# 第二部分 模块化开发

##### 调用函数

###### 内建函数

Python内置了很多有用的函数，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。

abs(100)

abs(-20)

abs(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：abs()有且仅有1个参数，但给出了两个：

abs(1, 2)

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')

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)

max(2, 3, 1, -5)

###### 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')

int(12.34)

float('12.34')

str(1.23)

str(100)

bool(1)

bool('')

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

##### 定义函数

###### 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。

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。

###### 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)

但是如果参数类型不对，Python解释器就无法帮检查。试试my\_abs和内置函数abs的差别：

my\_abs('A')

abs('A')

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')

###### 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，就可以同时获得返回值：

x, y = move(100, 100, 60, math.pi / 6)

print(x, y)

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)

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。

###### 函数总结

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，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

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。

当调用power函数时，必须传入有且仅有的一个参数x：

power(5)

power(15)

现在，如果要计算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？不可能定义无限多个函数。

可以把power(x)修改为power(x, n)，用来计算xn，：

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)

power(5, 3)

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)

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，当调用power(5)时，相当于调用power(5, 2)：

power(5)

power(5, 2)

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，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')

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。

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')

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，也会出问题，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，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。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。

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，必须确定输入的参数。由于参数个数不确定，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，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)

也可以传入任意个数的关键字参数：

person('Bob', 35, 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，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，也能收到。试想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'])

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)

\*\*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，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')

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')

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)

f1(1, 2, c=3)

f1(1, 2, 3, 'a', 'b')

f1(1, 2, 3, 'a', 'b', x=99)

f2(1, 2, d=99, ext=None)

最神奇的是通过一个tuple和dict，也可以调用上述函数：

args = (1, 2, 3, 4)

kw = {'d': 99, 'x': '#'}

f1(\*args, \*\*kw)

args = (1, 2, 3)

kw = {'d': 88, 'x': '#'}

f2(\*args, \*\*kw)

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。

##### 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，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。测试：

fact(1)

fact(5)

fact(100)

如果计算fact(5)，可以根据函数定义看到计算过程如下：

fact(5)

执行过程：

|  |
| --- |
| 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 |

执行结果：

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)

执行结果：

|  |
| --- |
| 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)

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。

###### 扩展

思考：如何优化尾递归，使得不发生堆栈溢出？

##### 高阶函数

###### map/reduce

Python内建了map()和reduce()函数。

先看map。map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

举例说明，比如有一个函数f(x)=x2，要把这个函数作用在一个list [1, 2, 3, 4, 5, 6, 7, 8, 9]上，就可以用map()实现如下：

def f(x):

... return x \* x

...

r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

list(r)

map()传入的第一个参数是f，即函数对象本身。由于结果r是一个Iterator，Iterator是惰性序列，因此通过list()函数让它把整个序列都计算出来并返回一个list。

不需要map()函数，写一个循环，也可以计算出结果：

L = []

for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:

L.append(f(n))

print(L)

的确可以，但是，从上面的循环代码，能一眼看明白“把f(x)作用在list的每一个元素并把结果生成一个新的list”吗？

所以，map()作为高阶函数，事实上它把运算规则抽象了，因此，不但可以计算简单的f(x)=x2，还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串：

list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

只需要一行代码。

再看reduce的用法。reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，其效果就是：

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

比方说对一个序列求和，就可以用reduce实现：

from functools import reduce

def add(x, y):

return x + y

...

reduce(add, [1, 3, 5, 7, 9])

当然求和运算可以直接用Python内建函数sum()，没必要动用reduce。

但是如果要把序列[1, 3, 5, 7, 9]变换成整数13579，reduce就可以派上用场：

from functools import reduce

def fn(x, y):

... return x \* 10 + y

...

reduce(fn, [1, 3, 5, 7, 9])

这个例子本身没多大用处，但是，如果考虑到字符串str也是一个序列，对上面的例子稍加改动，配合map()，就可以写出把str转换为int的函数：

from functools import reduce

def fn(x, y):

... return x \* 10 + y

...

def char2num(s):

... digits = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}

... return digits[s]

...

reduce(fn, map(char2num, '13579'))

