高阶函数

2921次阅读

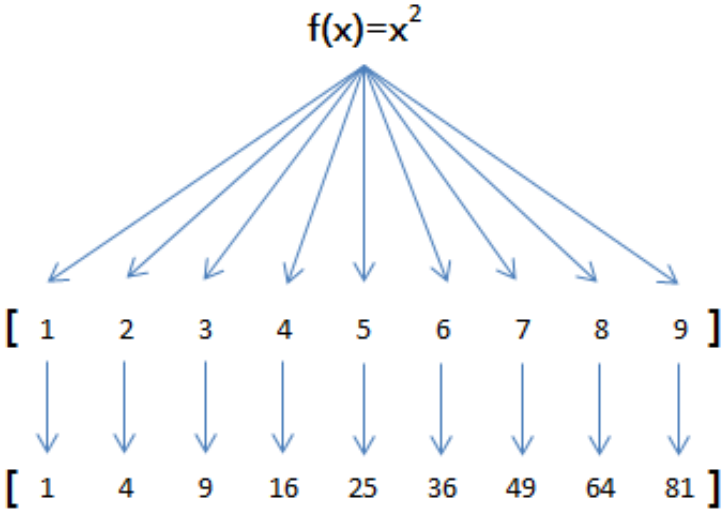
传入函数

要理解“函数本身也可以作为参数传入”，可以从Python内建的map/reduce函数入手。

如果你读过Google的那篇大名鼎鼎的论文“MapReduce: Simplified Data Processing on Large Clusters”，你就能大概明白map/reduce的概念。

我们先看map。map()函数接收两个参数，一个是函数，一个是序列，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list返回。

举例说明，比如我们有一个函数$f(x)=x^2$，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：



现在，我们用Python代码实现：

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

请注意我们定义的函数f。当我们写f时，指的是函数对象本身，当我们写f(1)时，指的是调用f函数，并传入参数1，期待返回结果1。

因此，map()传入的第一个参数是f，即函数对象本身。

像map()函数这种能够接收函数作为参数的函数，称之为高阶函数（Higher-order function）。

你可能会想，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，我们不但可以计算简单的$f(x)=x^2$，还可以计算任意复杂的函数。

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：

```
>>> 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：

```
>>> def fn(x, y):
...     return x * 10 + y
...
>>> reduce(fn, [1, 3, 5, 7, 9])
13579
```

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：

```
>>> def fn(x, y):
...     return x * 10 + y
...
>>> def char2num(s):
...     return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[s]
...
>>> reduce(fn, map(char2num, '13579'))
13579
```

整理成一个str2int的函数就是：

```
def str2int(s):
    def fn(x, y):
        return x * 10 + y
    def char2num(s):
        return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[s]
    return reduce(fn, map(char2num, s))
```

还可以用lambda函数进一步简化成：

```
def char2num(s):
    return {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}[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，你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！

lambda函数的用法在下一节介绍。

排序算法

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，我们可以直接比较，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。通常规定，对于两个元素x和y，如果认为x < y，则返回-1，如果认为x == y，则返回0，如果认为x > y，则返回1，这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序。

Python内置的sorted()函数就可以对list进行排序：

```
>>> sorted([36, 5, 12, 9, 21])
[5, 9, 12, 21, 36]
```

此外，sorted()函数也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。比如，如果要倒序排序，我们就可以自定义一个reversed_cmp函数：

```
def reversed_cmp(x, y):
    if x > y:
        return -1
    if x < y:
        return 1
    return 0
```

传入自定义的比较函数reversed_cmp，就可以实现倒序排序：

```
>>> sorted([36, 5, 12, 9, 21], reversed_cmp)
[36, 21, 12, 9, 5]
```

我们再看一个字符串排序的例子：

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

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，我们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，只要我们能定义出忽略大小写的比较算法就可以：

```
def cmp_ignore_case(s1, s2):
    u1 = s1.upper()
    u2 = s2.upper()
    if u1 < u2:
        return -1
    if u1 > u2:
        return 1
    return 0
```

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，我们给sorted传入上述比较函数，即可实现忽略大小写的排序：

```
>>> sorted(['about', 'bob', 'Zoo', 'Credit'], cmp_ignore_case)
['about', 'bob', 'Credit', 'Zoo']
```

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。

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 sum at 0x10452f668>
```

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。

小结

把函数作为参数传入，或者把函数作为返回值返回，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

假设Python没有提供map()函数，请自行编写一个my_map()函数实现与map()相同的功能。

Python提供的sum()函数可以接受一个list并求和，请编写一个prod()函数，可以接受一个list并利用reduce()求积。
