**第8章 函数和函数式编程**

函数最重要的目的是方便人们重复使用相同的一段程序，它可以对程序进行结构化处理， 将一些特定功能的代码，巧妙地封装成一个方便管理的代码块，这些操作的代码隶属于函数。 后期在开发过程中，一旦想实现相同的操作，只需调用函数名就可以，而不需要重复复制大量 的代码，这样不仅节省了空间，也提高了编码效率。

既1讥飒商教

在学习本章内容之前，其实我们已经接触过了一些函数，比如使用pm<)函数输出结果。 函数是组织好的，可重复使用的，用来实现单一或相关联功能的代码段。它的作用主要体现在 两方面：

**1.**提高应用鮑模媛雄，最小化代醯冗亲

在计算机编程中，经常有一些逻辑算法需要打包，从而使其能够在不同的地方不止一次地 使用。函数允许整合这些代码，并将这些代码在整合之后可以多次使用。在Python程序中，函 数不仅可以提高程序的模块性，最大程度的减少代码冗余，而且为后期代码的维护节省了不少 时间。

**2.**流程駒分解

使用函数可以将整个流程分隔为一个一个子任务，这样后期只要将这些子任务实现，就完 成了整个流程。例如，将食物放进冰箱这个程序，我们可以分解为三个子任务：

(1)打开冰箱。

(2 )将食物放入冰箱。

(3)关闭冰箱。

然后使用函数分别实现这三个子任务即可。

本章将介绍如何自己编写函数，并且可以让这些函数使用起来和内置函数一样，可以传递 值，可以返回值，实现期望的功能。

So 2禰數的定夏SW厕

在Python中，你可以定义一个自己想要功能的函数，自定义函数的语法格式如下所示。

def函数名(［参数］):

"函数\_文档字符串叫

函数体—

return ［表达式］

关于上述语法格式的介绍如下所示。

•函数代码块以d©f关键词开头，后接函数标识符名称和圆括号()。

•任何传入参数必须放在圆括号中间，圆括号之间可以用于定义参数。函数参数可以有多 个，也可以没有，但圆括号要保留。

•定义函数的冒号后面，第一行语句可以选择性地使用文档字符串，专门用于存放函数 说明。

-函数内容以冒号起始，并且缩进。

0 return ［表达式］结束函数,选择性地返回一个值给调用方。不带表达式的return相当于 返回None。Python中的函数允许没有返回值,也就是不用return。当然,return也可以 返回多个值，以逗号分隔，相当于返回一个元组。如果wturn后面没有返回值，函数将 自动返回None, No眼是Python中的一个特别的数据类型，表示什么都没有

默认情况下，参数值和参数名称是按函数声明中定义的顺序匹配起来的。

下面是自定义的一个函数，用于计算两个数的乘积，示例如下：

def fun\_test(azb):

return a\*b #使用return将a\*b的结果返回

上述代码的相关介绍如下所示。

(1 )使用def这个关键字通知Python：在定义一个函数，funjtest是函数名。当Python运 行到def语句时，它会生成一个新的函数对象，并将其赋值给函数名。

(2)括号中的a, b是函数的参数，是对函数的输入。

(3 )关于冒号和缩进,我们已经在循环和选择语句中见过,它表7K的是隶属关系。

(4)匆\*b是函数内部进行的运算。

(5 ) return a\*b用于将a\*b的运算结果返回,也就是输出的功能。

(6 )当Python程序代码包含注释时，编译器会跳过注释。

観22 囑魇 "

定义好函数之后，就可以在后面程序中使用这一函数。函数的调用非常简单，只需要使用“函 数名()”即可，例如歹调用上述fun\_test()函数的格式如下所示。

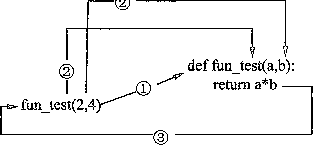
上述代码中9当调用函数ftm\_test()时，传递了两 个值。在这个例子中，忍被赋值为2, b被赋值为4, 使用return语句将a\*b的结果返回。函数调用的过程如 图84所示。

图8-1函数调用过程

在调用函数时，我们还可以传递不同类型的值， 示例如下：

>>> fun\_test(? x \* 7 6)

\* XXXXXX \*

这次和上次调用相比，函数的作用完全不同，这次调用中，我们将一个字符串和一个数字 传递给為和b，它们相乘的结果是一个新的字符串。乘号\*对数字和序列都有效，这些都在前 面有过相关讲解。

Pythoii自带了很多在线的模块资源，它们提供了丰富的功能，当我们使用这些模块时，如 果每次都去网上查找，势必会耗费很多时间，结果也不一定准确。

这里，给大家讲一下Python自带的查看帮助功能一hEp()命令，它可以快速准确地找到 指定模块或者函数的使用方法。例如，我们使用help()查看input函数的使用方法，查看命令 如下所示。

help (input)

此时，我们可以直接看到input()函数的相关说明，具体如图奖所示。

= 謹膈腿劍皿奇hgl桌面 …口 X

文件(円 编辑(曰 ie(v)搜索⑸ 缪端⑴

Helo 05 bJil t- *in* fjqction in rodjle b^iltins：

input(prospt= hicme, /)

Read a string standard inpyt. The trailing newline is stripped.

The prompt 15 p^i^ted to £t&湖aP output witnoyt a £整

trailing newline before reading input,

:「t*ir* \_1: EOF fw l-D, eg 卄 1-Z+Return), raise EOFError. If Or；*細\** systeros, deadline is ysed if ayaila&le.

图8-2使用help查看input函数

在团队开发中，面对众多自定义函数，如果希望自己的函数调用时，也能通过help()命令 快速准确查询函数的功能或者使用方式，就需要我们在定义函数时，增加函数的相关说明。前 面讲过，函数的第1行语句可以使用字符串来存放函数说明，接下来，我们自定义一个函数， 并使用help()命令验证是否能得到该函数的说明。

>>> def sum(arb):

"该函数用于两个数的求和操作"

*..。* return a+b

查看结果如下所示。

Help on function sum in module main :

sum(az b)

该函数用于两个数的求和操作

(END)

后续讲解函数的参数传递时，会提到形参和实参两个名词，这里，先来对形参和实参进行 一个简单介绍。

(1)形参，全称是“形式参数”，它是在定义函数名和函数体时使用的参数，目的是接 收调用该函数传递的参数。

(2 )实参,全称是"实际参数”，它是在调用时传递给函数的参数，即传递给被调用函数的值。 实参可以是常量、变量、表达式、函数等，它不管是哪种类型，在进行函数调用时，必须具有 确定的值，这些值会传递给形参。

下面看一个函数，具体如下：

def sum(azb): return a+b

在sum函数中，参数a和b就是形参。此时，如果调用sum函数，需要传递两个值，例如：

sum(10\*20)

上述代码在调用wm函数时，传入了两个具体的值10和20,这两个值就是实参。实参是 形参被具体赋值之后的值，它参与实际运算，具有实际的作用。

当函数获取实参后，有的可以在参数内部将实参进行修改，有的则不能修改。能不能修改， 取决于参数的类型，并不是在函数内部对参数重新赋值就能够改变参数的值。

字符串、数字和元组是不可变的，这种类型的参数是无法被修改的(即在函数内部不能真 正改变传入的参数)。当参数是列表、字典类型时是可以修改的。字符串、数字、元组类似于 c中的值传递，列表、字典类似于c中的引用传递。

当参数是列表，而又不想修改传入参数时，可以通过创建列表的切片来传入列表的一个副 本，即当name是一个列表时，则传入name[:]来避免在函数内部将参数改变。

&roturo

假设，我们把函数看作工厂里的机器，我们往机器里面输送生产用的材料，最终机器会将 产品生产好。这里，我们可以把产品的材料看作函数的参数，把做好的产品看作函数的输出， 而生产的过程就是函数体的代码了。

例如，下面这段代码：

def work (material\_017 material\_02material\_03):

product\_01 = material\_01 + material\_02 product\_02 = product\_01 \* material\_03

这个机器在一系列指令下,使用material\_01、materiaI\_02、material\_03三种材料,生产出 T product\_0k product\_02两个产品，此时，使用冀turn可以返回给我们想要的产品，假如， 我们需要product\_02物品，那么可以使用电urn返回给我们，具体代码如下：

def work(material\_01, material\_02z material\_03): product\_01 = material\_01 \* material\_02 product\_02 = product\_01 + material\_03 return product\_02

此时，只要调用work()这个函数，就可以让机器干活，并拿到我们希望得到的物品。

result = work (17 2 3)

此时，我们会得到对应的结果((1\*2)+3)。

在前面定义函数的时候，我们已经讲过return用于返回函数值。可以说，每个函数都有一 个:return语句，如果函数没有定义返回值，那么返回值就是None, None表示没有任何值，属 于NoneType类型。下面看一段代码。

»> def fun\_test ():

。。。 print ( ? he]\_lo world5 )

>>> result = fun\_test()

hello world

>>> type(result) <class ? NoneType 5 >

另外，Python函数返回值可以是一个或者多个值，如果显式地使用fetum返回一个值，那 么该返回值类型就是对应值的类型，如果使用冀tum返回多个值，那么这些值会聚集起来并以 元组类型返回。

返回值只有一个的示例如下所示。

>>> def sum(a7b):

... c = a+b

.o. return c

»> type (sum (1,2))

<class 5 int, >

返回值有多个的示例如下所示。

»> def number (af b):

。。 return b

>>> type(number(1f 2))

<class ? tuple 1>

| 返回值个数 | 返回值类型 |
| --- | --- |
| 0 | None |
| 1 | object |
| 大于1 | tuple |

返回值个数与返回值类型的对应关系如表84所示。

表8』返回值对应类型

M 3函魏胸参數俺通

之前，我们已经接触过函数的参数，它主要用于接收用户的 输入。总的来说,Python函数的参数传递方式有很多种，包括位 置参数传递、关键字参数传递、默认参数传递、可变参数传递、解包裹以及混合方式传递。下 面将结合案例对这些参数的传递进行介绍。

8.3J

先观察下面两个函数，看看什么是位置参数。

函数1：

>>> def fun\_01(xf *yf* z): .•. return *xr* y, z

>>> print(fun\_01(1, 2f 3)) (1, 2, 3)

函数2：

>>> def fun\_02(x, *yf* z):

• . . return z, xz y

>>> print(fun\_02(1f *2r* 3))

(3, 1, 2)

函数fon\_01()和ftm\_02()在定义时的参数名是一样的，并且在调用函数时传入的参数也是 一样的，但是返回的结果却是不同的。这是为什么呢？

问题就出在冀turn返回的参数位置上，当调用函数时，传入的参数位置是和定义函数的 参数位置对应的，也就是说，当调用函数传入参数的那一刻，函数的参数值已经确定了，如 x=l, y=2, z=3。至于函数输岀结果的顺序，是由冀turn返回值的顺序决定的。另外，使用位 置参数传递时，调用函数传入参数的数量，要和定义函数的参数数量相同，否则程序会出错。

