第 3 章　Python 函数式编程


λ 演算与函数式编程

20 世纪 60 年代的软件危机爆发后不久，人们便开始从不同的角度寻找出路：一部分人以改革的思想开辟了结构化程序设计路径，开辟了软件工程领域；另一部分人则开始怀念数学高度的抽象性和严密的逻辑性，创立了一个“程序设计方法学”的研究方向，想从问题的需求出发一步一步地推导出程序来。他们认为这样得到的程序一定是严密的、不会有错误的程序。几十年来尽管人们付出了不懈的努力，在循环不变式等方面取得了一些成果，但一个完整的体系一直没有建立起来。

“有意栽花花不活，无心插柳柳成荫”。程序设计方法学研究领域的专家们想把数学逻辑应用到程序设计的追求却被一位研究人工智能的学者在 1958 年找到了突破口。这位学者就是麻省理工学院教授，后来被称为“人工智能之父”的约翰·麦卡锡(John McCarthy，1927—2011，见图 3.1)。他在研究符号计算时，从数学家阿隆佐·邱奇(Alonzo Church，1903—1995)的 λ 演算(lambda calculus)中得到启发，并基于 λ 演算开发出了人工智能领域的第一个程序设计语言——LISP(LISt Processing)语言(这也是第一个有别于命令式语言的声明式体系中的函数式程序设计语言)，并于 1960 年 4 月，以《递回函数的符号表达式以及由机器运算的方式(第一部)》为题，发表了相关论文。

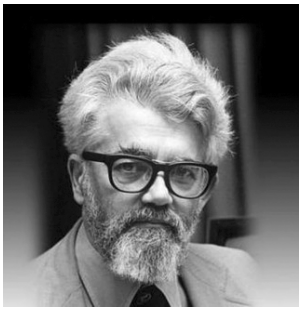
图 3.1　约翰·麦卡锡

然而，这并没有引起太多的关注。致使在邱奇-麦卡锡的函数式编程思想被“冷藏”了近 20 年，直到 2010 年，由于 JavaScript 引入了 λ 演算并发挥出其强大功能，才让人们认识到它的意义。于是，几乎所有的程序设计语言都开始从引入 lambda 表达式(也称 lambda 函数或匿名函数)入手，开始支持函数式编程。Python 也不例外，并且做得更好。

3.1　Python 函数式编程基础

早在 1938 年，邱奇就提出了在 λ 演算中的每个函数都必须有如下特征。

(1) 单一参数。

(2) 函数的参数是一个单一参数的函数。

(3) 函数的(返回)值是一个单一参数的函数。

但是，那是程序设计还没有出现之前提出的。近年来，随着函数式编程得到广泛青睐，函数式编程思想也开始针对命令式编程中存在的问题，不断进行完善，具体如下。

(1) 函数是一类对象，并称为一种对象类型。

(2) 只用“表达式”(expression)，不用“语句”(statement)。

(3) 没有副作用，使用纯函数，不修改状态，数据不变，不含任何赋名语句。

(4) 引用透明性：表达式的值不依赖可以改变值的全局变量；对同样的输入，总是返回

同样的结果。

(5) 惰性计算。惰性计算也称延迟求值(lazy evaluation,也称作 call-by-need),是指表达式不在它被绑定到变量时就立即求值,而是在该值被用到的时候才计算求值。

在前面介绍的内容中,Python 在这些方面的改进已经初见端倪,如将多数数据对象定义为不可变类型;将变量定义为赋名型,不再是原生型;赋名操作符(=)不再具有赋值功能等。这一节主要介绍 Python 支持函数式编程的其他一些基本机制。

3.1.1 函数作为"第一等对象"

"第一等对象"(first-class object)指的是函数与其他数据类型一样,处于平等地位,可以被变量引用,也可以作为参数,传入另一个函数,或者作为别的函数的返回值。这一特征 Python 是满足的。

1. Python 函数也是对象,有它自己的类型、身份码和值

Python 一切皆对象。函数也是一类数据对象,它们都满足 Python 对象的 3 个基本属性:具有身份码、类型和值。函数名就是指向函数对象的名字。

代码 3.1 获取函数的类型、身份码和值示例。

```
>>> def func():
...     print ('I am a function')
...     return (None)
...
>>> print (type(func))                    #输出函数的类型
    <class 'function'>
>>> print (id(func))                      #输出函数的身份码
    2182932023360
>>> print (func())                        #输出函数的值
    I am a function
    None
```

可见,Python 函数是一类 Python 数据类型,其类型名就是 function;而每个函数都是一个 function 类型的对象。None 是一个特殊值。

2. Python 函数是第一等对象

作为对象的一种,在 Python 中就可以像其他数据对象一样使用函数。具体地说,就是可以在下列 3 种情况下使用函数。这就为 Python 函数式编程提供了有力支持,为此也将 Python 函数称为第一等对象。

1) Python 函数可以作为元素添加到容器对象中

代码 3.2 函数作为元素存储在容器中示例。

```
>>> def disp1():
...     print("abcd")
...     return None
...
>>> def disp2():
...     print("efgh")
...     return None
```

```
...
>>> def disp3():
...     print("1357")
...     return None
...
>>> disps = [disp1,disp2,disp3]
>>> for d in disps:
...     d()
...
    abcd
    efgh
    1357
```