整理成一个str2int的函数就是：

from functools import reduce

DIGITS = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}

def str2int(s):

def fn(x, y):

return x \* 10 + y

def char2num(s):

return DIGITS[s]

return reduce(fn, map(char2num, s))

还可以用lambda函数进一步简化成：

from functools import reduce

DIGITS = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}

def char2num(s):

return DIGITS[s]

def str2int(s):

return reduce(lambda x, y: x \* 10 + y, map(char2num, s))

也就是说，假设Python没有提供int()函数，完全可以自己写一个把字符串转化为整数的函数，而且只需要几行代码！

###### filter

Python内建的filter()函数用于过滤序列。

和map()类似，filter()也接收一个函数和一个序列。和map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

def is\_odd(n):

return n % 2 == 1

list(filter(is\_odd, [1, 2, 4, 5, 6, 9, 10, 15]))

# 结果: [1, 5, 9, 15]

把一个序列中的空字符串删掉，可以这么写：

def not\_empty(s):

return s and s.strip()

list(filter(not\_empty, ['A', '', 'B', None, 'C', ' ']))

# 结果: ['A', 'B', 'C']

可见用filter()这个高阶函数，关键在于正确实现一个“筛选”函数。

注意到filter()函数返回的是一个Iterator，也就是一个惰性序列，所以要强迫filter()完成计算结果，需要用list()函数获得所有结果并返回list。

###### sorted

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，可以直接比较，但如果是字符串或者两个dict呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。

Python内置的sorted()函数就可以对list进行排序：

sorted([36, 5, -12, 9, -21])

此外，sorted()函数也是一个高阶函数，它还可以接收一个key函数来实现自定义的排序，例如按绝对值大小排序：

sorted([36, 5, -12, 9, -21], key=abs)

key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序。

sorted(['bob', 'about', 'Zoo', 'Credit'])

默认情况下，对字符串排序，是按照ASCII的大小比较的，由于'Z' < 'a'，结果，大写字母Z会排在小写字母a的前面。

现在，提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要能用一个key函数把字符串映射为忽略大小写排序即可。忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

这样，给sorted传入key函数，即可实现忽略大小写的排序：

sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)

要进行反向排序，不必改动key函数，可以传入第三个参数reverse=True：

sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, reverse=True)

对字典排序

students = [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

print(sorted(students, key=lambda s: s[2]))

print(sorted(students, key=lambda s: s[2], reverse=True))

从上述例子可以看出，高阶函数的抽象能力是非常强大的，而且，核心代码可以保持得非常简洁。

##### 返回函数

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

def calc\_sum(\*args):

ax = 0

for n in args:

ax = ax + n

return ax

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数：

def lazy\_sum(\*args):

def sum():

ax = 0

for n in args:

ax = ax + n

return ax

return sum

当调用lazy\_sum()时，返回的并不是求和结果，而是求和函数：

f = lazy\_sum(1, 3, 5, 7, 9)

f

调用函数f时，才真正计算求和的结果：

f()

在这个例子中，在函数lazy\_sum中又定义了函数sum，并且，内部函数sum可以引用外部函数lazy\_sum的参数和局部变量，当lazy\_sum返回函数sum时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当调用lazy\_sum()时，每次调用都会返回一个新的函数，即使传入相同的参数：

f1 = lazy\_sum(1, 3, 5, 7, 9)

f2 = lazy\_sum(1, 3, 5, 7, 9)

f1==f2

f1()和f2()的调用结果互不影响。

##### 闭包

注意到返回的函数在其定义内部引用了局部变量args，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了f()才执行。来看一个例子：

def count():

fs = []

for i in range(1, 4):

def f():

return i\*i

fs.append(f)

return fs

f1, f2, f3 = count()

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

连续调用f1()，f2()和f3()结，观察结果：

f1()

f2()

f3()

全部都是9！原因就在于返回的函数引用了变量i，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量i已经变成了3，因此最终结果为9。

**返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。**

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

def count():

def f(j):

def g():

return j\*j

return g

fs = []

for i in range(1, 4):

fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()

return fs

再看看结果：

f1, f2, f3 = count()

f1()

f2()

f3()

