函数

我们知道圆的面积计算公式为：$S = \pi r^2$

当我们知道半径`r`的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

```
r1 = 12.34
r2 = 9.08
r3 = 73.1
s1 = 3.14 * r1 * r1
s2 = 3.14 * r2 * r2
s3 = 3.14 * r3 * r3
```

当代码出现有规律的重复的时候，你就需要当心了，每次写`3.14 * x * x`不仅很麻烦，而且，如果要把`3.14`改成`3.14159265359`的时候，得全部替换。

有了函数，我们就不再每次写`s = 3.14 * x * x`，而是写成更有意义的函数调用`s = area_of_circle(x)`，而函数`area_of_circle`本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，Python也不例外。Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

抽象

抽象是数学中非常常见的概念。举个例子：

计算数列的和，比如：`1 + 2 + 3 + ... + 100`，写起来十分不方便，于是数学家发明了求和符号∑，可以把`1 + 2 + 3 + ... + 100`记作：$\sum_{n=1}^{100} n$

这种抽象记法非常强大，因为我们看到∑就可以理解成求和，而不是还原成低级的加法运算。

而且，这种抽象记法是可扩展的,比如：$\sum_{n=1}^{100} 2n + 1$

还原成加法运算就变成了：

`(1 x 1 + 1) + (2 x 2 + 1) + (3 x 3 + 1) + ... + (100 x 100 + 1)`

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。

调用函数

Python内置了很多有用的函数，我们可以直接调用。

要调用一个函数，需要知道函数的名称和参数，比如求绝对值的函数`abs`，只有一个参数。可以直接从Python的官方网站查看文档：

http://docs.python.org/3/library/functions.html#abs

也可以在交互式命令行通过`help(abs)`查看`abs`函数的帮助信息。

调用`abs`函数：

```
>>> abs(100)
100
>>> abs(-20)
20
>>> abs(12.34)
12.34
```

调用函数的时候，如果传入的参数数量不对，会报`TypeError`的错误，并且Python会明确地告诉你：`abs()`有且仅有1个参数，但给出了两个：

```
>>> abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
```

如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报`TypeError`的错误，并且给出错误信息：`str`是错误的参数类型：

```
>>> abs('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

而`max`函数`max()`可以接收任意多个参数，并返回最大的那个：

```
>>> max(1, 2)
2
>>> max(2, 3, 1, -5)
3
```

数据类型转换

Python内置的常用函数还包括数据类型转换函数，比如`int()`函数可以把其他数据类型转换为整数：

```
>>> int('123')
123
>>> int(12.34)
12
>>> float('12.34')
12.34
>>> str(1.23)
'1.23'
>>> str(100)
'100'
>>> bool(1)
True
>>> bool('')
False
```

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”：

```
>>> a = abs # 变量a指向abs函数
>>> a(-1) # 所以也可以通过a调用abs函数
1
```

练习

请利用Python内置的`hex()`函数把一个整数转换成十六进制表示的字符串：

```
# -*- coding: utf-8 -*-
n1 = 255
n2 = 1000
```

小结

- 调用Python的函数，需要根据函数定义，传入正确的参数。如果函数调用出错，一定要学会看错误信息，所以英文很重要！

定义函数

在Python中，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。

我们以自定义一个求绝对值的`my_abs`函数为例：

```
# -*- coding: utf-8 -*-
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

```
print(my_abs(-99))
```

请自行测试并调用`my_abs`看看返回结果是否正确。

请注意，函数体内部的语句在执行时，一旦执行到`return`时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有`return`语句，函数执行完毕后也会返回结果，只是结果为`None`。`return None`可以简写为`return`。

在Python交互环境中定义函数时，注意Python会出现`...`的提示。函数定义结束后需要按两次回车重新回到`>>>`提示符下：

```
┌──────────────────────────────────────────────────────┐
|Command Prompt - python                        - □ x |
├──────────────────────────────────────────────────────┤
|>>> def my_abs(x):                                    |
|...     if x >= 0:                                    |
|...         return x                                  |
|...     else:                                         |
|...         return -x                                 |
|...                                                   |
|>>> my_abs(-9)                                        |
```

```
|9                                                          |
|>>> _                                                      |
|                                                           |
|                                                           |
└───────────────────────────────────────────────────────────┘
```

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）：

```
┌───────────────────────────────────────────────────────────┐
|Command Prompt - python                          - □ x     |
├───────────────────────────────────────────────────────────┤
|>>> from abstest import my_abs                             |
|>>> my_abs(-9)                                            |
|9                                                          |
|>>> _                                                      |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
└───────────────────────────────────────────────────────────┘
```

`import`的用法在后续模块一节中会详细介绍。

空函数

如果想定义一个什么事也不做的空函数，可以用`pass`语句：

```
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。

`pass`还可以用在其他语句里，比如：

```
if age >= 18:
    pass
```

缺少了`pass`，代码运行就会有语法错误。

参数检查

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出`TypeError`：