2）Python 函数可以作为参数传递给其他函数

如在代码 2.19 中，triArea()函数作为 print()的参数。

3）Python 函数可以被一个变量引用并作为函数返回的对象

如在代码 2.21 中，函数 type()成为函数 triArea()的返回值。

3.1.2 纯函数

函数式编程是抽象性很高的编程模式。理想的函数式编程基于数学中的函数映射来考虑问题求解，组织程序代码。这些数学层面上的函数的基本特征是计算的透明性(transparency)，即给定相同输入总能得到相同输出的函数，也就是把函数看作黑箱，其输出仅与输入参数有关。

现代多数程序设计环境还是基于命令式编程的，要在这种环境下实现数学层面上的函数难度不小，但是也并非是做不到的。实际上，只要对命令式环境中的函数进行约束，它们同样可以具有上述透明性。为了区别于一般的命令式函数，将这种具有透明性的函数称为纯函数(pure function)，纯函数应当具有如下两个特征。

(1) 函数的返回值只与输入参数相关，对于相同的输入参数一定会得到相同的返回值。

(2) 函数没有副作用。

这两个特征是通过相应的约束实现的。

1. 保证函数返回值只与函数参数相关的约束

为保证函数返回值只与函数参数相关，函数只能通过显式通道(参数)从环境中获取数据对象，而不能通过隐式通道从环境中获取数据对象。函数获取数据的隐式通道有如下 3 种。

(1) 全局变量。

(2) input 函数。

(3) 调用非纯函数。

代码 3.3 非纯函数示例之一。

```
>> x = 3
>>> def add1(y):
...     return x + y
...
>>> add1(5)
    8
```

在函数 add1()中，返回表达式由两个数据对象组成，即显式通道获得的数据对象 x 和隐式通道获得的数据对象 y。由于 y 的存在就无法保证该函数的返回值仅与输入的参数相关。在函数设计时，对于同样的输入参数 x，因无法保证函数一定会从环境中获得同样的 y，因而无法保证同样的输入参数一定会得到同样的返回值。所以这个函数不是一个纯函数。

2. 保证函数无副作用的约束——计算环境不变原则

在程序中，一个操作在进行本职工作的同时，得到了超出本职工作要求的结果，就称其产生了副作用(side effect)。前面介绍过分配语句和分配表达式就是容易形成副作用的操作。在一般情况下，它们的本职工作是计算一个表达式的值，并把这个值对象引用到一个变量。但是，对于一个可变对象来说，操作的同时会修改变量所引用的对象，从而修改了计算环境，这就是它们的副作用。为了最大限度地降低这类操作的副作用，Python 采取了两项措施：一是摒弃了多数程序设计语言中使用的原生型变量，改用引用型变量；二是将多种数据对象定义为不可变对象。

对函数来说，其本职工作是完成某些计算。在函数式程序设计中，函数被用来实现输入参数到函数返回之间的映射。这就是函数的本职工作。但是，如果一个函数在完成本职工作的同时，还改变了函数环境，也就有了副作用。为保证函数没有副作用，就需要关闭所有隐式输出通道，只能通过返回语句向调用者返回输入参数的映射。函数的隐式输出通道大概有如下 6 种。

(1) 修改全局变量。

(2) 修改参数值。

(3) 对打印或其他设备进行控制操作。

(4) 与文件、数据库、网络等的数据交换。

(5) 抛出异常。

(6) 调用非纯函数。

代码 3.4 非纯函数示例之二。

```
>>> x = 3
>>> y = 5
>>> def add2(a,b):
...     global x
...     x = a + b
...     return x
...
>>> add2(x,y)
    13
```

函数 add2()包含修改外部变量 x 的操作，所以它不是纯函数。

3. 纯函数的优越性

函数式编程之所以备受青睐，是因为纯函数可以带来如下好处。

(1) 便于直接测试。相同的参数可以产生相同的返回值，从而使纯函数可以方便地进行直接测试。

(2) 提高程序设计效率,并便于维护和重构。纯函数不存在与外界环境进行交互的隐形通道,而且它具有对外部环境没有副作用的透明性,使得一段代码可以在不改变整个程序运行结果的前提下用其等价的运行结果替代,这意味着可以进行与数学中的等式推导类似的推导。这种等式推导就可以实现人们梦寐以求的程序代码自动生成,还为理解代码带来极大便利,使得代码维护和重构更加容易。

(3) 支持并行处理。一般说来,在多线程环境下并行操作共享的存储数据可能会出现意外情况,而纯函数不依赖于环境状态的特点,使其根本不需要访问共享的内存。纯函数不需要访问数据,所以在并行环境下可以随意运行纯函数。

3.1.3 lambda 表达式

这里说的 lambda 表达式,也称 lambda 函数或匿名函数,是在 λ 演算模型基础上衍化出来的一种程序设计语言表达形式。它的基本作用是把一个函数定义写成一个表达式,而不是语句块。

