函数对象

1 python中高阶函数

函数名其实就是指向函数的变量

高阶函数：能接受函数做参数的函数。

例子 接受abs函数：

```
def add(x,y,f):
    return f(x)+f(y)
>>>add(-5,9,abs)
14
```

2 map()函数

map()是python内置的高阶函数，他接收一个函数f和一个list，并通过把函数f依次作用在list的每个元素上，得到一个新的list并返回。

例如，对于list [1, 2, 3, 4, 5, 6, 7, 8, 9] 如果希望把list的每个元素都作平方，就可以用map()函数：

我们只需要传入函数f(x)=x*x，就可以利用map()函数完成这个计算：

```
def f(x):
    return x*x
print map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
```

注意：map()函数不改变原有的 list，而是返回一个新的 list。

利用map()函数，可以把一个 list 转换为另一个 list，只需要传入转换函数。由于list包含的元素可以是任何类型，因此，map() 不仅仅可以处理只包含数值的 list，事实上它可以处理包含任意类型的 list，只要传入的函数f可以处理这种数据类型。

3 reduce()函数

reduce()函数也是Python内置的一个高阶函数。reduce()函数接收的参数和map()类似，一个函数 f，一个list，但行为和 map()不同，reduce()传入的函数 f 必须接收两个参数，reduce()对list的每个元素反复调用函数f，并返回最终结果值。

例如 **编**写一个f函数，接收x和y，返回x和y的和：

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

调用 reduce(f, [1, 3, 5, 7, 9])时，reduce函数将做如下计算：

先计算头两个元素：f(1, 3)，结果为4； 再把结果和第3个元素计算：f(4, 5)，结果为9； 再把结果和第4个元素计算：f(9, 7)，结果为16； 再把结果和第5个元素计算：f(16, 9)，结果为25； 由于没有更多的元素了，计算结束，返回结果25。

上述计算实际上是对 list 的所有元素求和。虽然Python内置了求和函数sum()，但是，利用reduce()求和也很简单。

reduce()还可以接收第3个可选参数，作为计算的初始值。如果把初始值设为100，计算：

```
reduce(f, [1, 3, 5, 7, 9], 100)
```

结果将变为125，因为第一轮计算是： 计算初始值和第一个元素：f(100, 1)，结果为101。

4 filter()函数

filter()函数是 Python 内置的另一个有用的高阶函数，filter()函数接收一个函数 f 和一个list，这个函数 f 的作用是对每个元素进行判断，返回 True或False，filter()根据判断结果自动过滤掉不符合条件的元素，返回由符合条件元素组成的新list。

例如 **要**从一个list [1, 4, 6, 7, 9, 12, 17]中删除偶数，保留奇数，首先，要编写一个判断奇数的函数：

```
def is_odd(x):
    return x % 2 == 1
```

然后，利用filter()过滤掉偶数：

```
filter(is_odd, [1, 4, 6, 7, 9, 12, 17])
```

结果：[1, 7, 9, 17]

利用filter()，可以完成很多有用的功能，例如，删除 None 或者空字符串：

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

结果：['test', 'str', 'END']

注意: s.strip(rm) 删除 s 字符串中开头、结尾处的 rm 序列的字符。

```
#当rm为空时，默认删除空白符（包括'\n', '\r', '\t', ' '），如下：
a = '     123'
a.strip()
```

结果： '123'

```
a='\t\t123\r\n'
a.strip()
```

结果：'123'

5 自定义排序函数

Python内置的 sorted()函数可对list进行排序：

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

但 sorted()也是一个高阶函数，它可以接收一个比较函数来实现自定义排序，比较函数的定义是，传入两个待比较的元素 x, y，如果 x 应该排在 y 的前面，返回 -1，如果 x 应该排在 y 的后面，返回 1。如果 x 和 y 相等，返回 0。

```
#因此，如果我们要实现倒序排序，只需要编写一个reversed_cmp函数：
def reversed_cmp(x, y):
    if x > y:
        return -1
    if x < y:
        return 1
    return 0


#这样，调用 sorted() 并传入 reversed_cmp 就可以实现倒序排序：
>>> sorted([36, 5, 12, 9, 21], reversed_cmp)
[36, 21, 12, 9, 5]
```

sorted()也可以对字符串进行排序，字符串默认按照ASCII大小来比较：

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

6 返回函数

Python的函数不但可以返回int、str、list、dict等数据类型，还可以返回函数！

例如 定义一个函数 f()，我们让它返回一个函数 g，可以这样写：

```
def f():
    print 'call f()...'
    # 定义函数g:
    def g():
        print 'call g()...'
    # 返回函数g:
    return g
```

仔细观察上面的函数定义，我们在函数 f 内部又定义了一个函数 g。由于函数 g 也是一个对象，函数名 g 就是指向函数 g 的变量，所以，最外层函数 f 可以返回变量 g，也就是函数 g 本身。

调用函数 f，我们会得到 f 返回的一个函数：

```
>>> x = f()    # 调用f()
call f()...
>>> x    # 变量x是f()返回的函数:
<function g at 0x1037bf320>
>>> x()    # x指向函数，因此可以调用
call g()...    # 调用x()就是执行g()函数定义的代码
```

请注意区分返回函数和返回值：

```
def myabs():
    return abs    # 返回函数
def myabs2(x):
    return abs(x)    # 返回函数调用的结果，返回值是一个数值
```

