函数

圆的面积公式为：

$S = \pi r^2$

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：

In [ ]:
```
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，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$\sum$，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 $\sum$ 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：

$$\sum_{n=1}^{100} (n^2 + 1)$$

还原成加法运算就变成了：

$$(1 \times 1 + 1) + (2 \times 2 + 1) + (3 \times 3 + 1) + \cdots + (100 \times 100 + 1)$$

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 (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：

In [1]:
```
print(abs(100))
print(abs(-20))
print(abs(12.34))
```

```
100
20
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：

In [2]:
```
abs(1, 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-05b55862d84c> in <module>()
----> 1 abs(1, 2)

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：

In [3]:
```
abs('a')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-3b9a69fe3abb> in <module>()
----> 1 abs('a')

TypeError: bad operand type for abs(): 'str'
```

而max()可以接收任意多个参数，并返回最大的那个：

In [4]:
```
print(max(1, 2))
print(max(1, 2, 3, 4))
```

```
2
4
```

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。

In [12]:
```
print(int('123'))
print(int(12.34))

print(float('12.34'))
print(str(1.23))

print(bool(1))
print(bool(0))
print(bool(''))
```

```
123
12
12.34
1.23
True
False
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”：

In [14]:
```
a = abs
a(-1)
```

Out[14]: 1

定义函数

在Python中，定义一个函数要使用def关键字，依次写出

- 函数名
- 括号
- 括号中的参数
- 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：

In [17]:
```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x

print(my_abs(-1))
```

```
1
```

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。

假设已将my_abs()的函数定义保存为abstest.py文件，那么，在该文件的当前目录下启动Python解释器，可用from abstest import my_abs来导入my_abs()。

注意abstest是文件名（不含.py扩展名).

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：

In [18]:
```
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：

In [ ]:
```
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：

In [20]:
```
my_abs(1, 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-7b1d9a5f5e7e> in <module>()
----> 1 my_abs(1, 2)

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。

In [23]:
```
1 my_abs('A')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-fe695080d9cd> in <module>()
----> 1 my_abs('A')

<ipython-input-17-9b924b03673e> in my_abs(x)
      1 def my_abs(x):
----> 2     if x >= 0:
      3         return x
      4     else:
      5         return -x

TypeError: unorderable types: str() >= int()
```

In [24]:
```
1 abs('A')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-8669504e2fca> in <module>()
----> 1 abs('A')

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：

In [1]:
```
1 def my_abs(x):
2     if not isinstance(x, (int, float)):
3         raise TypeError('bad operand type')
4     if x >= 0:
5         return x
6     else:
7         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。

In [26]:
```
1 my_abs('A')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-fe695080d9cd> in <module>()
----> 1 my_abs('A')

<ipython-input-25-2390eb4c2721> in my_abs(x)
      1 def my_abs(x):
      2     if not isinstance(x, (int, float)):
----> 3         raise TypeError('bad operand type')
      4     if x >= 0:
      5         return x

TypeError: bad operand type
```

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，给定坐标、位移和角度，就可以计算出新的坐标：

In [27]:
```
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：

In [28]:
```
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：

In [29]:
```
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。

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
- 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：

In [31]:
```
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：

In [32]:
```
print(power(5))
print(power(25))
```

```
25
625
```

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：

In [33]:
```
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：

In [35]:
```
print(power(5, 2))
print(power(5, 3))
```

```
25
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：

In [36]:
```
1 power(5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-1fcd865a69f9> in <module>()
----> 1 power(5)

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，默认参数就派上用场了。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：

In [37]:
```
1 def power(x, n=2):
2     s = 1
3     while n > 0:
4         n = n - 1
5         s = s * x
6     return s
```

这样，当我们调用power(5)时，相当于调用power(5, 2)：

In [39]:
```
1 print(power(5))
2 print(power(5, 2))
```

```
25
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：

1. 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）；
2. 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。

举个例子，我们写个大一新生注册的函数，需要传入name和gender两个参数：

In [64]:
```
1 def enroll(name, gender):
2     print('name:', name)
3     print('gender:', gender)
```

这样，调用enroll()函数只需要传入两个参数：

In [65]:
```
1 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：

In [67]:
```
def enroll(name, gender, age=18, city='Wuhan'):
    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：

In [68]:
```
enroll('Sarah', 'F')
```