1. 用 lambda 表达式表示单参数函数

1) lambda 表达式有参数,可以调用并传递参数

代码 3.5 一个单参数函数的 lambda 表达式。

```
>>> g = lambda x:x + 1
>>> g(1),g(2),g(3)
    (2, 3, 4)
```

显然,这个 lambda 表达式就是下面函数的语法形式。

```
>>> def g(x):
...     return x + 1
...
>>> g(1),g(2),g(3)
    (2, 3, 4)
```

所以,用 lambda 表达式定义一个函数时,它的基本语法由如下冒号(:)分隔的两部分组成:

```
lambda 参数 :表达式
```

前面的部分是参数说明,用关键字 lambda(λ 演算中的 λ)将后面的自变量绑定到后面的表达式中。

2) 参数可以有默认值。

代码 3.6 一个有默认值的单参数函数的 lambda 表达式。

```
>>> g = lambda x = 2:x + 1
>>> g()
    3
>>> g(6)
    7
```

显然,lambda 表达式简化了函数定义的书写形式,使代码更为简洁。

2. 多参数函数的 lambda 表达式

代码 3.7 一个计算三数之和，部分参数有默认值的 lambda 表达式。

```
>>> sum = lambda a, b = 3, c = 5: a + b + c     #定义一个 lambda 表达式 f,有 3 个参数
>>> sum(1)                                      #调用表达式 f,参数 a 为 1,b、c 为默认值 3、5
    9
>>> sum(3,5,7)                                  #调用表达式 f,参数 a、b、c 为 3、5、7
    15
```

3. 选择结构的 lambda 表达式

代码 3.8 求绝对值。

```
>>> abs = lambda x : x if x >= 0 else -x
>>> abs(3)
    3
>>> abs(-3)
    3
```

这里使用了一个三元运算符。

4. lambda 表达式作为其他函数的参数

代码 3.9 lambda 表达式作为 print()的参数，打印出列表 [0,1,2,3,4,5,6,7,8,9] 中能被 3 整除的数组成的列表。

```
>>> print ([x for x in [0,1,2,3,4,5,6,7,8,9] if x %3 == 0])
    [0, 3, 6, 9]
```

一个清晰的写法如下：

```
>>> foo = [0,1,2,3,4,5,6,7,8,9]
>>> print ([x for x in foo if x %3 == 0])
    [0, 3, 6, 9]
```

更简便的写法如下：

```
>>> print ([x for x in range(10) if x %3 == 0])
    [0, 3, 6, 9]
```

这里顺便介绍了含重复结构的 lambda 表达式的写法。

习题 3.1

一、选择题

1. 下列关于匿名函数的说法中，正确的是(　　)。

A. lambda 是一个表达式，不是语句

B.在 lambda 的格式中，lambda 参数 1，参数 2，…是由参数构成的表达式

C. lambda 可以用 def 定义一个命名函数替换

D.对于 mn＝(lambda x,y:x if x＜y else y)，mn(3,5)可以返回两个数字中的大者

2. 关于 Python 的 lambda 函数，以下选项中描述错误的是(　　)。

A. lambda 函数将函数名作为函数结果返回

B. f=lambda x,y:x+y 执行后，f 的类型为数字类型

C. lambda 用于定义简单的、能够在一行内表示的函数

D. 可以使用 lambda 函数定义列表的排序原则

二、判断题

1. 命令式编程以冯·诺依曼计算机为环境，函数式编程以图灵计算机为环境。(　　)

2. 第一等对象就是与其他对象具有相同的作用的对象。(　　)

3. 计算透明性要求函数中不使用全局变量。(　　)

4. Python 函数都是第一等对象。(　　)

三、代码分析题

阅读下面的代码，指出程序运行结果并说明原因。也可以先在计算机上执行，得到结果，再分析得到这种结果的理由。

1.

```
d = lambda p: p; t = lambda p: p * 3
x = 2; x = d(x); x = t(x); x = d(x); print(x)
```

2.

```
def is_not_empty(s):
    return s and len(s.strip()) > 0

print (filter(lambda s:s and len(s.strip())>0, ['test', None, '', 'str', '','END']))
```

四、实践题

1. 使用 lambda 匿名函数完成以下操作：

```
def add(x,y):
    return x+y
```

2. 台阶问题。一只青蛙一次可以跳 1 级台阶，也可以跳 2 级台阶。求该青蛙跳一个 n 级的台阶总共有多少种跳法。用函数和 lambda 表达式分别求解。

3. 变态台阶问题。一只青蛙一次可以跳 1 级台阶，也可以跳 2 级台阶……也可以跳 n 级。求该青蛙跳一个 n 级的台阶总共有多少种跳法。用函数和 lambda 表达式分别求解。

4. 矩形覆盖。可以用 2×1 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 2×1的小矩形无重叠地覆盖一个 2×n 的大矩形，总共有多少种方法？用函数和 lambda 表达式分别求解。

五、简答题

1. 命令式编程有哪些不足之处？

2. 简述函数式编程的核心特征。

3. 计算的透明性包括哪些内容?