8.3.2美鱸寥養敵億邏

假设某个函数的参数非常复杂，为了不让程序出错，我们可以把参数的名称与值绑定在一 起，使用参数名提供的参数叫作关键字参数。下面看一个例子：

>>> def connect(urlzport):

。。。 return 5 url is %s z and port is %s , % (url7 port)

>>> connect(url = 1wwwo itcast.cn J 7port = 5 8080 1 )

1url is [www.itcast.cn](http://www.itcast.cn7)[7](http://www.itcast.cn7) and port is 80801

使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为Python解释器能够用

参数名匹配参数值。例如，下面调用connectO函数的结果与上面的结果是一致的。

>» connect (port = 1 8080 \ url = 1 www. itcast. cn \* )

1url is www.itcast.cn,and port is 80801

关键字参数可以和位置参数混合使用9但是位置参数必须要出现在关键字参数之前，例如 下面的代码：

»> def test (azbf c):

... return azb,c

>>> print(test(1rc=2fb=3))

dz 3, 2)

飢鼠舫養數獸飒値

在定义函数的时候，使用形如定19的方式，可以给参数赋予默UB (default)。如果该参 数最终没有被传递值，将使用该默认值。示例如下：

>>> def sum(azbz c=2 0):

□ . . return a+b+c

»> print (sum (10f 10))

40

>» print (sum (10,10z 10))

30

上述代码中，当我们第一次调用wm()函数的时候，并没有传入足够个数的值，参数c没 有被赋值，将使用默认值20。第二次调用函数的时候，参数c被赋值为10,不再使用默认值。

8.3.4園囊鑿數 ’

在Python中，函数可以定义可变参数，顾名思义，可变参数指的就是传入函数的参数是 可变的，在函数中定义可变参数的语法格式如下所示M

def functionname([formal\_args,] \*argsf \*\*kwargs):

”函数—文档字符串w - function\_\_suite return [expression]

上述格式中，以星号(\*)开始的变量args会存放所有未命名的变量参数，args为元组， 示例如下：

>>> def test(\*args):

• . . print(args)

»> test (1,2,3, 'a\ fbS y

dr 2, 3, !as \*b\ !c!)

以\*\*开始的变量kwargs会存放命名参数，即形如key=value的参数，kwargs为字典。示

例如下:

>>> def test(\*\*kwargs):

。… print (kwargs)

>» test (a = 1 b = 2 f c = 3 f d = 4)

{?c?: 3, !d5: 4, : 1, ?bJ: 2} •

如果加了\*和心的参数混合使用，那么传入的顺序必须和声明时的顺序一致，示例如下:

>>> def fun(\*argsr \*\*kwargs):

.。. print(args)

,.. print(kwargs)

>>> fun(1,2,3z a = 4,b = 5)

(1, 2, 3)

{\*as: 4, ,bJ: 5}

»> fun (1,2,3)

(L 2, 3)

{}

>>> fun(a = 4rb = 5)

0

{: 4, ?b!: 5}

上述代码中，调用函数传入的参数顺序与定义时一致，如果调用函数时没有指定参数，那 么它就是一个空的元组或者空的字典。

至此，我们可以针对可变参数的定义做如下总结:

•使用\*可以将未命名的参数打包成元组类型。

•使用\*\*可以将命名的参数打包成字典类型。

8b3o5 •

上一节，我们讲解了使用\*和心，对函数参数进行包裹。这里9我们要给大家讲解如何解 包裹。所谓解包裹就是在调用函数的时候使用\*.和\*\*。需要说明的是，解包裹和可变参数的传 递并不是相反的操作，他们是两个独立的过程。下面看一个解包裹的例子。

>>> def func(af br c): ... print(a,c)

>>> args = (1,2,3) •

>>> func(\*args)

12 3

上述案例中,在调用函数fimc()的时候,使用\*是为了告诉Python,我要把args拆分为三 个元素,分别传递给a、b、Co

大家试想一下，如果在调用fiinc()时，不在wgs前面添加\*会是什么后果？我们先来测试 一下：

»> func (args)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: func () missing 2 required positional arguments: f b v and J c J 从上述报错信息可以看出，提示我们在传递参数的时候，缺少两个位置参数b和C。这是 因为如果不使用七系统会把args当作一个元素传递给函数，所以导致b和c位置参数缺失的 情况。

相应的9如果要对字典解包裹，需要在调用的时候，添加心符号，示例如下：

»> kwargs = {Ja,:lzvbJ:2z?c,:3}

>>> func(\*\*kwargs)

12 3

在传递字典kwwrgs时,字典的每个键值对作为一个关键字传递给函数fonco

前面介绍了函数参数的若干种传递方式，这些方式在定义或者调用函数时可以混合使用。 混合使用的基本原则如下：

1. 先位置参数传递。
2. 再关键字参数传递。

(3 )再可变参数传递。

无论参数采用哪种传递方式，需要仔细斟酌。下面给大家看一个例子。

>>> def test (a^b =

.时。 print(az bz d)

|  |  |  |  |
| --- | --- | --- | --- |
| »> | test (1) | # | 将1传递给a, b采用默认值为4,参数c和d为空 |
| 1 4 | 0 {) |  |  |
| »> | test(1,2) | # | 将1传递给a,将2传递给切参数c和d为空 |
| 12、 | .0 {} |  |  |
| »> | test (1A 2 z 3) | # | 将2传递给a和如 将3传递给c\*参数d为空 |
| 1 2 | (3,) {) |  |  |
| »> | test (1z 2 z 3 z 4) | # | 将1、2传递给a和b,将3和4传递给c,参数d为空 |
| 1 2 | (3, 4) {} |  |  |
| »> | test(a = *1)* | # | 将1传递给使用默认值4,参数c、d为空 |
| 1 4 | 0 {) |  |  |
| »> | test(a = 1,b = 2) | # | 将1传递给羽将2传递给3参数c、d为空 |
| 1 2 | 0 {) |  |  |
| »> | test(a = l,b = 2,c = | 3) # | 将1传递给为将2传递给如 将3传递给c,参数d为空 |
| 1 2 | (){ ex 3} |  |  |
| »> | test(1b = 2) | # | 将1传递给迪将2传递给b,参数c、d为空 |
| 1 2 | 0 {} |  |  |
| »> | test(1,2,b = 3) |  |  |
| Traceback (most recent | | call | last): |
| File n<stdin>ur line | | 1f in <module> | |
| TypeError: test () got multiple values for argument !b1 | | | |
| »> | test(1,2,3p 4,x = 1) |  |  |

1 2 (3, 4) {>x?: 1} :

»> test (1,2,3,4, x = 1, y = 2, z = 3)

1 2 (3, 4) {1zJ : 3, 'x\* : 1, !yr: 2) '

上面代码演示了参数不同组合的传递方式，对函数的定义而言，参数的顺序需要遵循下列 原则：

(1 )参数arg=<value>必须位于args后。

(2 )参数 \*args 必须在 arg=<value> 后。

1. 参数\*\*理必须在\*arg后。

如果调用混合传递模式的函数，参数的赋值的过程是这样的：

(I )按照顺序将args实参赋值给对应的形参。

(2 )将args^value形式的实参赋值给形参。

(3 )将多余的arg形式的实参，打包组成一个元组传递给\*argso

1. 将多余args=value形式的实参，打包组成一个字典传递给\*\*argso

:So 4递归画数

在函数内部，可以调用其他函数。如果一个函数在内部调用函数本身，这个函数就是递归 函数。我们以计算阶乘丑! =1 x 2 x 3x…x 〃为例,如果用函数fonc(n)表示,可以写成:

func(n)= n! = 1\*2\*3\* …\*(n-l) \* n = (n-1)! \* n = func(n-l) \* n

依次类推,fanc(n)用递归可以写成下列形式:

def func(n)

if n <= 1：

return 1 return n\*func(n-1)

上述fonc()函数就是一个递归函数。假设,n的值为5,那么这个函数的执行过程如图X所示。

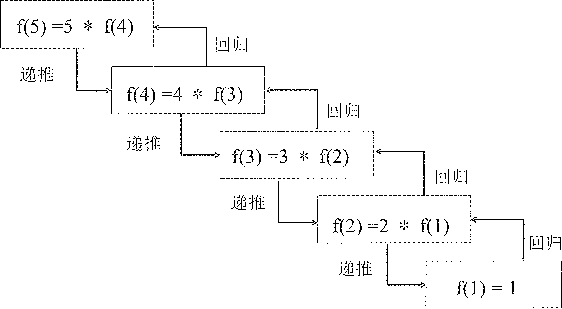


图8；递归过程

从上述递归函数的例子，我们可以看出，递归函数具有如下特征:

1. 递归函数必须有一个明确的结束条件。
2. 递归的递推和回归过程，跟入栈和出栈类似。这是因为在计算机中，函数的调用其 实就是通过栈这种数据结构实现的。每调用一次函数，就会执行一次入栈，每当函数返回，就 执行一次出栈。由于栈的大小是有限的，因此，递归调用的次数过多，会导致栈内存的溢出。

在Python中，函数本身也可以作为参数传入另一函数并进行调用。示例如下：

>>> def func\_a(func):

…. print(func())

>>> def func\_b():

... return 'haha1

>>> func\_a(func\_b)

haha

|  |  |  |  |
| --- | --- | --- | --- |
| 上述代码中,将函数名fonc\_b作为参数传递给 | fiinc\_a f | [def func\_a(Juncj) 「 [ | |
| 函数ftmc\_a(),此时,参数fonc指向的是func\_b函 |  | | |  | print(func()) | 1 1  1 1  !: |
| 数体，具体如图糧4所示。 |  |  | *\ \* |
|  | def ftmc\_b (): | i  1 |
| 此时,fimc指向的是func\_b函数体,如果调用 | func\_b — ~~-► | return 'haha' |  |
| fiinc()函数，相当于执行fimc\_b中的代码，将“haha”  返回,print ( foncQ )相当于 print(?haha?)o | 图泓4 | 函数作为参数传递 |  |

上面的例子须作为参数的函数本身是没有传递参数的，下面看一个有参数的例子，具体如下:

>>> def func\_a(func^ \*argsz \*\*kwargs):

。… print (func (\*args))

>>> def func\_b(\*args):

..« return args

>>> func\_a(func\_br17 2 f 3) dz 2. 3)

上述代码中,将函数fimc\_b作为函数fimc\_a的参数传入,将函数fonc\_b的参数以元组 args的形式传入,并在调用fonc\_b时，作为func b的参数。

8 So 6 JkmAdh匿名函数

lambda()函数用于创建匿名函数。匿名函数就是没有名称的函数，也就是不再使用def语 句定义的函数。lambdaQ函数的语法格式如下所示。

lambda [argl [7 arg2z argn]] :expression '

