IMG_256 介绍

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

高阶函数

\*变量也可以指向函数，函数名其实就是指向函数的变量

如果把函数名指向其他对象，函数就无法调用了，例如abs

当然由于abs函数实际上是定义在import builtins模块中的，所以要让修改abs变量的指向在其它模块也生效，要用import builtins; builtins.abs = 10。

定义

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

**def add**(x, y, f):  
    **return** f(x) + f(y)

重要方法

1.map/reduce

Python内建了map()和reduce()函数。

1)我们先看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。

>>> **def f**(x):  
...     **return** x \* x  
...  
>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])  
>>> list(r)  
[1, 4, 9, 16, 25, 36, 49, 64, 81]

注

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。

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]))  
['1', '2', '3', '4', '5', '6', '7', '8', '9']

2)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])  
25

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])  
13579

IMG_257 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()，我们就可以写出把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'))  
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))

Filter()

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。

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

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。对比原始的list和经过key=abs处理过的list：

list = [36, 5, -12, 9, -21]

keys = [36, 5,  12, 9,  21]

然后sorted()函数按照keys进行排序，并按照对应关系返回list相应的元素：

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

IMG_258 返回函数

函数作为返回值

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。

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

当我们调用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  
<**function** lazy\_sum.<locals>.sum at 0x101c6ed90>

调用函数f时，才真正计算求和的结果：

>>> f()  
25

在这个例子中，我们在函数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。

 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。

IMG_259 返回函数

阅读: 276646

函数作为返回值

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。

我们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

当我们调用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  
<**function** lazy\_sum.<locals>.sum at 0x101c6ed90>

调用函数f时，才真正计算求和的结果：

>>> f()  
25

在这个例子中，我们在函数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，当我们调用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  
False

f1()和f2()的调用结果互不影响。

IMG_260 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。我们来看一个例子：

**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。

你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果是：

>>> f1()  
9  
>>> f2()  
9  
>>> f3()  
9

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

IMG_261 装饰器

在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）

IMG_262 偏函数

functools.partial就是帮助我们创建一个偏函数的，不需要我们自己定义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')  
64  
>>> int2('1010101')  
85

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。