4. 查询资料,列出所有你知道的支持函数式编程的语言。

3.2 Python 函数式编程模式

3.2.1 高阶函数

高阶函数(higher-order function)是函数式编程的基本机制,是至少满足下列一个条件的函数:

- 接受一个或多个函数作为参数输入。
- 输出一个函数(返回值中包含函数名)。

高阶函数的意义在于它能使功能相关的函数,像一串代数式一样,形成一个无副作用的函数调用链条或函数处理流水线。

Python 将函数作为第一类对象,就提供了对高阶函数的支持。此外它还内置了一些高阶函数,让程序员可以轻松地拿来构建应用函数链。

1. Python 内置的高阶函数

1) filter()

filter()用于过滤掉可迭代对象中不符合条件的元素,返回符合条件的对象列表,语法如下:

```
reduce(function,iterable[,initializer])
```

通常它必须含如下两个参数:

- function:有两个参数的函数。
- iterable:可迭代象。

initializer 是一个可选的初始参数。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。

代码 3.10 在一个 list 中,删掉偶数,只保留奇数。

```
>>> def is_odd(n):
...     return n %2 == 1
...
>>> list(filter(is_odd, [1,2,3,4,5,6,7,8,9]))
    [1, 3, 5, 7, 9]
```

也可以用 lambda 表达式作为参数,表示为:

```
>>> list1 = list(filter(lambda x : x %2 == 1,[1,2,3,4,5,6,7,8,9]))
>>> print(list1)
    [1, 3, 5, 7,9]
```

2) sorted()

sorted()可以对 list 进行排序,语法如下:

```
sorted(iterable,key=None,reverse=False)
```

参数说明：

- iterable：序列容器。
- key：只有一个参数的函数，指定比较对象，默认本身数字值。
- reverse：排序规则，True 为降序，False 为升序（默认）。

代码 3.11 用 sorted()函数进行简单序列元素排序示例。

```
>>> sorted([22,5, -111, 99, -33])                          #按本身值比较升序排序
    [-111,-33, 5, 22, 99]
>>> sorted([22, 5, -111, 99, -33], key = abs)              #按绝对值比较升序排序
    [5, 22, 99,-33,-111]
>>> sorted(['bcde','opq','asp','kmn'])                     #对字符串升序排序
    ['asp','bcde','kmn','opq']
>>> sorted(['bcde','opq','asp', 'kmn'],reverse=True)       #对字符串降序排序
    ['opq','kmn','bcde','asp']
>>> L=[('b',2),('a',1),('c',3),('d',4)]
>>> sorted(L, key=lambda x:x[1])                           #基于各元组第二个元素排序
    [('a', 1),('b', 2),('c', 3),('d', 4) ]
```

说明： 排序的基本操作是比较和移位。其中，数字或基于数字（字符串）的大小比较非常简单。但是，对于无法取得数字值的元素，就必须使用自定义 key()函数进行特别的比较。

2. 自定义高阶函数与尾递归

仿照上面的内置高阶函数并不难。此外，上面的例子都是参数中有函数的。返回值有函数的典型是尾递归函数。

尾递归函数是指递归调用是整个函数体中最后执行的语句，且它的返回值不属于表达式的一部分时。这个递归调用就是尾递归。这样的递归函数没有任何递延操作，进行尾递归调用（或任何尾调用）时，调用者的返回位置不需要保存在调用栈上；当递归调用返回时，它将直接在先前保存的返回位置上进行分支。因此，尾部递归既节省了空间又节省了时间。在代码 2.30 中的 fact()函数的 return 语句中的递归调用是作为表达式的一部分，所以它不是尾递归函数。

代码 3.12 用尾递归函数计算阶乘。

```
>>> def fact_rial(n,acc = 1):
...     if n < 0:
...       return '错误的参数'
...     elif n== 1 or n == 0:
...       return acc
...     else:
...       acc = n * acc
...       return fact_rial(n - 1,acc)
...
>>> fact_rial(1)
    1
```

```
>>> fact_rial(5)
    120
>>> fact_rial(-2)
    '错误的参数'
```

3.2.2 闭包

1. 闭包的概念

闭包(closure)是一种特殊的嵌套函数。通俗地说,如果在一个嵌套函数中内层函数使用了包围(外层)函数中定义的局部变量,并且包围函数的返回值是内函数的引用,就构成了一个闭包。闭包可以用来在一个函数与一组私有变量之间创建关联关系。在给定函数被多次调用的过程中,这些私有变量能够保持其持久性。

代码 3.13 闭包性质演示。

```
>>> def showName(name):
...     def inner(age):
...         print ('My name is:',name)
...         print ('My age is:',age)
...     return inner                    #函数作为返回值
...
>>> #执行下面的语句
>>> f1 = showName                       #函数被变量 f1 引用
>>> f2 = f1('Zhang')                    #用 f1 代表 showName,其返回值(即 inner)被 f2 引用
>>> f2(18)                              #用 f2 代替 inner
    My name is: Zhang
    My age is: 18
>>>
>>> F2(20)                              #不需要前面两行代码
    My name is: Zhang
    My age is: 20
```

显然,代码 3.13 中的嵌套函数是符合闭包的定义的。在那里作为内函数中第一个 print()函数的 name 是包围函数的临时变量,并且包围函数的返回值是内函数的引用(函数名)。但是,讨论这样一个结构有什么意义呢? 若重新执行一下这个闭包,就会发现其具有的特别之处。