上述格式中，u[argl [9arg29\_oargn]]n表示的是函数的参数，"expression”是一个表达式,

第8章 函数和函数式编程149 它是函数的返回值，不用写rdum。示例如下：

lambda a7 b: a + b

上述代码创建了一个两数求和的匿名函数，为了保证创建的这个匿名函数不被内存回收， 最好使用一个变量保存，这样，后期可以随时使用这个函数，示例如下:

»> sum = lambda az b: *a + b*

»> sum (1,2)

3

由于切mbda()函数的语法特点，一般只能创建简单的函数，函数的返回值只能是一个对象 或者一个表达式，它不可以像def创建的函数一样，使用if或者版等语句， [就总结：与def相比，lambda创建的函数有很多不同的地方。

(1) def创建的函数是有名称的,而lambda没有函数名称,这是最明显的区别之一°

(2 ) lambda返回的结果通常是一个对象或者一个表达式，它不会将结果赋给一个变量， 而def则可以。

(3 ) lambda只是一个表达式，函数体比def简单很多,而def是一个语句。

(4 ) lambda表达式'':〃后面只能有一个表达式,def则可以有多个。

1. 像if或for等语句不能用于lambda中，def则可以。

(6 ) lambda 一般用来定义简单的函数,而def可以定义复杂的函数。

8o7常關爾數

8o7d1 map 圖灘

map函数会根据提供的函数对指定的序列做映射。

map函数的定义如下:

map(func, \*iterables) -> map object

上述格式中，第1个参数是函数的名称，第2个参数是一个迭代类型，它返回的结果是一 个object类型。

map()函数的作用是以参数序列中的每个元素分别调用fimc()函数，把每次调用后返回的 结果保存到返回值中。

示例如下:

func = lambda x:x+2

result = map(func, [1,2f 3Z 4f 5]) print(list(result))

在上述示例中，定义了匿名函数并赋值给变量fane,接着通过map()函数把序列的每个元 素取出来，作为参数调用fimc()函数，然后把结果放到临曲中。

程序输岀的结果如下:

[3, 4, 5, 6f 7]

冋 [Ep 字 Ep Ep Ep ] function | return x+2 |

newList [「% | 4 5 *6* |7| ]

图8-5 map函数原理图

接下来，通过一张图来描述上述示例执行的 原理歹如图糧5所示。

当传入的函数需要两个参数的时候，同样需 要传递两个列表。示例如下：

result = map (lambda xz y: x+y\* [ 1 p 2"3], [4, 5, 6])

print(list(result))

#结果为

[5, 7, 9]

[亨注意:

f:广f"”

2.7)如下：

在Python 3以前,当传入的函数为None时，相当于合并参数为元组。示例(Python

result = map (None,, [ 1 f 3, 5, 7 f 9] r *[2f 4 r* 6, 8r 10]) print(result)

#结果为

[(1； 2), (3, 4) 」(5, 6), (7, 8), (9/ 10)]

如果两个序列的元素个数不一致，那么元素少的序列会以None补齐。示例(Python 2.7 ) 如下:

result = map(None, [17 3, 5, 7z 9], *[2f* 4, 6]) print(result)

#结果为 [(1/ 2), (3, 4) , (5Z 6) , (7, None) , (9, None)]

需要注意的是,在Python 2中map函数无法处理两个序列长度不一致、对应位置操作数类 型不一致的情况，一旦遇到不一致的情况，程序会报类型错误。

filter()函数会对指定序列执行过滤操作。

filter()函数的定义如下:

filter (function or Nonez iterable) -> filter object

在上述定义中，第1个参数可以是函数的名称或者None；第2个参数指的是可迭代类型， 返回值是一个object类型的对象。这里需要提醒的是，第一个参数如果是function,它只能接 收一个参数，而且返回值是布尔值(True或Fa屁)。

示例如下：

func = lambda x:x%2

result = filter (funcz [ 1 z 2f 3, 4’ 5]) print(list(result))

在上述示例中，定义了匿名函数并赋值给变量fimc,接着通过句翊()函数把序列的每个元 素取出来，作为参数调用foncO函数，然后把结果放到冀suit中。

程序输出的结果为

[1, 3, 5]

fi农r()函数的作用，是以参数序列中的每个 元素分别调用fiinction()函数，最后返回的结果 包含调用结果为True的元素。为了让大家理解 句饨r()函数的作用，接下来，通过一张图来描述 上述示例执行的原理，如图8・6所示。

图8『6分析了上述程序执行的原理。由图8-6 可知,程序把原始序列的元素执行取余操作以后， 将不能被2整除(结果为True)的元素筛选出来

seq function filter newList

[M E E Ep [p ]

I return x%2 |

I ~~I^e ]

tS 5 5]

图8-6 map函数原理图

构成一个新的列表。

*B.7.3*

reduceO函数会对参数序列中的元素进行累积。

reduce()函数的定义如下:

reduce(function, sequence[, initial]) -> value

在上述定义中，function()是一个带有两个参数的函数；第2个参数可以是序列、元组、 字符串；ini飼 表示固定的初始值。reduce会依次从sequence中取出每个元素，和上一次调用 function的结果作为参数再次调用function0

需要注意的是，ftmctionQ函数不能为Noneo在Python 3中reduce函数已经被从全局名字 空间里面移除了，它现在被放置在fonctools模块中，使用时需要先引入如下格式：

from functools import reduce

下面是Python 3中reduce函数的用法。

from functools import reduce

func = lambda xzy: x + y

result = reduce(func^ *[lf 2,* 3Z 4, 5])

print(result)

在上述示例中定义了匿名函数fonc(),接着通过reduce函数把序列的每个元素取出来,和 上次调用的结果都作为参数调用fonc()函数，最后把结果放在result中，运行结果如下所示。

15

如果在第一次调用fonction函数时为其提供了 initial参数，那么函数会以sequence中的第 一个元素和initial作为参数进行调用,在Python2.7中的示例如下:：

from functools import reduce

result = reduce(lambda x, y: x + y, [1, 2, 3/ 4]f 5) print(result)

结果如下所示。

15

sequence可以是字符串类型,示例如下:

from functools import reduce

result = reduce(lambda x7 y: x + *yr* [ \* aaf 7 \*bb1z 'cc']« J ddJ) print(result)

结果如下所示。

ddaabbcc

So 8变輦作腳域

8n8J 叠屬釁囊靈晨離變聂

变量的作用域始终是Python学习中一个必须理解和掌握的环节,下面我们从局部变量和全 局变量开始全面解析Python中变量的作用域。

所谓局部变量，指的是定义在函数内的变量，即定义在def函数内的变量，只能在def函 数内使用，它与函数外具有相同名称的其他变量没有任何关系。不同的函数，可以定义相同名 字的局部变量，并且各个函数内的变量不会互相影响。示例如下：

»> def test\_01 ():

..o num = 100

print (' test\_01 中的 number 值为 %df % number)

»> def test\_02 ():

... number = 200

…• print ( 1 test\_02 中的 number 值为 %d' %number)

>>> test\_01() test\_01 中的 number 值为 100

>>> test\_02() test\_02 中的 number 值为 200

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。全局 变量是定义在函数外的变量，它拥有全局作用域。示例如下：

result = 100 #全局变量

def sum(a,b):

result = a+b # 局部变量

print (\*函数内的result的值为f , result) #result在这里是局部变量 return result

#调用sum函数

sum(100f 200)

print ( !函数外的变量result是全局变量,等于1 , result)

输岀结果为:

函数内的result的值为300

函数外的变量result是全局变量，等于100

8b8d2 global fl nooBocaB 美纜寥

当内部作用域想修改外部作用域的变量时，就要用到global和nonhsl关键字了。下面分 别对global和nonlocal的作用进行介绍。

t global

global关键字用来在函数或其他局部作用域中使用全局变量。但是如果不修改全局变量也 可以不使用global关键字° 7K例如下:

»> a = 100

>>> def test ():

".• a+=100

... print(a)

>>> test ()

Traceback (most recent call last):

File "<stdin>° r line 1f in <module>

File "<stdin>n, line 2, in test UnboundLocalError: local variable \* a1 referenced before assignment

上述程序报错，提示“在赋值前引用了局部变量a”。但是，前面我们明明是可以在函数 中访问全局变量的呀？那这里为什么会出错呢？

这就是Python与其他语言的不同之处了。在Pythoii中，如果在函数内部对全局变量a进 行修改? Python会把变量a当作是局部变量，而在进行％+=100”之前，我们是没有声明局部 变量a的，因此，程序会出现上述错误提示。

为了使全局变量生效，我们可以在函数内使用global^®字进行声明，下面，我们对程序 进行修改，示例如下：

»> a = 100

»> def test (): ... global a

a+=100

... print(a)

»> test ()

200

此时，程序不但没有报错，而且成功对全局变量a进行了修改。

2. nonlocaS 关譏字

nonlocal是在Python3.0中新增的关键字,python2.x不提供支持。使用nonlocal关键字可 以在一个嵌套的函数中修改嵌套作用域中的变量。接下来,看一个例子：

>>> def func ():

o。. count=l

•.. def func in():

count = 12 func\_in() print(count)

»> func()

1

上面的程序中，在嵌套的函数中，对变量count赋值，同样会创建一个新的变M 而非使用count = 1语句中的count,如果要修改嵌套作用域中的count,就要使用nonlocal关键 字，示例如下：

>>> def func():

o o. count = 1

... def func\_in():

.。。 nonlocal count

... count = 12

... func\_\_in ()

..。 print(count)

>>> func ()

12

上述代码中，在fonc\_in()函数中使用了 nonlocal关键字，就会告诉Python在func函数中 使用嵌套作用域中的变量count,因此对变量count进行修改时，会直接修改嵌套作用域中的 count变量，程序最后也就输出12 了。

*〔食*注意：使用global关键字修饰的变量之前可以不存在，而使用nonlocal关键字修饰的变量 在嵌套作用域中必须已经存在。

*1☆多*学一招：LEGB原则

*''I' '-J- 1 ”*

Python中，程序的变量并不是在哪个位置都可以访问的，访问权限决定于这个变量是在哪 里赋值的。我们先来看一段代码：

»> a = 10

>» def test ():

... a = 2 0

…• print (! a 的值是 %d1 %a)

>>> test ()

a的值是20

上述代码有两个变量a,当在test函数中输出变量a的值时,为什么输出的是20,而不是 10呢？其实，这是因为变量作用域不同导致的。

变量的作用域决定了哪一部分程序可以访问哪个特定的变量名称。Python的作用域一共有 4种，分别是： ，

° L (Local)函数内的区域，包括局部变量和形参。

0 E (Enclosing)外面嵌套函数区域，常见的是闭包函数外的函数。

。G ( Global)全局作用域。

。B ( Built4n)内建作用域。

Python中变量是采用L->E->G->B的规则查找，即Python检索变量的时候,先是在局 部变量中查找，如果找不到，便会去局部外的局部找(例如闭包)，再找不到就会去全局作用域找, 再者去內建作用域中找。

;8o9 「

8.9J圍數齣則麗

先来看一段代码，具体如下《

>>> def func():

print ( \* hello world?)