##### 匿名函数

当在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，对匿名函数提供了有限支持。还是以map()函数为例，计算f(x)=x2时，除了定义一个f(x)的函数外，还可以直接传入匿名函数：

list(map(lambda x: x \* x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

通过对比可以看出，匿名函数lambda x: x \* x实际上就是：

def f(x):

return x \* x

关键字lambda表示匿名函数，冒号前面的x表示函数参数。

匿名函数有个限制，就是只能有一个表达式，不用写return，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

f = lambda x: x \* x

f

f(5)

同样，也可以把匿名函数作为返回值返回，比如：

def build(x, y):

return lambda: x \* x + y \* y

##### 装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

def now():

... print('2019-9-29')

...

f = now

f()

函数对象有一个\_\_name\_\_属性，可以拿到函数的名字：

now.\_\_name\_\_

f.\_\_name\_\_

现在，假设要增强now()函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改now()函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

本质上，decorator就是一个返回函数的高阶函数。所以，要定义一个能打印日志的decorator，可以定义如下：

def log(func):

def wrapper(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

return func(\*args, \*\*kw)

return wrapper

观察上面的log，因为它是一个decorator，所以接受一个函数作为参数，并返回一个函数。要借助Python的@语法，把decorator置于函数的定义处：

@log

def now():

print('2019-9-29')

调用now()函数，不仅会运行now()函数本身，还会在运行now()函数前打印一行日志：

now()

call now():

把@log放到now()函数的定义处，相当于执行了语句：

now = log(now)

由于log()是一个decorator，返回一个函数，所以，原来的now()函数仍然存在，只是现在同名的now变量指向了新的函数，于是调用now()将执行新函数，即在log()函数中返回的wrapper()函数。

wrapper()函数的参数定义是(\*args, \*\*kw)，因此，wrapper()函数可以接受任意参数的调用。在wrapper()函数内，首先打印日志，再紧接着调用原始函数。

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

def log(text):

def decorator(func):

def wrapper(\*args, \*\*kw):

print('%s %s():' % (text, func.\_\_name\_\_))

return func(\*args, \*\*kw)

return wrapper

return decorator

这个3层嵌套的decorator用法如下：

@log('execute')

def now():

print('2019-9-29')

和两层嵌套的decorator相比，3层嵌套的效果是这样的：

now = log('execute')(now)

来剖析上面的语句，首先执行log('execute')，返回的是decorator函数，再调用返回的函数，参数是now函数，返回值最终是wrapper函数。

以上两种decorator的定义都没有问题，但还差最后一步。因为讲了函数也是对象，它有\_\_name\_\_等属性，但去看经过decorator装饰之后的函数，它们的\_\_name\_\_已经从原来的'now'变成了'wrapper'：

now.\_\_name\_\_

因为返回的那个wrapper()函数名字就是'wrapper'，所以，需要把原始函数的\_\_name\_\_等属性复制到wrapper()函数中，否则，有些依赖函数签名的代码执行就会出错。这个复制过程由python解释器完成。

##### 偏函数

Python的functools模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下：

int()函数可以把字符串转换为整数，当仅传入字符串时，int()函数默认按十进制转换：

int('12345')

但int()函数还提供额外的base参数，默认值为10。如果传入base参数，就可以做N进制的转换：

int('12345', base=8)

int('12345', 16)

假设要转换大量的二进制字符串，每次都传入int(x, base=2)非常麻烦，于是，想到，可以定义一个int2()的函数，默认把base=2传进去：

def int2(x, base=2):

return int(x, base)

这样，转换二进制就非常方便了：

int2('1000000')

int2('1010101')

functools.partial就是帮助创建一个偏函数的，不需要自己定义int2()，可以直接使用下面的代码创建一个新的函数int2：

import functools

int2 = functools.partial(int, base=2)

int2('1000000')

int2('1010101')

所以，简单总结functools.partial的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

##### 使用模块

###### 模块

Python本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。

以内建的sys模块为例，编写一个hello的模块：

# -\*- coding: utf-8 -\*-

' a test module '

\_\_author\_\_ = 'Michael Liao'

import sys

def test():

args = sys.argv

if len(args)==1:

print('Hello, world!')

elif len(args)==2:

print('Hello, %s!' % args[1])

else:

print('Too many arguments!')

if \_\_name\_\_=='\_\_main\_\_':

test()