这时就会惊奇地发现,第二段不需要前面两行代码,就得到了结果 My name is: Zhang 和 My age is: 20。

一般来说,一旦一个函数执行了返回语句,该函数内部的临时变量所占有的存储空间就会被释放掉,其值不会被保存。再次调用时,要为其重新分配内存。但是在 Python 中,函数对象有一个 __closure__属性。当内嵌函数引用了它的包围函数的临时变量后,这些被引用的自由变量会被保存在该包围函数的 __closure__属性中,成为包围函数本身的一部分;也就是说,这些自由变量的生命周期会和包围函数一样,并被称为闭包变量。这就是 Python 对函数式编程的重要支持机制之一。或者说不同于函数,闭包把函数和运行时的引用环境打包成一个新的整体,不同的引用环境和相同的函数组合可以产生不同的实例。如在代码 3.13 中,每次调用 ShowName()函数时都将返回一个新的闭包实例。这些实例之间是隔离的,分别包含调用时不同的引用环境现场。

2. 闭包的作用

闭包是函数式编程中的重要机制，其主要作用如下。

（1）通常内部函数运行结束后，其运行的状态（局部变量）是不能保存的，而闭包使函数的局部变量信息依然可以保存下来。对于希望函数的每次执行结果都是基于这个函数上次的运行结果的程序设计，该机制非常有用。

（2）闭包有效地减少了函数所需定义的参数数目，这对于并行运算来说有重要的意义。在并行运算的环境下，可以让每台计算机负责一个函数，然后将一台计算机的输出与另一台计算机的输入串联起来，形成流水线式的工作，即由串联的计算机集群一端输入数据，而从其另一端输出数据。这样的情境最适合只有一个参数输入的函数。

（3）避免了使用全局变量。全局变量是一个程序文件中副作用最大的变量，为该程序文件中的所有表达式所共有。一处引用，有可能影响到其他处。因此，函数式编程要求使用闭包将函数与其所操作的某些数据（环境）关联起来。这样不同的函数需要同一个对象时，就不需要使用全局变量了。

（4）可以根据闭包变量使内嵌函数展现出不同的功能。这有点类似配置功能，可以修改外部的变量，闭包根据这个变量展现出不同的功能。

3. 闭包示例

代码 3.14 在 50×50 的棋盘上，用闭包从方向（direction）和步长（step）两个参数的变化上描述棋子跳动过程的代码（去掉所有的提示符）。

```
>>> origin = [0, 0]                    #坐标系统原点
>>> legal_x = [0, 50]                  #x 轴方向的合法坐标
>>> legal_y = [0, 50]                  #y 轴方向的合法坐标
>>> def create(pos = origin):
...     def player(direction,step):
...         new_x = pos[0] + direction[0] * step
...         new_y = pos[1] + direction[1] * step
...         pos[0] = new_x
...         pos[1] = new_y
...         return pos
...     return player
...
>>> player = create()                  #创建棋子 player,起点为原点
>>> print (player([1,0],5))            #向 x 轴正方向移动 5 步
    [5, 0]
>>> print (player([0,1],10))           #向 y 轴正方向移动 10 步
    [5, 10]
>>> print (player([-1,0],3))           #向 x 轴负方向移动 3 步
    [2, 10]]
>>> print (player([0,1],3))            #向 y 轴正方向移动 3 步
    [2, 13]
```

说明：

（1）棋子移动是基于前一个位置的。由上述运行结果可以看出，闭包的记忆功能十分适合这种问题。

(2) 该程序代码仅用于说明闭包的作用，并非一个完整的棋子移动程序，还有许多功能需要补充，例如，每跳一步还需要判断是否出界等。

3.2.3 装饰器

1. 软件开发的开闭原则与 Python 装饰器

软件开发是一种高强度的脑力劳动，稍有不慎就会酿成大祸。为了尽量避免错误，在长期的开发实践和应对软件危机的过程中，人们总结出一些基本原则。

(1) 单一职责原则(single responsibility principle，SRP)。

(2) 里氏替换原则(Liskov substitution principle，LSP)。

(3) 依赖倒置原则(dependence inversion principle，DIP)。

(4) 接口隔离原则(interface segregation principle，ISP)。

(5) 迪米特法则(law of Demeter，LOD)。

(6) 开闭原则(open closed principle，OCP)。

其中，开闭原则是勃兰特·梅耶(Bertrand Meyer，1950—)在 1988 年提出的。其目的是给出一个软件在运行中随着需求改变应如何与时俱进地进行维护的原则：软件实体应当对扩展开放，对修改关闭。其核心思想是尽量对原来的软件进行功能扩张使其满足新的需求，而不是通过修改原来的软件使其满足新的需求，以保持和提高软件的适应性、灵活性、稳定性和延续性，避免由修改带来的可靠性、正确性等方面的错误，降低维护成本。

Python 装饰器(decorator)是一项基于开闭原则的技术，它可以在不侵入原有代码的前提下，从外部用一个 Python 函数或类对一个函数或类进行功能扩充，是代码复用的高级形式。这里，以对函数进行功能扩充的装饰器为例，介绍 Python 装饰器的基本原理。