>» func () hello world.

»> func

<function func at 0xl01cb4el8>

从上述代码可以看出，使用和“fimc”大相径庭。这是因为ftmc()表示调用 fonc()函数,程序会执行函数中的代码。而“fimc”是一个变量,它引用的是一个函数块，func 在内存中的引用方式如图寥7所示。

既然ftmc是一个对象a那么它就可以在不调用的时候使用，我们将firnc赋值给一个新的 函数名fun\_new ( func\_new=func ),那么fiin new指向的也是func指向的函数块，具体如图8-8 所示。

fimc —

def func(a,b): return a+b

def fimc(a,b):

return a+b

fimc

fim\_new

图&7函数的引用 图8-8将feme赋值给一个新的函数名

此时,func new和feme指向的是同一个函数块,如果使用func\_new()调用函数应该也是 可以的，示例如下:

>>> func\_new = func

>>> func\_new

<function func at 0xl01cb4el8>

>>> func\_new()

hello world

竄凱2特遂、

闭包的概念比较难以解释，这里我们通过一个例子来说明。

def test(number\_one):

def test\_in(number\_two):

print(number\_one+number\_two)

return test\_in

上述代码中,test\_in函数就是一个闭包，它满足下列三个条件：

1. 嵌套在函数里面，test函数嵌套了 test\_in函数。

(2 ) test\_\_in中的变量是外部函数test的参数nuniber\_oneo

(3)外部函数test的返回值是内部函数test\_in的引用。

这里，我们调用一下te砒()函数，看看结果是什么样的。

>>> test(100)

<function test.<locals>.test\_in at OxlOlcd?9d8>

可能有的读者会感到诧异，为什么没有将数值计算的结果返回呢？这是因为执行test(lOO) 时，程序只调用了翊t()函数，并没有调用testjn()函数。而fest函数的返回值是函数test\_in的 引用，所以结果也就是一个引用值。调用翊f函数的过程如图&9所示。

test(lOO)-—调用\*

def test(number\_one)

def test\_in(number\_\_two) prmt(n.iimber\_\_one+nimiber\_two) return test in

返回

X

图8-9调用test()函数

如果希望调用test\_in()函数，输出print语句中的结果，可以将test()函数返回的引用赋给 一个新的变量，然后使用()调用，示例如下：

>>> test\_new = test (100)

>>> test\_new(100)

200

上述代码的函数调用过程，如图8J0所示。

test\_new=test( 100)—调用 test- test\_new(100)—调用 test\_in-

def test(number\_one):

a def test\_in(number\_two): print(number\_one+number\_two) return test\_in

图8」0闭包函数调用过程

假设有一个方程式*y=ax+b,*如果编写函数计算\*的值，没有闭包的话，我们需要在创建函 数的时候，传入3个参数，分别表示厶如旳 这种做法不仅需要传递较多的参数，而且代码 的可移植性也不太好。

学习了闭包，我们就可以使用闭包来完成功能，具体代码如下所示。

»> def line\_conf (a,b):

... def line(x):

•。o return a\*x+b

... return line

»> line\_one = line\_conf (1,3)

»> line\_two = line\_conf (4,5)

>>> line\_one(5)

8

>>> line\_two(5)

25

上述代码中，函数line\_conf是外部函数，a、b是函数line\_conf的参数。函数line是内部函数, *x*是函数line的参数。在line内部，我们通过line\_conf的参数a, b确定了闭包的最终形式(\*=\*+3 和戶心),我们只需要交换参数a, b,就可以获得不同的表达式,由此,我们可以看到,闭 包具有提高代码可复用性的作用。

8=10類歸器

编者之前在知乎上看到一个很形象的比喻来描述装饰器，它是这么说的:

“内裤可以用来遮羞，但是到了冬天没法为我们防风御寒，聪明的人发明了长裤，有了长 裤后就不会再冷了。装饰器就像是这里说的长裤，在不影响内裤作用的前提下，给我们的身体 提供了保暖的功效。”

读完上面的句子，不知道大家有没有体会到装饰器的好处？好了，下面言归正传，我们开 始装饰器的学习吧。

BJOJ ft遂矗曩鑑器

装饰器本质是一个Pythoii函数，它可以在不改动其他函数的前提下，对函数的功能进行 扩充。通常情况下，装饰器用于下列场景：

1. 引入日志。
2. 函数执行时间统计。
3. 执行函数前预备处理。
4. 执行函数后清理功能。

(5 )权限校验。

(6)缓存。

先看一个简单的例子。

def func\_one():

print(1func one \*)

现在有一个需求，希望可以在函数中输出一句话，用于记录函数正在执行，这时，有人会 这样实现：

def func\_one(): print(\* func\_one \*) print(1func\_one is runningf) func\_one()

但是,如果函数ftmc\_two()、函数fonc\_three()都有类似需求,那么现在这样的做法会出现 大量重复的代码。为了减少代码重复，我们可以创建一个新的函数专门记录函数执行日志，示 例如下:

def print\_logo():

print ('函数正在运行中')

def func\_one():

print(\* func\_one1) print\_logo()

上述代码虽然可以实现功能，但是却破坏了原有代码的逻辑结构。如果要求已经实现的函 数，不能修改，只能扩展，即遵守“封闭开放”原则，那么是不允许在函数fonc„one内部进行 修改的。

装饰器可以满足上述需求。在Python中，装饰器的语法是以@开头,下面,我们写一个 简单的装饰器。

def wrapper(func):

print ( 1正在装饰\*)

def inner():

print ( f正在验证权限') func ()

return inner

©wrapper

def test():

print"test"

test ()

下面我们来分析一下程序的执行过程:

1. 当程序执行test()时,发现函数test()上面有装饰器@wrapper,所以会先执行@ wrapper。@wrapper 等价于 test=wrapper(test), 它可以拆分为两步:
2. 执行wrapper(test)9将函数名test作为参数传递给wrapper。在调用wrapper函数的过程中, 首先会执行print语句,输出“正在装饰",然后会将形参fdnc指向test()函数体,并将inner() 函数的引用返回给wrapper(test),作为wrapper(test)的返回值。具体如图841所示。
3. 将wrapper(test)的返回值赋给於st,此时须fest指向inner。函数。如图8-12所示。

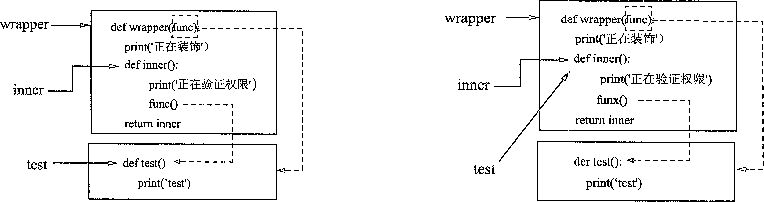


图 8-11 执行 wrapper(test) 图 8-12 执行 test=wrapper(test)

到此,我们完成了函数test()的装饰。

(2 )调用fest()指向的函数。因为test指向的是imier()函数，所以此时，调用化必)函数 相当于调用inne『()函数，输出过程如下：

1. 输出print语句“正在验证权限”。
2. 调用fimc指向的函数体，输出"test"。

観10 口2鈴令赣鱸器 •

多个装饰器可以应用在一个函数上，它们的调用顺序是自下而上的。 下面给大家举例演示。

def wrapper\_one(func):

print ( ?--正在装饰 1-- \* )

def inner():

print (，一正在验证权限1— 1 )

func()

return inner

def wrapper\_two(func):

print ('正在装饰2--')

def inner():

print ('--正在验证权限2— \* )

|  |  |  |
| --- | --- | --- |
| func()  return inner | wrapper\_onRT | def wrappei\\_oiiG (func): |
| @wrapper\_one | inner \_\_\_\_\_ | print('—正在.羨饰1」)  F>dcf inner(): |
| @wrapper\_two |  | printC—正在.验证 ') |
| def test (): |  | iunc() |
| print ( ' test ')  #调用test,调用test之前,已经装饰过了 fl() |  | return inner |
| wrapp\_two — | clefwrapper\_;hvo (tunc):  卩而正在装饰2」) |
| 上述代码中，恍st()函数上面有两个装饰器，按照多个装 | inner-—J | —^def iiineiX):  prin.t(f-正在脸毎杈限2”') |
| 饰器从下往上的调用顺序,程序首先会执行@wrapper\_two, 再执行@wrappH\_one。下面，\_步\_步来分析程序的执行过程。  (1 )程序将 wrapper\_one、wrapper two» test 函数加载 到内存中,加载完成后如图8-13所TKo |  | func()  return inner |
|  | def test():  printf—test-—1) |
| 图843 |
| (2)testQ函数上有两个加载器，首先程序会执行@ | 将函数加载到内存 |

wrapper two,也就是test=wrapper\_two(test),执行完毕后的内存如图8-14所示。

(3)执行@wrapper\_one,也就是test=wrapper\_one(test),这个过程执行完毕后,其实, 装饰器对函数的装饰已经完成，会输出下列语句：

一正在装饰2 —

一正在装饰1 —

装饰后的函数关系如图845所示。

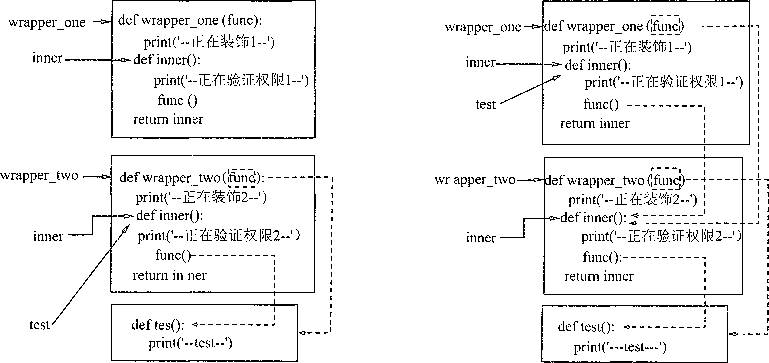


图8-14执行@w2

图8-15执行@wl

(4)装饰完毕后，如果使用"顷()”语句调用坨砒函数，程序会按照从上向下的顺序运行, 最终程序的输出结果如下：

—正在验证权限1 —

--正在验证权限2 —

test

・前面我们介绍的装饰器，都是对无参数的函数进行装饰，如果要对有参数的函数进行装饰, 那么参数如何设置呢？下面先看一段代码：

>>> def wrapper(func):

.•. def inner():

…“ print ( \*开始验证权限, )

.。“ func ()

... return inner

>>> ©wrapper

.。«. def test (azb):

…• print('a = %d,b = %d'%(a,b))

»> test (lf2)

Traceback (most recent call last):

