num
```

## 测试

```
print('mul(5) =', mul(5)) print('mul(5, 6) =', mul(5, 6)) print('mul(5, 6, 7) =', mul(5, 6, 7)) print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9)) if mul(5) != 5: print('mul(5)测试失败!') elif mul(5, 6) != 30: print('mul(5, 6)测试失败!') elif mul(5, 6, 7) != 210: print('mul(5, 6, 7)测试失败!') elif mul(5, 6, 7, 9) != 1890: print('mul(5, 6, 7, 9)测试失败!') else: try: mul() print('mul()测试失败!') except TypeError: print('测试成功!')
```



鏡華水月 @ 2025/12/21 07:27:45

```
def mul(x,*args):
    result = x
    for n in args:
        result = result * n
    return result

print('mul(5) =', mul(5))
print('mul(5, 6) =', mul(5, 6))
print('mul(5, 6, 7) =', mul(5, 6, 7))
print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9))
if mul(5) != 5:
    print('mul(5)测试失败!')
elif mul(5, 6) != 30:
    print('mul(5, 6)测试失败!')
elif mul(5, 6, 7) != 210:
    print('mul(5, 6, 7)测试失败!')
```

[Read More ▼](#)



lv @ 2025/12/19 00:50:02

```
def mul(x, *args):
    for n in args:
        x *= n
    return x
```



Victoria @ 2025/12/17 09:41:15

```
def mul(*x): s = 1 for i in x: s = s * i return s print(mul(5)) print(mul(5,6)) print(mul(5,6,7))
```



轨迹 ^0^ @ 2025/12/15 06:09:17

```
def mul(x, y=1, *arg):
    if len(arg) == 0:
        return x * y
    else:
        result = x * y
        for n in arg:
            result *= n
        return result

# 测试
print('mul(5) =', mul(5))
print('mul(5, 6) =', mul(5, 6))
print('mul(5, 6, 7) =', mul(5, 6, 7))
print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9))
if mul(5) != 5:
    print('mul(5)测试失败!')
```

[Read More ▼](#)



## PYTHON教程

1. 简介
2. Python历史
3. 安装Python
  - 3.1. Python解释器
4. 第一个Python程序
  - 4.1. 使用文本编辑器
  - 4.2. 输入和输出
5. Python基础
  - 5.1. 数据类型和变量
  - 5.2. 字符串和编码
  - 5.3. 使用list和tuple
  - 5.4. 条件判断
  - 5.5. 模式匹配
  - 5.6. 循环
  - 5.7. 使用dict和set
6. 函数
  - 6.1. 调用函数
  - 6.2. 定义函数
  - 6.3. 函数的参数
  - 6.4. 递归函数
7. 高级特性
  - 7.1. 切片
  - 7.2. 迭代
  - 7.3. 列表生成式
  - 7.4. 生成器
  - 7.5. 迭代器
8. 函数式编程
9. 模块
10. 面向对象编程
11. 面向对象高级编程
12. 错误、调试和测试
13. IO编程
14. 进程和线程
15. 正则表达式
16. 常用内建模块
17. 常用第三方模块
18. 图形界面
19. 网络编程
20. 电子邮件
21. 访问数据库
22. Web开发
23. 异步IO
24. FAQ
25. 期末总结



某不科学动物 @ 2025/12/7 02:08:35

```
def mul(*num):
    if len(num) == 0:
        raise TypeError("至少需要一个参数")

    s=1
    for n in num:
        s=s*n
    return s
print('mul(5) =', mul(5))
print('mul(5, 6) =', mul(5, 6))
print('mul(5, 6, 7) =', mul(5, 6, 7))
print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9))
if mul(5) != 5:
    print('mul(5)测试失败!')
elif mul(5, 6) != 30:
    print('mul(5, 6)测试失败!')
elif mul(5, 6, 7) != 210:
```

[Read More ▼](#)



啊, 你以为我傻 @ 2025/12/4 05:35:51

```
def mul(*numbers):
    if not numbers:
        raise TypeError
    s = 1
    for i in numbers:
        s = s * i
    return s
```



Floris @ 2025/11/26 03:04:52

```
def mul(x, *y):
    for i in y:
        x *= i
    return x

print(mul(10,0,1,2,3))
```



追梦路上 @ 2025/11/25 08:44:04

```
def mul(*numbers):
    sum = 1
    for i in numbers:
        if isinstance(i,(int,float)):
            sum *= i
        else:
            raise TypeError(f'{i} not is numbers')
    return sum
r = mul(5,6,"hhhh",9)
print(r)
```



Hypersomnia @ 2025/11/24 01:32:56

```
def mul(*numbers):
    if not numbers:
        raise TypeError
    s = 1
    for n in numbers:
        s = s * n
    return s
```



易 @ 2025/11/20 03:06:51

```
def mul(x,*y):
    if y.count == 0:
        return x
    else:
        for y1 in y:
            x = x*y1
        return x
```



Meow @ 2025/11/13 02:45:26

```
def mul(*num):
    if not num:
        raise TypeError
```

## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结

```
result = 1
for n in num:
    result *= n
return result

# 测试
assert mul(0) == 0
assert mul(5) == 5
assert mul(5, 6) == 30
assert mul(5, 6, 7) == 210
assert mul(5, 6, 7, 9) == 1890
```

[Read More](#) ▼



Moon @ 2025/11/12 02:35:34

错的地方：1、return应该和for齐平，for循环完成之后返回值 2、raise TypeError报错，return TypeError不对 都是错在小问题上