```
>>> my_abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: my_abs() takes 1 positional argument but 2 were given
```

但是如果参数类型不对，Python解释器就无法帮我们检查。试试`my_abs`和内置函数`abs`的差别：

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in my_abs
TypeError: unorderable types: str() >= int()
>>> abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

当传入了不恰当的参数时，内置函数`abs`会检查出参数错误，而我们定义的`my_abs`没有参数检查，会导致`if`语句出错，出错信息和`abs`不一样。所以，这个函数定义不够完善。

让我们修改一下`my_abs`的定义，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：

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return -x
```

添加了参数检查后，如果传入错误的参数类型，函数就可以抛出一个错误：

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in my_abs
TypeError: bad operand type
```

错误和异常处理将在后续讲到。

返回多个值

函数可以返回多个值吗？答案是肯定的。

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的坐标：

```
import math

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

`import math`语句表示导入`math`包，并允许后续代码引用`math`包里的`sin`、`cos`等函数。

然后，我们就可以同时获得返回值：

```
>>> x, y = move(100, 100, 60, math.pi / 6)
>>> print(x, y)
151.96152422706632 70.0
```

但其实这只是一种假象，Python函数返回的仍然是单一值：

```
>>> r = move(100, 100, 60, math.pi / 6)
>>> print(r)
(151.96152422706632, 70.0)
```

原来返回值是一个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。

小结

- 定义函数时，需要确定函数名和参数个数；
- 如果有必要，可以先对参数的数据类型做检查；
- 函数体内部可以用`return`随时返回函数结果；
- 函数执行完毕也没有`return`语句时，自动`return None`。
- 函数可以同时返回多个值，但其实就是一个tuple。

练习

请定义一个函数`quadratic(a, b, c)`，接收3个参数，返回一元二次方程 $ax^2 + bx + c = 0$的两个解。

提示：

一元二次方程的求根公式为：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

计算平方根可以调用`math.sqrt()`函数：

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

```
# -*- coding: utf-8 -*-
import math
def quadratic(a, b, c):
    pass
```

```
# 测试:
print('quadratic(2, 3, 1) =', quadratic(2, 3, 1))
print('quadratic(1, 3, -4) =', quadratic(1, 3, -4))

if quadratic(2, 3, 1) != (-0.5, -1.0):
    print('测试失败')
elif quadratic(1, 3, -4) != (1.0, -4.0):
    print('测试失败')
else:
    print('测试成功')
```

函数的参数

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。

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：

```
# 测试
print('product(5) =', product(5))
print('product(5, 6) =', product(5, 6))
print('product(5, 6, 7) =', product(5, 6, 7))
print('product(5, 6, 7, 9) =', product(5, 6, 7, 9))
if product(5) != 5:
    print('测试失败!')
elif product(5, 6) != 30:
    print('测试失败!')
elif product(5, 6, 7) != 210:
    print('测试失败!')
elif product(5, 6, 7, 9) != 1890:
    print('测试失败!')
else:
```

```
    try:
        product()
        print('测试失败!')
    except TypeError:
        print('测试成功!')
```

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。

递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。
举个例子，我们来计算阶乘`n! = 1 x 2 x 3 x ... x n`，用函数`fact(n)`表示，可以看出：
`fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n`

所以，`fact(n)`可以表示为`n x fact(n-1)`，只有n=1时需要特殊处理。

于是，`fact(n)`用递归的方式写出来就是：

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

```
>>> fact(1)
1
>>> fact(5)
120
>>> fact(100)
93326215443944152681699238856266700490715968264381621468592963895217599993229915
608941463976156518286253697920827223758251185210916864000000000000000000000000
```

如果我们计算`fact(5)`，可以根据函数定义看到计算过程如下：

```
===> fact(5)
===> 5 * fact(4)
===> 5 * (4 * fact(3))
===> 5 * (4 * (3 * fact(2)))
===> 5 * (4 * (3 * (2 * fact(1))))
===> 5 * (4 * (3 * (2 * 1)))
===> 5 * (4 * (3 * 2))
===> 5 * (4 * 6)
===> 5 * 24
===> 120
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试`fact(1000)`：

```
>>> fact(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in fact
  ...
  File "<stdin>", line 4, in fact
RuntimeError: maximum recursion depth exceeded in comparison
```

解决递归调用栈溢出的方法是通过**尾递归**优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的`fact(n)`函数由于`return n * fact(n - 1)`引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到，`return fact_iter(num - 1, num * product)`仅返回递归函数本身，`num - 1`和`num * product`在函数调用前就会被计算，不影响函数调用。

`fact(5)`对应的`fact_iter(5, 1)`的调用如下：

```
===> fact_iter(5, 1)
===> fact_iter(4, 5)
===> fact_iter(3, 20)
===> fact_iter(2, 60)
===> fact_iter(1, 120)
===> 120
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的`fact(n)`函数改成尾递归方式，也会导致栈溢出。

小结

- 使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。
- 针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。
- Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

练习

汉诺塔的移动可以用递归函数非常简单地实现。

请编写`move(n, a, b, c)`函数，它接收参数`n`，表示3个柱子A、B、C中第1个柱子A的盘子数量，然后打印出把所有盘子从A借助B移动到C的方法，例如：

```
# 期待输出:
# A --> C
# A --> B
# C --> B
# A --> C
# B --> A
# B --> C
# A --> C
move(3, 'A', 'B', 'C')
```