File "<stdin>", line 1r in <module>

TypeError: inner() takes 0 positional arguments but 2 were given

上述代码在执行“翊t(l,2)”时报错，提示inner()函数不需要参数，但是，我们在调用的时候， 传递了两个参数。下面我们同样画图来分析。

())执行©wrapper,等价于test=wrapper(test)9执行完毕后如图8-16所示。

1. 执行代码utest(l, 2)”，此时test指向的是inner()函数,程序会执行inner()函数 中的代码。可是，创建的函数是不需要参数的，而我们调用test时传入了两个参数，因 此程序会报错，提示不需要参数，但我们传入了两个参数。

下面对imier()函数进行修改，在创建inner()函数时添加两个参数，再次调用test()函数， 具体代码如下：

»> def wrapper (func):

e. def inner (azb):

…。 print ( 5开始验证权限\* )

... func()

o o。 return inner

>>> ^wrapper

.•。def test (a^b):

... print ( 'a = %dfb = %d \* %(az b))

»> test (1,2)

开始验证权限

Traceback (most recent call last):

File "<stdin>?,, line 1, in <module>

File n<stdin>° r line 4 f in inner

TypeError: test () missing 2 required positional arguments: \* a \* and 1b\*

程序还是报错，同样是缺少函数参数引起的错误。这是因为，在击壶此函数内部执行 fon()语句时,fonc()缺少两个参数,具体如图8-17所示。

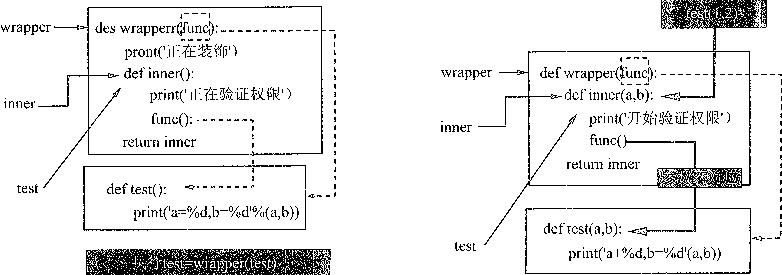


图8-16执行@wl

图8-17程序报错原因分析

再次修改inner()函数,在调用fonc()时,将a、b参数传入,示例如下:

>>> def wrapper(func):

..。 def inner(a^b):

…. print ( f开始验证权限1 )

... func(a7 b) 、

o.. return inner

>>> @wrapper

.•. def test(a^b):

。… print ( ? a = %d7 b = %d1 % (az b))

>>> test (17 2)

此时，程序输出结果如下：

开始验证权限 a = 1, b = 2

上面学习的装饰器只是对两个参数的函数适用，如果无法确定函数的参数个数以及参数类 型，我们可以使用可变参数来传递，示例如下：

>>> def wrapper(func):

”\*。 def inner(\*argsf \*\*kwargs)):

…。 print ( \*开始验证权限1 )

。… func(\*args f \*\*kwargs))

。.. return inner

>>> ©wrapper

…" def test (\*argsz \*\*kwargs):

,•。 print ( ? test J )

此时，调用不同参数的函数，发现装饰器适用于不同参数的函数，示例如下：

»> test ()

开始验证权限

>>> test(1,2,3)

开始验证权限

>>> test(a=lf b=2 c=3)

开始验证权限

8J0.4

前面介绍的装饰器9都是对不带返回值的函数进行装饰，如果要对有返回值的函数进行装 饰，那么该如何实现呢？先来看一段代码：

»> def test ():

…。 return \* itheima 5

>>> result = test ()

>>> result 'itheima1

上述代码创建了一个test()函数，并使用result变量保存调用test函数后的返回值。此时， 如果我们对带有返回值的函数进行装饰，那么，调用test函数后，函数的返回值还能输出吗？ 下面编码来尝试。

»> def func (fuction\_\_name):

*。.*。 def func\_in():

... fuction\_name()

..• return func\_in

>>> @func

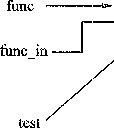
«, o . def test ():

• . . return 1itheima

>>> result = test ()

>>> result

程序执行后，发现此时调用test()函数，函数没有返回值，也就是None。这是因为，当使



用@fiinc对test函数装饰后,test指向了 func\_in() 函数，而函数本身是没有返回值的。程序执 行情况如图8』8所示。

在图8J8中，test指向了新的函数，原本tesfO函 数的返回值被fimction\_name接收了。因此，如果想输 出"itheima" 9需要使用Tefum语句将调用后的结果返 回。对装饰器进行修改，修改后的代码如下所示。

def func(fcniction\_name)

def testQ:

return 'itheima'

图8J8函数调用情况分析

def func(function\_name):

def func\_in():

return function\_name()

return func\_in

此时，再次调用性砒()函数，发现成功得到了返回值。

>>> result = test ()

>>> result

'itheima \*

8J0.5

通用装饰器，简单来说，就是既能对无参数、有参数的函数进行装饰，也能装饰无返回值、 有返回值的函数。基于前面对装饰器的了解，这里我们直接看一个通用装饰器的案例/

def func(function\_name):

def func\_in(\*args,\*\*kwargs):

print ( \*正在装饰, )

return function\_name(\*argsf \*\*kwargs)

return func\_in

该装饰器可以对不同类型的函数进行装饰，具体如下：

(1)装饰无参数、无返回值的函数，示例如下：

>>> @func

..。def test\_one():

*.。*• print('hello itheima\*)

>>> test\_one()

正在装饰

hello itheima '

1. 装饰无参数，有返回值的函数。示例如下:

>>> @func

.。.def test\_two():

... return 5 hello itheima1

»> test\_two ()

正在装饰

\* hello itheima\*

1. 装饰有参数，无返回值的函数。示例如下:

>>> @func

…。def test\_three (arb):

。… print ( 5 *a* = %df b = %d1 % (af b))

>>> test\_three(10f 20)

正在装饰~

a = 10,b = 20

1. 装饰有参数，有返回值的函数。示例如下:

>>> @func

…。def test\_four (af b):

... return a+b

>>> test\_four(10,20) 正在装饰

30

8J0.®

前面讲解的装饰器，都是不带参数的，这些装饰器最终返回的都是函数名。如果给装饰器 添加参数，那么需要增加一层封装，先传递参数，然后再传递函数名。下面看一段代码:

>>> def func\_arg(args):

.•. def func(function\_name):

。.。 def func\_in():

*…*. print (1 --记录日志 -args = %s \* %args)

... function\_name()

.•。 return func\_in

.昨 \* return func

»>

@func\_arg(\* haha!)

def test.():

print (" test——-\*)

»> test ()

一记录日志-args = haha

test-

上述代码中,u@func\_arg('haha\*)装饰函数test,等价于:

test = func\_arg('haha ?) (test)

由于函数fiinc\_arg的返回值是fonc函数的引用，也就是函数名ftmc，因此，上述代码等价于：

test = func (test)

看到这行代码，大家应该很熟悉了，这就是前面我们学习的无参数的装饰器。相比无参数 的装饰器，带参数的装饰器只是用来“加强装饰”的，如果希望装饰器可以根据参数的不同罗 对不同的函数进行不同的装饰，那么带参数的装饰器是个不错的选择。

8oll生戚器

BJiJ tt忽暴鎚甌器

在讲生成器之前，我们先看一个例子。

>>> a = [x\*2 for x in range (10)]

»> a

上述代码中，a是新创建的一个列表，但是，受内存限制，列表的容量肯定是有限的。如 果创建一个包含一千万个元素的列表，例如：

a = [x\*2 for x in range(100000000)]

而我们仅仅需要访问前面若干个元素，那么该列表不仅会占用很大的存储空间，而且绝大 多数元素占用的空间都白白浪费了。

其实有这么一种机制，它可以在序列循环时，只推算出我们需要的后续元素，而不必创建 完整的序列，这样可以节省大量空间，这种一边循环一边计算的机制，叫作生成器。

生成器(generator)主要目的是构成一个用户自定义的循环对象。它可以看做是一个带有 yield的函数，yield是一个关键字，一旦函数被yield修饰，Pythoii解释器会将被修饰的函数看 做是一个生成器。一个生成器中可以有多个yield。当生成器遇到一个yield时，会暂停运行生成器， 返回yield后面的值。当再次调用生成器的时候，会从刚才暂停的地方继续运行，直到下一个 yield o

下面使用生成器实现斐波那契数列。斐波那契数列指的是这样一个数列L L 2, 3, 5, & 13, 21, 34,…，也就是说，数列的前两项都为1,第3项开始，每一项都是前两项之和。 代码如下：

>>> def creat\_number():

az b = 0 f 1

... for i in range(5):

*..。* yield b

…。 *afb = bf a+b*

»> list (creat\_number ())

[1, 1, 2, 3, 5]

上述代码中，creat\_number就是一个生成器，它可以获得指定范围内的斐波那契数列，感 兴趣的读者可以尝试输出其他范围的数列。

8J1.2鎚咸器遑行飜誡

函数一旦包含了 yield,那么它就不是一个普通的函数，而是一个生成器。与普通函数返回 rduni语句后的值不同，生成器返回的是一个对象。如果我们希望得到生成器的值，需要调用 next函数或者访问 \_next\_方法(Python 3新增的)。

为了更加明显看出生成器的运行情况，下面在生成器creat\_number中增加若干行输岀语句, 具体代码如下：

>>> def creat\_number():

•… a, b = 0,1

*。*。. for i in range(5):

-… print('~~1一 ?)

-。. yield b

.。» print(,--2-- 1 )

…。 azb = b7 a+b

..。 print('一3— ')

…。 print ( \* 一stop-- ?)

使用next()函数获得生成器creat\_number的值，示例如下: 第1次调用9此时a,b=0,l

>>> result = creat\_number()

>>> next(result)

1

1 : . .「

第2次调用，此时a,b=b5(a+b)=l,l

>>> next(result)

——2——

一一3——

1

第3次调用，此时，此时a5b=b5(a+b)=l,2

>» next (result)

——2——

—3 —

第4次调用?此时a,b=b,(a+b)=2,3

>>> next(result)



第5次调用,此时a,b=b,(a+b)=3,5

>>> next(result)

——2 — 一



第6次调用，循环结束。

>>> next(result)

2

——3——

——stop——

Traceback (most recent call last):

File "<stdin>"r line 1, in <module>

StopIteration

不知道大家有没有看出规律？当调用next函数向生成器索要值时,生成器运行到yield之处， 返回yiEd后面的值且在这个地方暂停，所有的状态都会被保持住，直到下次next函数被调用， 或者碰到异常循环退岀。

8J1n3 yisBd .