```
name: Sarah
gender: F
age: 18
city: Wuhan
```

只有与默认参数不符的学生才需要提供额外的信息：

In [69]:
```
enroll('Bob', 'F', 19)
enroll('Adam', 'M', city='Beijing')
```

```
name: Bob
gender: F
age: 19
city: Wuhan
name: Adam
gender: M
age: 18
city: Beijing
```

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：

In [3]:
```
def add_end(L=[]):
    L.append('END')
    return L
```

当你正常调用时，结果似乎不错：

In [4]:
```
print(add_end([1, 2, 3]))
print(add_end(['x', 'y', 'z']))
```

```
[1, 2, 3, 'END']
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

In [5]:
```
print(add_end())
```

```
['END']
```

但是，再次调用add_end()时，结果就不对了：

In [7]:
```
print(add_end())
```

```
['END', 'END', 'END']
```

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。

所以，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：

In [77]:
```
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：

In [78]:
```
print(add_end())
print(add_end())
print(add_end())
```

```
['END']
['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。 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。

例： 给定一组数字a，b，c，$\cdots$，请计算$a^2 + b^2 + c^2 + \cdots$。

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，$\cdots$作为一个list或tuple传进来，这样，函数可以定义如下：

In [79]:
```
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：

In [81]:
```
print(calc([1, 2, 3]))
print(calc((1, 3, 5, 7)))
```

```
14
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

In [ ]:
```
calc(1, 2, 3)
calc(1, 3, 5, 7)
```

所以，我们把函数的参数改为可变参数：

In [84]:
```
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：

In [86]:
```
print(calc(1, 2))
print(calc())
```

```
5
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：

In [87]:
```
nums = [1, 2, 3]
calc(nums[0], nums[1], nums[2])
```

Out[87]: 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：

In [90]:
```
nums = [1, 2, 3]
calc(*nums)
```

Out[90]: 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：

In [92]:
```
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：

In [93]:
```
person('Michael', 30)
```

```
name: Michael age: 30 other: {}
```

也可以传入任意个数的关键字参数：

In [94]:
```
person('Bob', 35, city='Beijing')
```

```
name: Bob age: 35 other: {'city': 'Beijing'}
```

In [95]:
```
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。

当然，上面复杂的调用可以用简化的写法：

In [96]:
```
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：

In [97]:
```
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：

In [98]:
```
person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123456)
```

```
name: Jack age: 24 other: {'city': 'Beijing', 'addr': 'Chaoyang', 'zipcode': 12
3456}
```

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：

In [99]:
```
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：

In [105]:
```
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：

In [101]:
```
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：

In [102]:
```
person('Jack', 24, 'Beijing', 'Engineer')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-102-70c5f703212a> in <module>()
----> 1 person('Jack', 24, 'Beijing', 'Engineer')

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：

In [106]:
```
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：

In [108]:
```
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：

In [110]:
```
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：

1. 必选参数
2. 默认参数
3. 可变参数
4. 命名关键字参数
5. 关键字参数。

比如定义一个函数，包含上述若干种参数：

In [111]:
```
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。

In [115]:
```
f1(1, 2)
f1(1, 2, 3)
f1(1, 2, 3, 'a', 'b')
f1(1, 2, 3, 'a', 'b', x = 99)
```

```
a = 1 b = 2 c = 0 args = () kw = {}
a = 1 b = 2 c = 3 args = () kw = {}
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
```

In [122]:
```
f2(1, 2, d = 99)
f2(1, 2, d = 99, ext = None)
f2(1, 2, ext = None)
```

```
a = 1 b = 2 c = 0 d = 99 kw = {}
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-122-3d88e7382e80> in <module>()
      1 f2(1, 2, d = 99)
      2 f2(1, 2, d = 99, ext = None)
----> 3 f2(1, 2, ext = None)

TypeError: f2() missing 1 required keyword-only argument: 'd'
```

最神奇的是通过一个tuple和dict，你也可以调用上述函数：

In [126]:
```
1 args = (1, 2, 3, 4)
2 kw = {'d': 99, 'x':'#'}
3 f1(*args, **kw)
```

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

In [128]:
```
1 args = (1, 2, 3)
2 kw = {'d': 88, 'x':'#'}
3 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。

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。

- 要注意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：

    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。

In [2]:
```
1 print(range(100))
```

```
range(0, 100)
```

```
1
```