2. Python 函数装饰器的实现

一个 Python 函数装饰器就是另一个函数，以函数作为参数并返回一个替换函数的可执行函数。或者说，装饰器就是一个返回函数的高阶函数。

代码 3.15 简单的装饰器示例函数 add()只有两个数相加的计算功能，用装饰器来补充一个打印功能。

```
>>> def add(x,y):                    #功能函数,只进行计算
...     return x + y
...
>>> def logger(func):                #参数剥离
...     def wrapper(a,b):             #增加一阶
...         print(f'{a} + {b} = ',end = '')
...         return func(a,b)
...     return wrapper
...
>>> add = logger(add)
>>> add(3,5)
    3 + 5 = 8
```

说明：

(1) 根据前面介绍的关于 Python 函数第一等对象的特征，从语法的角度不难理解上述代码。在这段代码中，add()是一个原来设计好的功能函数，用于实现两个数的相加。

(2) logger 是一个装饰器，就是一个函数，它的参数 func 用来接收要包装的函数名，其内部定义 wrapper()函数来接收 func 的参数进行处理，并用 return wrapper 返回结果。这样，执行 add=logger(add)，就是让 add 指向装饰器返回的 wrapper。即调用 add，实际上是调用了 wrapper。

3. Python 装饰符

在代码 3.15 中，使用了语句 add=logger(add)来说明函数 logger()是功能函数 add()的装饰器。不过，Python 中装饰器语法并不需要每次都用引用语句来说明装饰关系，在功能函数定义时只要在其前面加上“@+装饰器名字”就可以了。@称为装饰符。

代码 3.16 代码 3.15 改用装饰符的情形。

```
>>> def logger(fun):                          #参数剥离
...     def wrapper(a,b):                     #增加一阶
...         print(f'{a} + {b} = ',end = '')
...         return fun(a,b)
...     return wrapper
...
>>> @logger
...  def add(x,y):
...     return x + y
...
>>> print(add(3,5))                           #第二次输入参数
    3 + 5 = 8
```

说明：在功能函数前添加@装饰器标注，就将装饰器绑定在了功能函数上。这时直接用功能函数名调用，也就添加了装饰器扩展功能。

3.2.4 函数柯里化

函数柯里化(Currying，以逻辑学家哈斯凯尔·加里(Hsakell Curry)命名)是单一职责原则在处理多参数函数时的应用，其基本思路是把多参数函数转化成每次只传递处理一部分参数(往往是一个参数)的函数链，即每个函数都接收一个(或一部分)参数并让它返回一个函数去处理剩下的参数。函数柯里化可以用高阶函数实现。

代码 3.17 一个两数相加的函数。

```
>>> def add(x,y):
...     return (x + y)
...
>>> add(3,5)
    8
```

用高阶函数进行柯里化的形式如下：

```
>>> def add(x):
...     def _add(y):
...         return x+y
...     return _add
...
>>> add(3)(5)
    8
```

3.2.5 偏函数

偏函数(partial function)机制是函数式编程中的一个重要机制，它的基本思想是将几个“残缺不全”的函数重新封装，形成一个完全的函数。其目的在于实现代码复用。为实现这一目的，需要借助于 Python 内置的 functools 模块。functools 模块提供了很多有用的概念，其中的 partial 用于实现偏函数。

下面通过一个例子来说明偏函数的机制。

例 3.1 内置的 int()函数称为 int 类的构造函数，它可以将一个任何进制的数字字符串转换为十进制整数。为此它需要两个参数：被转换的数字字符串和给定进制的 base 参数(默认为 10)。

1. 用不同的函数实现不同数制的转换

代码 3.18 内置函数 int()的应用示例。

```
>>> int('11100010110011',base = 2)
    14515
>>> int('111000101100111',2)
    29031
>>> int('12001222',base = 3)
    3698
>>> int('12312300123',4)
    1797147
>>> int('12341234001234',5)
    1897562694
>>> int('123450012345',6)
    522082505
>>> int('12345600123456',7)
    131869845750
>>> int('1234567001234567', base = 8)
    45954942450039
>>> int('123456780012345678',9)
    21107054117580488
>>> int('123456789abcd00123456789def', base = 16)
    230768749268213885728077953510 23
```

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。于是可以定义一个 int2()函数，默认把 base=2 传进去。

2. 用默认参数函数实现不同数制的转换

代码 3.19 内置定义有默认值参数的 int2()函数示例。

```
>>> def int2(x,base = 2):
...     return int(x,base = 2)
...
>>> int2('11100010110011')
    14515
>>> int2('111000101100111')
    29031
>>> int2('123456789abcd00123456789def', base = 16)
    Traceback (most recent call last):
      File "<pyshell#2>", line 1, in <module>
        int2('123456789abcd00123456789def', base = 16)
      File "<pyshell#0>", line 2, in int2
        return int(x,base = 2)
    ValueError: invalid literal for int() with base 2: '123456789abcd00123456789def'
```

这时,不可以再对其他进制字符串进行转换。

3. 用偏函数实现不同数制的转换

代码 3.20 由 functools.partial 创建一个偏函数 int2()示例。

```
>>> import functools
>>> int2 = functools.partial(int,base=2)      #functools.partial 创建偏函数 int2()
>>> int2('11100010110011')
    14515
>>> int2('111000101100111')
    29031
>>> int2('123456789abcd00123456789def', base = 16)
    2307687492682138857280779535 1023
```