在生成器中，如果没有免turn,则默认执行到函数完毕时返回Stoplterationo这一点特征在 讲生成器运行机制的时候，已经有所体现。但是，如果在生成器中遇到return,那么程序会直 接抛出Stopiteration异常终止迭代。示例如下:

>>> def test ():

n=0

••。 while n<10:

。。。 yield n

... n+=2

..• return

上述代码中，test是一个生成器，在生成器的末尾有一个return语句，此时，使用next函 数获取生成器的值，程序会抛岀Stopiteration异常。示例如下：

»> result=test () >>> next(result) 0

>» next (result)

#程序停留在"yield n°处，返回n的值

# 发现 return 语句,抛it Stopiteration 异常,这样 yield #语句不会再被执行

Traceback (most recent call last):

File °<stdin>", line 1f in <module>

StopIteration:

如果在冀turn后返回一个值，那么这个值为StoplMatig异常的说明，不是程序的返回值。 示例如下:

>>> def test ():

.… n = 0

..“ while n<10:

*。*。 yield n

。… n += 2

..° return '异常 ,

»> result = test ()

>>> next(result)

0

»> next (result)

Traceback (most recent call last):

File °<stdin>°, line 1, in <module>

Stopiteration:异常

箜熾麗蕊躊齢磨IS

生成器对象是一个迭代器。但是它比迭代器对象多了一些方法，包括send()方法，throw。 方法和c1osr()方法。这些方法主要是用于外部与生成器对象的交互，具体介绍如下：

IL send()

send()方法在一定意义上和next()方法的作用是相似的，区别在于，ssd()方法需要传递 一个参数,而next()方法不能传递特定的值,只能传递None。下面,我们通过一个例子来分析 send()和next()方法的区别,具体如下:

>>> def test (): 、

…。 n = 1

。。» while n<10:

*。*•. temp=yield n

..print (temp)

.。. n+=l

使用next获取生成器中的值，代码如下：

>>> result = test ()

»> next (result)

1

>>> next(result)

None

2

>» next (result)

None

3

从上述代码可以看出，“temp = yield iT返回的值是None。

下面使用send()方法来获取，具体代码如下:

>>> result.send(\* haha1)

Traceback (most recent call last):

File °<stdin>°, line 1f in <module>

TypeError: can11 send non-None value to a just-started generator

程序报错，这是什么原因呢？这是因为第一次调用send时，send()方法传递的参数没有 yield语句来接受这个值，因此，我们不能使用send发送一个非No眼的值，否则会出错。

明确了出错原因后，可以在第一次调用的时候，使用next或者send(None)来执行，示例 如下：

»> next (result) # 也可以使用 send (None) , result.next ()等价于 result. send (None)

1

>>> result.send(\*haha1)

haha

2

>>> result.send(\*heihei')

heihei

3

2. closeO 方■法 ,

当一个生成器在永远执行的时候(while True的时候),就用到了 close()来终止它。示例 如下：

»> def test ():

... while True:

°。® yield 5value-01'

* °。 - yield ? value-02 \*

°。。 yield \* value-03 J

* \* yield. ? value-04 1

此时，调用close。方法终止生成器运行，示例如下：

>>> result = test ()

»> next (result)

5 value-01 \*

>>> result.close()

>>> next(result)

Traceback (most recent call last):

File "<stdin>H, line 1, in <module>

Stopiteration

程序执行过程如下：

(1 )执行 next(result),输出"value-01",并停留在 yield' value-02'之前。

(2 )执行result.close()终止生成器。

(3)再次执行next(result)?程序报错，因为目前生成器test已经被终止了。

3. throwOBil . :

throw。方法主要是向生成器发送异常，从而使生成器能够捕获异常，示例如下：

>>> def test ():

.。。 while True:

... try:

。。。 yield \* value-01!

yield , value-02 ?

。.. except:

.<,. print (\* the exception is here 1 )

此时，调用血ow()方法向生成器发送异常须示例如下：

>>> result = test()

>>> next(result)

1 value-01'

»> result o throw (ValueError)

the exception is here

'value-01!

程序执行过程如下：

(1 ) next(result):会输出 value-01,并停留在 yield' value-02\* 之前。

(2 )执行 resultlhrow(ValueError),会跳过所有的 try 语句,也就是说,yield5 value-021 不 会执行,直接进入到except语句,打印出uthe exception is heren ,同时,会消耗一个yield, 输出 “value-01” o

8J1S雙鐵器廬屬博赣——侧雑 —，

在函数调用时，一般都是从函数的第1行代码开始执行歹直到遇到Mtum语句、异常或者 函数结束才终止，这是之前学习的很标准的一个流程。

生成器的强大功能之一就在于它提供了协同程序，协同程序是可以运行的独立函数调用须 可以暂停或者挂起，并从程序离开的地方继续或者重新开始。同时调用者也可以向程序传入额 外的数据或者异常等，传入完毕后仍能在上次暂停的地方继续执行。

下面看一个例子，代码如下：

def test\_one():

while True:

print(v 1-~~ \*)

yield None

def test\_two():

while True:

print(f 2\_\_\_ \*)

yield None result\_one = test\_one result two = test two

while True:

resultt\_one next () result two next ()

第8章函数和函数式编程w 171 上述程序运行后，程序的执行情况是，交替循环输出下列结果：

在Python中，很多对象都可以通过f成语句直接遍历的，如列表、字符串、字典等，这些对 象都可以称为可迭代对象。至于哪些对象可以被迭代访问，这时就要了解迭代器的相关知识了。

迭代器对象要求支持迭代器协议，即实现对象的\_next\_()和\_iter\_()方法。其中， \_ita\_()方法返回迭代器对象本身身\_next\_\_()方法返回容器的下一个元素，在结尾时引发 Stopiteration异常。对于可迭代对象而言，可以使用内建函数ifer()获取它的迭代器对象。

下面看一个示例，代码如下：

>>> array = [ 1F 2 z 3]

>>> iter = iter(array)

»> iter

<list\_iterator object at 0xl020e6908> >>> iter. next ()

1

>>> iter. next ()

2

>>> iter. next ()

3

>>> iter. next ()

Traceback (most recent call last): File n<stdin>°, line 1, in <module> Stopiteration

在上述示例中，通过iter()方法获得了列表的迭代器对象，调用\_next\_()方法访问列表 的元素。当没有可访问的元素后，抛出Stoplteration异常终止迭代器。其实，当我们使用&语 句时，它会自动通过iter()方法获得迭代器，并且通过方法获取下一个元素。

实际上，每个生成器都是迭代器，通过调用具有一个或者多个yi"d表达式(在Python 25 及更早版本中的yield语句)的函数来构建生成器，同时满足迭代器的定义。

生成器是迭代器的子类型。接下来，借助于issubclass(sub? super)和i前顽ance()两个函数 进行验证，前者用来判断sub类是否为supe『类的派生类，后者用来判断一个对象是否是一个 已知的类型。

具体如下:

import collections import types def gen():yield

»>

»>

»>

>>> simple\_gen = gen()

»> isinstance(simpl㊀collections.Iterator)

True

>>> issubclass(types.GeneratorType, collections.Iterator)

True

上述示例中，直接定义了一个简单的生成器，并使用isinstance()函数查看这个生成器是不 是Iterator类型，再使用issubclass()函数查看生成器是否为迭代器的子类。通过上述结果可以 看出，生成器是一种特殊的迭代器。

go 12 标維库——内置函戮

Python内置(built-in)函数随着Python解释器的运行而创建，它们总是可用的。Python 内置的函数，按字母顺序排列的清单如表8・2所示。

表8-2 Python内置函数清单

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abs() | delattrQ | hash() | memoryview() | set() |
| all() | dict() | help() | min() | setattr() |
| any() | dir()divmod() | hex() | next。 | slice() |
| ascii() | divmod() | id() | object() | sortedQ |
| bin() | enumerate() | input() | oct() | staticmethod() |
| bool() | eval() | int() | open() | str() |
| breakpoint() | exec() | isinstanceQ | ord() | sum() |
| bytearray() | filter() | issubclass() | pow() | super() |
| bytes() | float() | iter() | print() | tuple() |
| callableQ | fbrmat() | len() | property() | type。 |
| chr() | frozenset() | list。 | range() | vars() |
| classmethod | getattrQ | locals() | repr() | zip() |
| compileQ | globalsQ | map() | reversed() | \_\_import\_\_() |
| complexQ | hasattrQ | max() | roundQ |  |

上述内置函数中，很多函数我们已经学过，如priirt()、tiipk()等，这些内置函数是随着 Python解释器运行而创建的。在编写Python程序时,我们不需要进行定义,就可以随时调用这 些函数。

8J2J

Python内置函数的功能非常强大，在前面的学习中，我们已经学习过一些函数，这里，我 们反过头来看看之前学习过的内置函数。

•查看数据类型的函数:type()o

•查看帮助信息的函数：help()o

•获取集合长度的函数：len()o

•循环设计的相关函数:range() enumerate()o

•函数对象的相关函数:map()、filterQ^ reduce()o

®和输入输出相关的函数:input()、print()o

•不同进制转换的相关函数：bin()、hex()、oct()o

•和序列相关的函数:list。、tuple。、dict(). sort(). reverse()o

•整数之间转换的函数：int()、float(). str()、complex。。

8J2.2數寥遶寡帼美圖鐡

和数学运算相关的函数如下所示。

1 “ abs(x)

返回数字的绝对值，参数可以是整数、浮点数或复数。示例如下：

>>> abs(-5)

5

>» abs (3.14)

3.14

>>> abs(8+3j)

8.54400374531753

需要说明的是，如果参数是一个复数，那么匆bs()函数返回的绝对值是此复数与它的共轲 复数的乘积的算术平方根。

2" round(number[5 ndigits])

nnmd()函数用于对浮点数进行四舍五入求值，具体保留几位小数，以传入的ndigits参数 来控制。示例如下:

>>> round(1o134567,1)

1.1

>>> round(1。134567《4)

1.1346

ndigits参数为可选参数，当不传入时，即以默认保留0位小数进行取整，返回的是整数。 示例如下:

>>> round(1.134567)

1

ndigits参数传入0时,虽然与不传入ndigits参数一样保留0位小数进行四舍五入,但是返 回的值是浮点型。示例如下：

>>> round(1»134567^0)

1.0

nnind四舍五入时是遵循靠近0原则须所以-(X5和d5进行0位四舍五入，返回的都是0, 示例如下:

>>> round(0.5)

0

>>> round(-0.5)

0

对于浮点数求四舍五入有一个陷阱,有些四舍五入结果不像预期那样，比如round(3.565?2) 的结果是3.56而不是预期的357,这不是bug，而是浮点数在存储的时候因为位数有限，实际 存储的值和显示的值有一定误差。示例如下：

>>> round(3.565z 2)

3o56

对整数也能进行nnmd操作，返回值也是整型。示例如下:

>>>

4

»>

4

»>

4

»>

0

round(4)

round(4,3)

round(4 f 0)

round(4,-2)

3a pow(x5 y[9 z]) :

该函数的参数x、y是必选的，z是可选参数。返回的结果是x的y次蓦乘(相当于x\*\*y), 如果可选参数Z有传入值，则返回藉乘之后再对Z取模(相当于pow(x9y)%z) o示例如下：

>>> pow(37 3)

27

>>> pow(3,3,4)