```
def mul(*nums):
    if len(nums)==0:
        raise TypeError
    s=1
    for i in nums:
        s=s*i
    return s
print('mul(5) =', mul(5))
print('mul(5, 6) =', mul(5, 6))
print('mul(5, 6, 7) =', mul(5, 6, 7))
print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9))
if mul(5) != 5:
    print('mul(5)测试失败!')
```

[Read More](#) ▼



方智 @ 2025/11/4 03:04:46

```
def mul(x, y = 1, *args): a = 1 for i in args : a = i * a
```

```
return x * y * a
```

## 测试

```
print('mul(5) =', mul(5)) print('mul(5, 6) =', mul(5, 6)) print('mul(5, 6, 7) =', mul(5, 6, 7)) print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9)) if mul(5) != 5: print('mul(5)测试失败!') elif mul(5, 6) != 30: print('mul(5, 6)测试失败!') elif mul(5, 6, 7) != 210: print('mul(5, 6, 7)测试失败!') elif mul(5, 6, 7, 9) != 1890: print('mul(5, 6, 7, 9)测试失败!') else: try: mul() print('mul()测试失败!') except TypeError: print('测试成功!')
```



行简 @ 2025/10/29 04:45:36

```
def add_end(L = None): ... if L is None: ... L = [] ... L.append('END') ... return L
```



圆圈 @ 2025/10/28 23:29:57

```
def mul(*nums): m = 1 if len(nums) == 0: raise TypeError('至少需要一个参数!') else: for n in nums: m = m * n return m
```

```
print('mul(5) =', mul(5)) print('mul(5, 6) =', mul(5, 6)) print('mul(5, 6, 7) =', mul(5, 6, 7)) print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9))
```

```
if mul(5) != 5: print('mul(5)测试失败!') elif mul(5, 6) != 30: print('mul(5, 6)测试失败!') elif mul(5, 6, 7) != 210: print('mul(5, 6, 7)测试失败!') elif mul(5, 6, 7, 9) != 1890: print('mul(5, 6, 7, 9)测试失败!') else: try: mul() print('mul()测试失败!') except TypeError: print('测试成功!')
```

## PYTHON教程

### 1. 简介

### 2. Python历史

### 3. 安装Python

#### 3.1. Python解释器

### 4. 第一个Python程序

#### 4.1. 使用文本编辑器

#### 4.2. 输入和输出

### 5. Python基础

#### 5.1. 数据类型和变量

#### 5.2. 字符串和编码

#### 5.3. 使用list和tuple

#### 5.4. 条件判断

#### 5.5. 模式匹配

#### 5.6. 循环

#### 5.7. 使用dict和set

### 6. 函数

#### 6.1. 调用函数

#### 6.2. 定义函数

#### 6.3. 函数的参数

#### 6.4. 递归函数

### 7. 高级特性

#### 7.1. 切片

#### 7.2. 迭代

#### 7.3. 列表生成式

#### 7.4. 生成器

#### 7.5. 迭代器

### 8. 函数式编程

### 9. 模块

### 10. 面向对象编程

### 11. 面向对象高级编程

### 12. 错误、调试和测试

### 13. IO编程

### 14. 进程和线程

### 15. 正则表达式

### 16. 常用内建模块

### 17. 常用第三方模块

### 18. 图形界面

### 19. 网络编程

### 20. 电子邮件

### 21. 访问数据库

### 22. Web开发

### 23. 异步IO

### 24. FAQ

### 25. 期末总结



Sfz @ 2025/10/24 04:36:34

```
def mul(*numbers): sum = 1 if len(numbers)==0: raise TypeError("You should give at least 1 number!\n") for n in numbers: sum = sum * n return sum
```

## 测试

```
print('mul(5) =', mul(5)) print('mul(5, 6) =', mul(5, 6)) print('mul(5, 6, 7) =', mul(5, 6, 7)) print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9)) if mul(5) != 5: print('mul(5)测试失败!') elif mul(5, 6) != 30: print('mul(5, 6)测试失败!') elif mul(5, 6, 7) != 210: print('mul(5, 6, 7)测试失败!') elif mul(5, 6, 7, 9) != 1890: print('mul(5, 6, 7, 9)测试失败!') else: try: mul() print('mul()测试失败!') except TypeError: print('测试成功!')
```



坠落机尾 @ 2025/10/21 04:25:49

```
def mul(x, y=1,*z): if len(z) == 0: mul_value = x*y else: mul_value = x*y for i in z: mul_value = mul_value*i return mul_value
```



是季不是鸡 @ 2025/10/18 22:47:46

# 数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数

```
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)
```

```
def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

```
def mul(*number : int) :
    if number == () :
        raise TypeError
    result = 1
    for i in number :

        result = result * i
    return result
```

```
# 测试
print('mul(5) =', mul(5))
print('mul(5, 6) =', mul(5, 6))
print('mul(5, 6, 7) =', mul(5, 6, 7))
print('mul(5, 6, 7, 9) =', mul(5, 6, 7, 9))
if mul(5) != 5:
    print('mul(5)测试失败!')
elif mul(5, 6) != 30:
    print('mul(5, 6)测试失败!')
elif mul(5, 6, 7) != 210:
    print('mul(5, 6, 7)测试失败!')
elif mul(5, 6, 7, 9) != 1890:
    print('mul(5, 6, 7, 9)测试失败!')
else:
    try:
        mul()
        print('mul()测试失败!')
    except TypeError:
        print('测试成功!')
```

Collapse ▲