显然,这时还可以再对其他进制字符串进行转换。

3.2.6 生成器

1. 生成器的概念及其特点

生成器(generator)是 Python 函数式编程中的一个典型应用。它也是函数,但是是一种惰性求值(lazy evaluation)函数,即它不是一次调用就进行完所有计算,而是一次调用只进行一个值的计算,并且下一次调用会在前一次调用计算的基础上进行。

为什么会这样呢? 原因就在于它所使用的返回语句不是 return,而是 yield。yield 与 return 的区别在于,return 返回后,函数状态终止;而 yield 返回后仍会保存当前函数的执行状态,再次调用时会在之前保存的状态上继续执行。这种不断在前一个状态的基础上进行计算的过程称为迭代。这种迭代过程将在迭代不可再进行时结束。

具体地说,生成器具有如下特点。

(1) 生成器函数中会包含一个或多个 yield 语句。

(2) 生成器被调用时,会返回一个迭代器对象,但不会立即开始执行。

(3) 该函数一旦执行 yield,便会暂停,并将控制权转移给调用者。

(4) 调用者可以用 iter()和 next()这样的内置方法遍历迭代器对象,每触发一次,就会在生成器生成的可迭代对象中前进一步,给出所遍历到的一个元素对象。

(5) 每个生成器对象只能被迭代一次。

(6) 生成器可以直接应用于 for 循环。因为 for 循环接受一个迭代器,并使用 next()函数对其进行迭代。

(7) 当 StopIteration 被引发时,生成器自动结束。

图 3.2 斐波那契

例 3.2 斐波那契(Fibonacci,1175—1250,见图 3.2)是中世纪意大利数学家。他曾提出一个有趣的数学问题:有一对兔子,从出生后的第 3 个月起每个月都生一对兔子。小兔子长到第 3 个月又生一对兔子。如果生下的所有兔子都能成活,且所有的兔子都不会因年龄大而老死,问每个月的兔子总数为多少?这些数组成一个有趣的数列,人们将之称为 Fibonacci 数列。

代码 3.21 产生无穷 fibonacci 数列的生成器。

```
>>> def fib():
...     n,a,b = 0,0,1
...     while True:
...         yield b
...         a, b = b, a + b
...         n += 1
...
>>> f = fib()
>>> next(f)
    1
>>> next(f)
    1
>>> next(f)
    2
>>> next(f)
    3
>>> next(f)
    5
```

说明:

(1) 每用 next()向生成器请求一次数据,生成器将用下一个 yield 返回下一个数据。

(2) 这个生成器构造一个无穷 fibonacci 数列。但是,它不是一下子返回,而是随着用户的需要一个一个地陆续返回,充分显示出对内存的友好姿态。如果使用 return 把一整个序列返回,将会很快用尽内存。因此,利用生成器的惰性求值特点,可以在用多少生成多少的前提下构造一个无限的数据类型。

(3) 可以使用多个生成器对一系列操作进行流水线处理。

2. 生成器的其他应用方式

生成器运行后,除了可以 next()函数触发一步步地进行迭代,向生成器请求数据外,还有如下一些应用方法。

1) 用 for-in 向生成器请求数据

除了用 next(),还可以用 for。因为 for 中隐藏了一个 next()。它与直接用 next()请求

的不同之处在于能一次生成序列中的全部元素，除非用条件终止这个 for 结构。

代码 3.22 用 for-in 向生成器请求数据示例。

```
>>> def fib(n):
...     i,a,b =0,0,1
...     while i <= n - 1:
...       yield b
...       a,b = b,a + b
...       i += 1
...
>>> for i in fib(5):
...     print(i,end = ',')
...
    1,1,2,3,5,
```

2）以管道生成器的形式请求生成器中的数据

以管道生成器的形式请求生成器中的数据可以使用多个生成器对一系列操作进行流水线处理。

代码 3.23 将斐波那契数列生成器与一个平方数生成器连接成管道示例。

```
>>> def fib(n):
...     a,b = 0,1
...     for i in range(n):
...       a,b = b,a + b
...       yield b
...
>>> def square(n):
...     for i in n:
...       yield i ** 2
...
>>> print(sum(square(fib(5))))       #连接成管道
    103
```

3）以生成器表达式与列表解析式的形式向生成器请求数据

简化 for 和 if 语句，使用圆括号将之括起就形成一个生成器表达式，若使用方括号则形成一个列表解析式，以向生成器请求数据。

代码 3.24 以生成器表达式和列表解析式的形式向生成器 range()请求数据示例。

```
>>> #生成器表达式
>>> result1= (x for x in range(5))
>>> result1
    <generator object <genexpr> at 0x0000025659D836D0>
>>> type(result1)
    <class 'generator'>
>>> next(result1)
    0
>>> next(result1)
    1
>>> next(result1)
    2
>>> next(result1)
    3
>>> next(result1)
```

```
    4
>>> next(result1)
    Traceback (most recent call last):
      File "<pyshell#18>", line 1, in <module>
        next(result)
    StopIteration
>>> #列表解析表达式
>>> result2 = [ x for x in range(5)]
>>> type(result2)
    <class 'list'>
>>> result2
    [0, 1, 2, 3, 4]
>>> result2
    [0, 1, 2, 3, 4]
```