第1行是标准注释，该注释表示.py文件本身使用标准UTF-8编码；

第2行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

第3行使用\_\_author\_\_变量把作者写进去；

以上就是Python模块的标准文件模板，当然也可以全部删掉不写，但是，按标准办事肯定没错。

后面开始就是真正的代码部分。

可能注意到了，使用sys模块的第一步，就是导入该模块：

import sys

导入sys模块后，就有了变量sys指向该模块，利用sys这个变量，就可以访问sys模块的所有功能。

sys模块有一个argv变量，用list存储了命令行的所有参数。argv至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行python3 hello.py获得的sys.argv就是['hello.py']；

运行python3 hello.py Michael获得的sys.argv就是['hello.py', 'Michael]。

最后，注意到这两行代码：

if \_\_name\_\_=='\_\_main\_\_':

test()

当在命令行运行hello模块文件时，Python解释器把一个特殊变量\_\_name\_\_置为\_\_main\_\_，而如果在其他地方导入该hello模块时，if判断将失败，因此，**这种if测试可以让一个模块通过命令行运行时执行一些额外的代码**，最常见的就是运行测试。

可以用命令行运行hello.py看看效果：

python3 hello.py

python hello.py Michael

###### 作用域

在一个模块中，可能会定义很多函数和变量，但有的函数和变量希望给别人使用，有的函数和变量希望仅仅在模块内部使用。在Python中，是通过\_前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：abc，x123，PI等；

**类似\_\_xxx\_\_这样的变量是特殊变量，可以被直接引用，但是有特殊用途**，比如上面的\_\_author\_\_，\_\_name\_\_就是特殊变量，hello模块定义的文档注释也可以用特殊变量\_\_doc\_\_访问，自己的变量一般不要用这种变量名；

**类似\_xxx和\_\_xxx这样的函数或变量就是非公开的**（private），不应该被直接引用，比如\_abc，\_\_abc等；

之所以说，private函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

private函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

def \_private\_1(name):

return 'Hello, %s' % name

def \_private\_2(name):

return 'Hi, %s' % name

def greeting(name):

if len(name) > 3:

return \_private\_1(name)

else:

return \_private\_2(name)

在模块里公开greeting()函数，而把内部逻辑用private函数隐藏起来了，这样，调用greeting()函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

##### 安装第三方模块

###### 安装常用模块

一般来说，第三方库都会在Python官方的pypi.python.org网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者pypi上搜索，比如Pillow的名称叫Pillow，因此，安装Pillow的命令就是：

pip install Pillow

耐心等待下载并安装后，就可以使用Pillow了。

在使用Python时，经常需要用到很多第三方库，例如，上面提到的Pillow，以及MySQL驱动程序，Web框架Flask，科学计算Numpy等。用pip一个一个安装费时费力，还需要考虑兼容性。推荐直接使用Anaconda，这是一个基于Python的数据处理和科学计算平台，它已经内置了许多非常有用的第三方库，装上Anaconda，就相当于把数十个第三方模块自动安装好了，非常简单易用。

可以从Anaconda官网下载GUI安装包，安装包有500~600M，所以需要耐心等待下载。网速慢的同学请移步国内镜像。下载后直接安装，Anaconda会把系统Path中的python指向自己自带的Python，并且，Anaconda安装的第三方模块会安装在Anaconda自己的路径下，不影响系统已安装的Python目录。

安装好Anaconda后，重新打开命令行窗口，输入python，可以看到Anaconda的信息：

可以尝试直接import numpy等已安装的第三方模块。

###### 模块搜索路径

当试图加载一个模块时，Python会在指定的路径下搜索对应的.py文件，如果找不到，就会报错：

import mymodule

默认情况下，Python解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在sys模块的path变量中：

import sys

sys.path

['', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python36.zip', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6', ..., '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages']

如果要添加自己的搜索目录，有两种方法：

一是直接修改sys.path，添加要搜索的目录：

import sys

sys.path.append('/Users/michael/my\_py\_scripts')

这种方法是在运行时修改，运行结束后失效。

第二种方法是设置环境变量PYTHONPATH，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置Path环境变量类似。注意只需要添加自己的搜索路径，Python自己本身的搜索路径不受影响。