3

所有的参数必须是数值类型*，*

否则会报错。示例如下:

>>> pow(27 12 s)

Traceback (most recent

File "<stdin>"r line

TypeError: unsupported

call last):

1f in <module>

operand type(s) for \*\* or pow() : \*int1 and \* str 1

如果参数x, y中有浮点数，则结果会被转为浮点数。示例如下:

»> pow (2,3.2) 9.18958683997628

>>> pow(2.1^2)

4.41

如果参数x、y都是整型须那么返回的结果也是整型，除非其中一个参数是负数，返回的 结果是浮点数。

>>> pow(2,3)

8

>>> pow(2z-3)

0.125

需要提醒的是，因为浮点型不能进行取模运算，所以'，如果返回的结果是浮点数，那么必 须省略参数Z。如果Z存在，那么X和y必须是整数，且参数y不能为负数。

4. divmod(a? b)

除法运算，参数a和b是非复数，返回的是除法结果和余数组成的元组。示例如下:

>>> divmod(10f2)

(5, 0)

如果参数a和b都是整数，相当于(部/b或者a%b),计算的结果会是整数，示例如下：

>>> divmod(10f3)

(3, 1)

如果参数有浮点数，那么等价于(mathfk)(K(a/b),a%b),示例如下：

>>> divmod(10.4^4)

(2eOr 2.4000000000000004)

5b max()函数

max()函数有两种格式：

max(iterablef \*[7 key, default])

max(argl, arg2z \*args[f key])

这个函数的功能是返回多个参数中的最大值，或者传入可迭代对象元素的最大值。在 max()函数中，可以传入命名参数k©y，其为一个函数，用来指定取最大值的方法？ default参数 用来指定最大值不存在时返回的默认值。

如果传入的参数是数值类型，那么，参数至少要传入两个参数，返回的是较大值。示例如下： >>> max(10,3/20,34,12)

34

如果传入的参数是可迭代类型，那么可迭代类型不能为空。示例如下：

>>> max([])

Traceback (most recent call last):

File "<stdin>", line 1r in <module>

ValueError: max() arg is an empty sequence

如果传入的参数只有一个，那么参数必须是可迭代类型，返回的结果是可迭代对象的最大 元素，示例如下：

»> max( [10,3,20,34,12])

34

如果传入的参数是多个可迭代类型，那么对于数值类型的元素，取大者；字符型元素，取 字母表排序靠后者，示例如下:

>>> max([1f 2 z 3], [ 3 f 2 r1])

»> max ( [1 a \ lb \ T c ? ] z [1 a \ 1 c \* z MT ]) [faS fcJ z M1 ]

如果比较的两个对象不是同一种类型，那么，原本是不能比较获取最大值的，但是，如果 此时使用可选参数key,则可以比较获取最大值了，示例如下：

»> max (1,2, 1 3 \* )

Traceback (most recent call last): File "<stdin>", line 1, in <module>

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

>>> max(1z 2, '31zkey = int)

f3f

当函数传入的一个可迭代对象为空时9则必须传入参数default,用来指定最大值不存在时, 函数返回的默认值。示例如下：

>>> max([])

Traceback (most recent call last): File °<stdin>n, line 1, in <module>

ValueError: max() arg is an empty sequence

>>> max([]zdefault = 10)

10

& min()函数

min()函数有两种格式：

min(iterablef \* [, key, default])

min (argl f arg2, \*sirgs [} key])

这个函数的功能是返回多个参数中的最小值，或者传入可迭代对象元素的最小值。由于 min函数和max函数的用法基本一致，这里就不过多进行介绍了。

*7.* sum(iterable[? start])

该函数用于求和。该参数有两个要求：

1. 接收对象是可迭代类型
2. 可迭代对象所有元素类型是数值类型。

示例如下:

»> sum ( [1,2,3,4m )

21

在sum()函数中，还有一个可选参数start,表示求和之前的初始化值，如果传入的参数是 一个空的可迭代类型，那么函数会返回初始值。示例如下：

»> sum ( [1,2Z 3Z 4] z 10)

20

>>> sum([],10)

10

BJ2.3襲靈瓣應帼襲囈灘

下面是和类型转换相关的函数，具体如下：

1D ord(c)

ord()函数用于返回Unicode字符对应的整数数值，参数c是一个Unicode字符。示例如下:

»> ord (1 a !)

97

2a chr(i)

该函数的功能和ord正好相反,用于返回Unicode字符。示例如下:

»> chr ( 97)

\* a ?

需要注意的是，传入的参数值范围必须在0〜1 114 111 (十六进制为OxlOFFFF)之间， 否则将报ValueError错误。示例如下:

>>> chr (-2)

Traceback (most recent call last):

File "<stdin>"r line 1, in <module>

ValueError: chr() arg not in range(0x110000)

»> chr(1114112)

Traceback (most recent call last):

File °<stdin>n f line 1r in <module>

ValueError: chr() arg not in range(0x110000)

3b bool(M)

该函数用于返回一个Tme或者False的布尔值，参数是可选的，如果参数省略，那么返回 的值*为*Falseo示例如下:

>>> bool ()

False

>>> bool(20)

True

这里需要给大家再次申明，在Python中，除下列对象外，其余的值都为True。

-[]o

-()o

-{}。

* 0o
* None。
* 0.0。

•空字符串。

示例如下:

»> bool (口)

False

»> bool (())

False

»> bool ({ })

False

>» bool (0)

False

»> bool (None)

False

>>> bool (0 » 0)

False

»> bool ( f \* )

False

4. slice 1W

slice有两种格式:

slice(stop)

slice(starts stop[r step])

slice函数实际上是一个切片类的构造函数，它返回的是一个切片对象。参数如rt、step、 stop分别表示的是开始、步长、停止的索引。参数加rt和拊p默认值为None。示例如下：

>>> a *= [1F 2f* 3 f 4 f 5 f 6] >>> c = slice (2 f 5) # c是一个切片对象$表示截取索引为2~5 (不包含)的元素 »> d = slice (2Z 5, 2) # d是一个切片对象，表示截取索引为2~5，步长为2的元素 »> a [b]

b是一个切片对象3表示截取索引为0~3

b = slice (3)

[lz 2, 3] >>> a[c]

[3, 4, 5] >>> a[d]

[3, 5]

& dir([object])

dir函数可以查看对象内的所有属性和方法。该函数对于Python初学者或者有经验的程序 员都是非常有用的。参数object是可选的，属于对象类型。

di『函数的参数是可选的，如果我们不传入参数，那么该函数会返回当前作用域的变量、 方法和定义的类型列表,示例如下:

»> dir ()

[' builtins

doc

loader

name

package

spec ?]

如果参数对象是一个模块，那么返回的是模块的属性、方法列表，示例如下:

>>> import math >>> dir(math)



loader

name

package

spec \* s acos ? r 1acosh'z ? as in v f ? as inh 1p 1atan1f 1atan2 5 z 'atanh s 7? ceil5 f !copysign1f 'cos 1 ? cosh , 7 !degrees f r \* e 5 z 1 erf ,J erfc5 f ? exp J

expml\*f s fabs1, 1 factorial5, 1 floor1f 1fmods, 1frexp', 1fsum', \* gamma1, gcd ' , 1hypot1, !inf J'isclose ' , \* isfinite', \* isinf 5, 1isnan5z 1Idexp1z

1gamma'7 1 log ? f 'logl0 ! z ?loglp1r slog21f ? modf1f ,nan!f v pi ? f ? pow 5

radians 5, ,sin 17 5 sinh'f 1sqrt5 r 'tan', 1tanh\* f !trunc']

如果参数对象是一个类，返回的是类及其子类的属性、方法列表，示例如下：

>>> class Test:

.。。 number *= 10*

.。. word = f name 1

»> test = Test ()

>>> dir(test)

[? class

delattr

diet

dir ' r \* doc

eq \* J format \* f ' ge *，？* 1 getattribute ' ? gt \* ' hash \* init J r ? le , , ' It \* ? module J r ! ne \* , ' new ? r

reduce \* 1 reduce\_ex \* , repr v ' setattr \* ' sizeof

str \*r ' subclasshook f r ' weakref \*, \* number!f 1 word \*]

如果参数对象定义了 \_dir\_()方法，那么返回的是\_血\_()方法的结果。示例如下言

>>> class Demo:

•.. def dir (self):

return ' hello dir !

>>> demo = Demo()

>>> dir(demo) [f 、 fdJ , !1S ?1\ \*o\ rr?]

关于类对象还有属性的介绍，我们会在面向对象章节详细讲解。这里大家只需要知道 函数的参数可以传入类，返回的是类及其子类的属性方法列表即可。

8J2.4癖刿鶴傕帼美圈讖 ’

下面是和序列相关的函数，具体如下：

1. all(iterable)

用于判断可迭代对象中的元素是否都返回True,相当于下列代码：

def all(iterable):

for element in iterable:

if not element:

return False

return True

从上述代码可以看出，疝函数接收一个可迭代器对象为参数。当参数为空或者不为可迭代 器对象时，程序会报错。示例如下：

»> all (10) :

Traceback (most recent call last):

File n<stdin>n line 17 in <module>

TypeError: 1int! object is not iterable

如果可迭代对象中每个元素均为True时，返回True,否则返回Falseo示例如下：

»> all( [1,2,3])

True

如果可迭代对象为空(元素个数为0),返回True,示例如下:

»> all([])

True

2a any(iterable)

如果有一个元素是True,结果就是True,如果序列元素为空，那么返回false。相当于下 列代码：

def any(iterable):

for element in iterable:

if element:

return True

return False

该函数接收的参数必须是可迭代类型，如果不符合条件，那么程序会报错，示例如下：

»> any () #参数为空,程序报错

Traceback (most recent call last):

File °<stdin>", line 1, in <module>

TypeError: any() takes exactly one argument (0 given)

»> any(20) #参数不是可迭代类型，程序报错

Traceback (most recent call last):

File °<stdin>?,, line 1, in <module>

TypeError: \*int \* object is not iterable

■如果传入的参数是可迭代类型，并且其中一个元素的值为Tnw,那么any()函数的返回值 就是True,示例如下：

»> any ([0,1])

True

如果元素的值都为False,或者没有元素，那么any()函数的返回值是False,示例如下：

»> any ( [0, False])

False

»> any (口)

False

8J2.5鬟。对録。屬膛帼美遴灘 -

下面是和类、对象、属性相关的函数，具体如下：

牝 hasattr(object5 name) 、 「 「

函数功能用来检测对象obj颈中是否含有名为nam©的属性，如果有则返回True,如果没 有返回Falseo示例如下:

>>> class Person:

... def init (selfz name):

... self.name = name

>>> p = Person(\* Jack1)

>>> hasattr(pA , name 1)

True

»> hasattr (pz ' age 1)

False

实际上§函数调用的getattr(object9name)函数，通过是否抛出AttributeErro『异常来判断是 否含有属性。

1. getattr(object? nameL default])

该函数用于从对象object中获取名称为name的属性,其作用等效于object.nameo示例如下:

»> class Person:

... def init (selff name):

".. self.name = name

»> p = Person ( ' Jack 1)

»> getattr (pz \* name \*)

\* Jack \*

>>> p。name

1 Jack ?

在getattr函数中，参数default可选，如果object中含有name属性，则返回name属性的值, 如果没有name属性,则返回default值。示例如下:

>>> class Person:

... def init (self7 name):

self . name = name

>>> p = Person(\* Jack1)

»> getattr *(pf 1* name ,)

Mack\*

>>> getattr(p^ 'age ? f18)

18

需要注意的是，在调用getattr 如果没有iw眼属性，也没有传入default,那么程序会报错。

具体如下：

>>> getattr (pf \* age \* ) # age不存在,而且没有设置default?会报错

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: \* Person \* obj ect has no attribute ,age ,

1. setattr(object? name, value)

该函数和getattr函数是对应的,不同的是,setattr函数用于设置对象的属性值,将object 对象的name属性设置为value值，而getatt『用于获取对象属性值。示例如下：

>>> class Person:

."。 def init (self,name):

…。 self. name = name

>>> p = Person(f Jack')

>>> p.name

'Jack \*

>» setattr (p^ v name 5 f 1 Rose ?)