说明：

(1) 生成器表达式只可向前迭代执行，不可逆执行；列表解析式则可以重复执行。

(2) 使用生成器表达式可以轻松地动态创建简单的生成器。它使得构建生成器变得容易。

3. Python 内置的生成器举例

为了方便用户，Python 自己定义了一些内置生成器，range()就是应用最多的一个生成器。前面已经介绍，这里不再赘述。但要注意的是，range()不可以用 next()函数迭代。

除此之外，还有一些，下面仅举两例。

1）zip ()函数

zip()是一个打包函数。它能将多个序列打包成一个元组列表。其打包过程如下：从每个序列里获取一项，把这些项打包成元组。如果有多个序列，以最短的序列为元组的个数。

执行 next()函数可以逐步观察到 zip()生成序列的过程。

代码 3.25 用 next()观察 zip()的工作过程。

```
>>> zipped = zip(a,b,c)
>>> next(zipped)
    (1, 'a', 5)
>>> next(zipped)
    (2, 'b', 6)
>>> next(zipped)
    (3, 'c', 7)
>>> next(zipped)
    Traceback (most recent call last):
      File "<pyshell#17>", line 1, in <module>
        next(zipped)
    StopIteration
```

next()称为迭代器。zip()称为可迭代对象(iterable)。

2）map ()函数

map()是一个生成器，它会根据提供的函数对指定的一个或多个可迭代对象进行映射。map()函数的语法如下：

```
map(function, iterable1 [,iterable 2, …])
```

map()的第一个参数 function 是一个一对一或多对一的函数，剩下的参数是一个或多个可迭代对象。Map()被调用后，将用可迭代对象中的每个元素调用函数，返回包含每次function()函数的返回值，最后形成一个可迭代对象。

注意：如果是多个序列，要求序列包含的元素个数相同。

代码 3.26 map()函数用法示例。

```
>>> tup1 = (1,3,5,7,9)
>>> tup2 = (10,8,6,4,2)
>>> tup3 = tuple(map(lambda x,y: x + y,tup1,tup2))
>>> print(tup3)
    (11,11, 11,11,11)
```

它的参数由一个函数和多个可迭代对象组成，所生成的数据序列中的每项，都是由函数参数依次对可迭代对象的各项进行计算的结果。

代码 3.27 map()以一个可迭代对象为参数示例。

```
>>> m = map(lambda x : x * 2, [1, 2, 3])
>>> next(m)
    2
>>> next(m)
    4
>>> next(m)
    6
>>> next(m)
    Traceback (most recent call last):
      File "<pyshell#27>", line 1, in <module>
        next(m)
    StopIteration
>>> m1 = map(lambda x, y : x * y, [1, 3, 5], [2, 4, 6])
>>> next(m1)
    2
>>> next(m1)
    12
>>> next(m1)
    30
>>> next(m1)
    Traceback (most recent call last):
      File "<pyshell#32>", line 1, in <module>
        next(m1)
    StopIteration
```

代码 3.28 map()以三个可迭代对象为参数示例。

```
>>> m2 = map(lambda x, y, z : str(x) + str(y) + str(z), ['a', 'b', 'c'], ['p', 'q','r'],['x', '
        y', 'z'])
>>> next(m2)
    'apx'
>>> next(m2)
    'bqy'
```

```
>>> next(m2)
    'crz'
>>> next(m2)
    Traceback (most recent call last):
      File "<pyshell#38>", line 1, in <module>
        next(m2)
    StopIteration
```

习题 3.2

一、判断题

1. 闭包是在其词法上下文中引用了自由变量的函数。 ( )

2. 包是由函数与其相关的引用环境组合而成的对象。 ( )

3. 包在运行时可以有多个实例，不同的引用环境和相同的环境组合可以产生不同的实例。 ( )

4. 闭包是延伸了作用域的函数，其中包含了函数体中引用而不是定义体中定义的非全局变量。 ( )

5. 对于生成器对象 x=(3 for i in range(5))，连续两次执行 list(x)的结果是一样的。 ( )

6. 包含 yield 语句的函数一般称为生成器函数，可以用来创建生成器对象。 ( )

7. 在函数中 yield 语句的作用和 return 完全一样。 ( )

8. 对于数字 n，如果表达式 0 not in [n%d for d in range(2,n)]的值为 True，则说明 n 是素数。 ( )

二、代码分析题

阅读下面的代码，指出程序运行结果并说明原因。也可以先在计算机上执行，得到结果，再分析得到这种结果的理由。

1.

```
def greeting_conf(prefix):
    def greeting(name):
       print (prefix, name)
    return greeting

mGreeting = greeting_conf("Good Morning")
mGreeting("Wilber")
mGreeting("Will")
aGreeting = greeting_conf("Good Afternoon")
aGreeting("Wilber")
aGreeting("Will")
```

2.

```
def count():
    fs = []
```