缩进规则 缩进表示嵌套关系

变量：由非数字开头的字母、数字、下划线组成，它的内容可以是数值、字符串、列表、元组和字典。

数值，就是数字

字符串，被引号括起来的内容

List 列表[]

Len()求长度

下标为负值表示倒数第几个元素

List.append(x)表示追加元素x到末尾

删除末尾元素pop()

删除指定索引位置元素pop(i)

C = [1,2,3,4,5,6,7]

用列表名[起：止]表示切片，从列表中切出相应的元素 前闭后开

C[0:2]切出[1,2]

C[:]切出[1,2,3,4,5,6,7]

用列表名[起：止：步长] 带步长的切片，步长有方向

C = [1,2,3,4,5,6,7]

切出[5,4,3,2]用c[4:0:-1]

切出[5,4,3,2,1]用c[4: :-1]

切出[6,4,2]用c[-2: :-2]从倒数第二个开始一直且到头，步长为-2

修改：列表名[索引号] = 新值

删除：del列表名[索引号]

插入：列表名.insert(插入位置索引号，新元素)

Tuple 元组()

元组是誓言，一旦定义不能改变

注意(1)既可以表示tuple也可以表示数学公式中的小括号

所以只有一个元素时tuple必须加一个,

如t = (1,)

tuple不变指的是tuple中每个元素指向永远不变，但如果tuple中有list元素，指向一个list虽然不能改成指向其他对象，但是指向的这个list本身可变

条件语句

if <条件判断1>:  
 <执行1>  
elif <条件判断2>:  
 <执行2>  
elif <条件判断3>:  
 <执行3>  
else:  
 <执行4>

例：

age = 3  
**if** age >= 18:  
 print('adult')  
**elif** age >= 6:  
 print('teenager')  
**else**:  
 print('kid')

输入(input函数)和类型转换(int())

s = input('birth: ')  
birth = int(s)  
**if** birth < 2000:  
 **print**('00前')  
**else**:  
 **print**('00后')

循环

1.for...in循环

names = ['Michael', 'Bob', 'Tracy']  
**for** name **in** names:  
 print(name)

\*Python提供一个range()函数，可以生成一个整数序列，再通过list()函数可以转换为list。比如range(5)生成的序列是从0开始小于5的整数

2.while循环

sum = 0  
n = 99  
**while** n > 0:  
 sum = sum + n  
 n = n - 2  
**print**(sum)

break/continue

n = 1  
**while** n <= 100:  
 **if** n > 10: *# 当n = 11时，条件满足，执行break语句*  
 **break** *# break语句会结束当前循环*  
 **print**(n)  
 n = n + 1  
**print**('END')

n = 0  
**while** n < 10:  
 n = n + 1  
 **if** n % 2 == 0: *# 如果n是偶数，执行continue语句*  
 **continue** *# continue语句会直接继续下一轮循环，后续的print()语句不会执行*  
 **print**(n)

字典（dict）

在其他语言中也称map（key-value）

names = ['Michael', 'Bob', 'Tracy']  
scores = [95, 75, 85]

d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}

要避免key不存在的错误

1.通过in判断key是否存在：

>>> 'Thomas' **in** d  
False

2.通过dict提供的get()方法，如果key不存在，可以返回None，或者自己指定的value：

>>> d.get('Thomas')  
>>> d.get('Thomas', -1)  
-1

要删除一个key，用pop(key)方法，对应的value也会从dict中删除

Set(集合）

要创建一个set，需要提供一个list作为输入集合：

>>> s = set([1, 2, 3])  
>>> s  
{1, 2, 3}

注意，传入的参数[1, 2, 3]是一个list，而显示的{1, 2, 3}只是告诉你这个set内部有1，2，3这3个元素，显示的顺序也不表示set是有序的。重复元素在set中自动被过滤

通过add(key)方法可以添加元素到set中，可以重复添加，但不会有效果

通过remove(key)方法可以删除元素：

set可以看成数学意义上的无序和无重复元素的集合，因此，两个set可以做数学意义上的交集、并集等操作：

>>> s1 = set([1, 2, 3])  
>>> s2 = set([2, 3, 4])  
>>> s1 & s2  
{2, 3}  
>>> s1 | s2  
{1, 2, 3, 4}

不可变对象

对于不可变对象，比如str，对str进行操作

>>> a = 'abc'  
>>> b = a.replace('a', 'A')  
>>> b  
'Abc'  
>>> a  
'abc'

函数

定义一个函数要使用def语句，依次写出函数名、括号、括号中的参数和冒号:，然后，在缩进块中编写函数体，函数的返回值用return语句返回

例

def my\_abs(x):

if x >= 0:

return x

else:

return -x

如果你已经把my\_abs()的函数定义保存为abstest.py文件了，那么，可以在该文件的当前目录下启动Python解释器，用from abstest import my\_abs来导入my\_abs()函数，注意abstest是文件名（不含.py扩展名）：

空函数

**def nop**():  
 **pass**

pass语句什么都不做，那有什么用？实际上pass可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个pass，让代码能运行起来。

对参数类型做检查，只允许整数和浮点数类型的参数。数据类型检查可以用内置函数isinstance()实现

例：

**def my\_abs**(x):  
 **if not** isinstance(x, (int, float)):  
 **raise** TypeError('bad operand type')  
 **if** x >= 0:  
 **return** x  
 **else**:  
 **return** -x

返回多个值

例

**import** math

**def move**(x, y, step, angle=0):  
 nx = x + step \* math.cos(angle)  
 ny = y - step \* math.sin(angle)  
 **return** nx, ny

但其实这只是一种假象，Python函数返回的仍然是单一值：

返回值是一个tuple，但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个tuple，但写起来更方便。

 定义默认参数要牢记一点：默认参数必须指向不变对象！

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。

误用例

**def add\_end**(L=[]):  
 L.append('END')  
 **return** L

>>> add\_end()  
['END']

>>> add\_end()  
['END', 'END']  
>>> add\_end()  
['END', 'END', 'END']

正确例

**def add\_end**(L=None):  
 **if** L **is** None:  
 L = []  
 L.append('END')  
 **return** L

可变参数

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

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)  
14

关键字参数

函数的参数

阅读: 604999

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x2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x3怎么办？可以再定义一个power3函数，但是如果要计算x4、x5……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xn，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x2，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  
 **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：

**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。

 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。

我们以数学题为例子，给定一组数字a，b，c……，请计算a2 + b2 + c2 + ……。

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

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'}

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'}

命名关键字参数

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)

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() takes 2 positional arguments but 4 were given

函数的参数

阅读: 604999

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x2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x3怎么办？可以再定义一个power3函数，但是如果要计算x4、x5……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xn，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x2，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  
 **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：

**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。

 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。

我们以数学题为例子，给定一组数字a，b，c……，请计算a2 + b2 + c2 + ……。

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

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。

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() takes 2 positional arguments but 4 were given

由于调用时缺少参数名city和job，Python解释器把这4个参数均视为位置参数，但person()函数仅接受2个位置参数。

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：

>>> 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。

 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。

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。

为什么python不需要swap

<https://blog.csdn.net/ccat/article/details/8349>