>>> p.name

'Rose 5

这里需要说明的是，使用setattr object对象的属性设置值时，如果属性已经存在，那么

会对这个属性值进行更新，如果属性不存在，那么对象会新建一个属性，并对其进行赋值。

1. delattr(object? name)

该函数用于删除object对象中的name属性。示例如下:

>>> class Person:

..o def init (self7 name):

•… self o name = name

>>> p = Person(\* Jack5)

>>> p.name

Mack 5

>>> delattr (p, ' name 1) #删除p对象name属性后,再次访问会提示 name属性不存在

>>> p.name

Traceback (most recent call last): File "<stdin>° 7 line 1r in <module>

AttributeError: !Person 1 object has no attribute 1 name!

5O isinstance(object? classinfo)

该函数用于判断对象是否是类型对象的实例，object参数表示需要检查的对象，classinfo 参数表示类型对象。示例如下：

>>> class Person:

def init (self z name):

。… self . name = name

>>> class Student:

def init (self name):

*。…* self . name = name

>>> person= person(\* Jack \*)

>>> student = Student(\* Rose v )

>>> isinstance(person^ Person)

True

>>> isinstance(student^ Person)

False

如果object参数传入的是类型对象，则始终返回False。示例如下：

>>> isinstance(bool,int)

False

»> isinstance (strz tuple) False

如果参数classinfo是一个元组类型对象，object是任何类型的实例，都会返回true,否则 返回Falseo示例如下:

>>> isinstance(person,(Person,Student))

True

如果参数classinfo不是一个类型对象或者由多个类型对象组成的元组，则会引发TypeError 异常。示例如下：

»> isinstance (person^ [Person^ Student])

Traceback (most recent call last):

File n<stdin>"r line 1f in <module>

TypeError: isinstance() arg 2 must be a type or tuple of types

上述讲解的内容都是与类对象和属性相关的函数。由于类对象和属性的知识会在面向对象 章节进行专门介绍，大家不妨学完面向对象的知识后再回过头来深刻理解这些函数的用法。这 里我们只需要明确Python针对类对象和属性提供了丰富的函数即可。

8J2.6疆电豌行帼集圈鐡 . ' \* \* 「

下面是和编译执行相关的函数，具体如下：

10 repr(object) . • 、.

该函数的功能与紀()类似，不同的是，函数啪()用于将对象转为可以阅读的字符串格式， 而repr用于将对象转为供解释器读取的形式。示例如下：

>>> words = ,hello itheima1

>>> str(words)

? hello itheima ?

>>> repr(words)

° v hello itheima \* °

对于一般的类型，当对应类型的对象调用冀函数时，返回的是其所属的类型和被定义的 模块，以及内存地址组成的字符串。示例如下：

>>> class Person:

•.。 def \_init\_(self, name):

self . name = name

>>> person = Person(?Jack 1)

>>> repr(person)

5 < main .Person obj ect at 0x1019e6cl8> v

如果要改变类型的Rpr函数显示信息，需要在类型中定义\_mpr\_()函数进行控制。示例 如下：

>>> class Person:

•.. def init (self, name):

•。. self.name = name

o \* . def repr (self):

.o. return (1 name 为:\* + self.name)

»> person = Person ( \* Jack ,)

>>> repr(person)

1. name 为:Jack
2. compile(source? filename, mode, flags=0, dontJnherit=False, optimize=-1) 该函数用于将source编译为代码或者AST对象，代码对象能够通过exec()或者eval()来 执行，其参数介绍如下：

* source:字符串或者AST对象，即需要动态执行的代码段。

。filename:代码文件名称。如果不是从文件读取代码，则传递一些可辨认的值，当传入了 source参数9 filename参数传入空字符串即可。

* mode：指定编译代码的种类,可以指定为“exec” “eval” “single”。当source中包含 流程语句时,model应该指定为“exec”，当source中只包含一个简单的求值表达式， model应该指定为“eval” ,当source中包含了交互式命令语句,model应该指定为“single”。

°可选参数flags和dont inherit是用来控制编译源码时的标志,可以查看PEP236文档来 了解这些参数，以及相关编译的说明。

下面看一些例子。

(1 )流程语句使用exec,示例如下:

>>> words = \* for i in range(5) :print(i) 5

>>> result = compile(words, 1? z 1 exec')

>>> exec (result)

0

1

2

3

4

(2)编译为表达式，表达式使用eval,示例如下：

>>> words = \*6\*6'

»> result = compile (str, 1 ' , 1 eval5 )

»> eval (result)

36

(3)交互式语句使用single,示例如下：

>>> wonds = , input (°请输入您的名字:")\*

>>> result = compile(words f '1r 1 single ?)

>>> exec(result) 请输入您的名字：Jack

'Jack

3a eval(expression? globals=Nones locals=None)

该函数用来动态执行一个表达式的字符串，或者compile函数编译出来的代码对象，函数 的参数介绍如下：

• expression：代表的是表达式字符串或者编译岀来代码对象的名称。

。globals：全局命名空间，指定代码执行时可以使用的全局变量以及收集代码执行后的全 局变量。

。locals：局部作用域命名空间，用来指定代码执行时可以使用的局部变量以及收集代码 执行后的局部变量。

接下来看一个例子，具体如下：

| 1 | x = | 100 |
| --- | --- | --- |
| 2 | def | func (): |
| 3 |  | y = 200 |
| 4 |  | a = eval(\* *x* + y\*) |
| 5 |  | print(s a:七 a) |
| 6 |  | b = eval(1x + y\*f |
| 7 |  | print(fb: ' b) |
| 8 |  | c = eval(\*x + y\*f |
| 9 |  | print(\* c: \* c) |
| 10 | d = | eval(!print(x, y) \*) |
| 11 | print(!d: \* d) | |
| 12 | func() |  |

{ ‘X、 1, 'y' : 2})

{成e. 1, *十:*2), {?yf: 3, \*z?: 4})

输出结果如下:

a: 300

b: 3

c: 4

100 200 d: None

上述代码的具体介绍如下：

第4行代码:evalQ函数省略了 globals和locals参数，因此,x和y的取值都是eval()函 数被调用作用域内的变量值，也就是x的值是100, y的值是200。最终，a=x+y=300。

第6彳亍代码,eval()函数提供了 globals参数,省略了 locals参数,因此locals会取globals 参数的值，即 x=l,y = 2, b = x + y = 3。

第8行代码，©V初1()函数的globals参数和locals都被提供了，那么eval()函数会先从全部 作用域globals中找到变量x,从局部作用域locals中找到变量y,即x = 1, y = 3, c = x + y = 4。

第10行代码，因为pE<)函数不是一个计算表达式，没有计算结果，因此返回值为None。

4L ®x@c(object[? globalsL locals]])

exec()函数和eval()函数类似,也是执行动态语句，只不过eval()函数只用于执行表达式 求值身而宓硕)函数主要用于执行语句块，因为exec()中的可选参数globals和locals的功能， 与eval()类似，这里就不再做介绍了。

Python还有一些其他的函数，这里我们选择其中一部分函数进行介绍，具体如下？

1 globalsO

返回当前作用域内全局变量的字典。示例如下：

>>> globals()

( ' name ' : ' main ' 7 ' loader ?:

<class s\_frozen\_importlib. Builtinlmporter ® >f ' doc ?: Nonez

f spec \* : Nonez f builtins 5 : <module \*builtins 1 (built-in) >z„ 5 package ?: None}

>>> a = 10

>>> globals () # 多了一个变量 a

! name 5 : J main \* 1 loader ?:

<class f\_frozen\_importlib. Builtinlmporter ? f ' doc \* : None7

5 a ?: 10 f 5 builtins \*: <module \*builtins1 (built-in)

! package \* : None^ ' spec ' : None}

2" localsO

其功能与globalsO函数类似，locals()函数用于返回当前作用域内的局部变量和值组成的字 典。示例如下；

>>> locals()

{ ' spec ' : Nonef \* loader ':

<class \*\_frozen\_importlib » Builtinlmporter1>z ' doc \*: None,

早 builtins J: <module \*builtins v (built-in)>f \* name 1: \* main ? package ! : None}

»> a = 10

>>> locals () # 多了一个变量 a

{ 1 spec 1 : Nonef ? loader ,:

<class rozen\_importlib o Builtinlmporter1>f 5 doc Nonez ' builtins 1 : <module fbuiltins ?' (built-in)>f

\* name v ' main \* J package \* : Nonez s a \* : 10 }

需要注意的是，locals函数返回的字典是不允许修改的。因为locals函数实际上是对局部 变量和值组成字典的复制，所以我们对它进行修改，修改的只是复制的内容，而实际的变量 并没有影响。

1. ascii(object)

返回一个字符串包含一个可打印的对象表示，如果是非ASCII字符，就会输岀\x、\u或\U 等字符。等效于Python2中的repr函数。

»> ascii (12)

r 12 v

»> ascii (97)

? 97 °

»> ascii (，中国。)

° ' Wu4e2d\\u56fd, °

4。caHable(object)

该函数用于检查对象是否可以被调用，这里指的可调用，指的是对象是否可以使用圆括号 来调用。如果可以被调用，那么返回结果是True,否则返回Fa屁。

示例如下:

>>> def func ():

.。o return !itheima'

»> f = func ()

>>> callable(func)

True

>>> callable(f)

False

在Python中，类对象都是可以被调用的对象，类的实例对象是否可以被调用，取决于类 是否定义了 —call\_()方法。示例如下:

>>> class A:

。… pass

>>> class B:

…° def call\_(self):

…. pass

»> a = A()

»> b = B ()

>>> callable(A)

True

>>> callable(a)

False

>>> callable(B)

True

>>> callable(b)

True