返回函数可以把一些计算延迟执行。例如，如果定义一个普通的求和函数：

```
def calc_sum(lst):
    return sum(lst)
```

```
#调用calc_sum()函数时，将立刻计算并得到结果：
>>> calc_sum([1, 2, 3, 4])
10
```

但是，如果返回一个函数，就可以“延迟计算”：

```
def calc_sum(lst):
    def lazy_sum():
        return sum(lst)
    return lazy_sum


# 调用calc_sum()并没有计算出结果，而是返回函数：
>>> f = calc_sum([1, 2, 3, 4])
>>> f
<function lazy_sum at 0x1037bfaa0>
# 对返回的函数进行调用时，才计算出结果：
>>> f()
10
```

由于可以返回函数，我们在后续代码里就可以决定到底要不要调用该函数。

7 闭包

在函数内部定义的函数和外部定义的函数是一样的，只是他们无法被外部访问：

```
def g():
    print 'g()...'

def f():
    print 'f()...'
    return g
```

将 g 的定义移入函数 f 内部，防止其他代码调用 g：

```
def f():
    print 'f()...'
    def g():
        print 'g()...'
    return g
```

```
#但是，考察上一小节定义的 calc_sum 函数：
def calc_sum(lst):
    def lazy_sum():
        return sum(lst)
    return lazy_sum
#注意：发现没法把 lazy_sum 移到 calc_sum 的外部，因为它引用了 calc_sum 的参数 lst。
```

像这种内层函数引用了外层函数的变量（参数也算变量），然后返回内层函数的情况，称为闭包（Closure）。

闭包的特点是返回的函数还引用了外层函数的局部变量，所以，要正确使用闭包，就要确保引用的局部变量在函数返回后不能变。举例如下：

```
# 希望一次返回3个函数，分别计算1x1,2x2,3x3:
def count():
    fs = []
    for i in range(1, 4):
        def f():
             return i*i
        fs.append(f)
    return fs

f1, f2, f3 = count()
#你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果全部都是 9（请自己动手验证）。



#原因就是当count()函数返回了3个函数时，这3个函数所引用的变量 i 的值已经变成了3。由于f1、f2、f3并
>>> f1()
9     # 因为f1现在才计算i*i，但现在i的值已经变为3
```

因此，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。

8 匿名函数

高阶函数可以接收函数做参数，有些时候，我们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
>>> map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9])
[1, 4, 9, 16, 25, 36, 49, 64, 81]
#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，不写return，返回值就是该表达式的结果。

使用匿名函数，可以不必定义函数名，直接创建一个函数对象，很多时候可以简化代码：

```
>>> sorted([1, 3, 9, 5, 0], lambda x,y: -cmp(x,y))
[9, 5, 3, 1, 0]
```

返回函数的时候，也可以返回匿名函数：

```
>>> myabs = lambda x: -x if x < 0 else x
>>> myabs(-1)
1
>>> myabs(1)
1
```

9 装饰器

定义了一个函数，想在运行的时候动态增加功能，又不想改变函数本来的代码

通过高阶函数来返回一个新函数

```
def f1(x):
    return x*2
def new_fn(f):#装饰器函数
    def fn(x):
        print 'call'+f.name_+'()'
        return f(x)
    return fn
#第一种方法
g1=new_fn(f1)
print g1(5)
#第二种方法
f1=new_fn(f1)
print f1(5)#f1的原始定义函数被彻底隐藏了
```

python内置的@语法就是为了简化装饰器调用

```
@new_fn
def f1(x):
    return x*2
```

相当于

```
def f1(x):
    return x*2
f1=new_fn(f1)
```

10 编写无参数decorator

Python的 decorator 本质上就是一个高阶函数，它接收一个函数作为参数，然后，返回一个新函数。

```
#使用 decorator 用Python提供的 @ 语法，这样可以避免手动编写 f = decorate(f) 这样的代码。
#考察一个@log的定义：
def log(f):
    def fn(x):
        print 'call ' + f.__name__ + '()...'
        return f(x)
    return fn
#对于阶乘函数，@log工作得很好：
@log
def factorial(n):
    return reduce(lambda x,y: x*y, range(1, n+1))
print factorial(10)
#结果
call factorial()...
3628800
#但是，对于参数不是一个的函数，调用将报错：
@log
def add(x, y):
    return x + y
print add(1, 2)
#结果
Traceback (most recent call last):
  File "test.py", line 15, in <module>
    print add(1,2)
TypeError: fn() takes exactly 1 argument (2 given)
#因为 add() 函数需要传入两个参数，但是 @log 写死了只含一个参数的返回函数。
#要让 @log 自适应任何参数定义的函数，可以利用Python的 *args 和 **kw，保证任意个数的参数总是能正
```

```
def log(f):
    def fn(*args, **kw):
        print 'call ' + f.__name__ + '()...'
        return f(*args, **kw)
    return fn
#现在，对于任意函数，@log 都能正常工作。
```

11 编写带参数decorator

考察上一节的 @log 装饰器:

```
def log(f):
    def fn(x):
        print 'call ' + f.__name__ + '()...'
        return f(x)
    return fn
```

发现对于被装饰的函数，log打印的语句是不能变的（除了函数名）。

如果有的函数非常重要，希望打印出'[INFO] call xxx()...'，有的函数不太重要，希望打印出'[DEBUG] call xxx()...'，这时，log函数本身就需要传入'INFO'或'DEBUG'这样的参数，类似这样:

```
@log('DEBUG')
def my_func():
    pass
```

把上面的定义翻译成高阶函数的调用，就是:

```
my_func = log('DEBUG')(my_func)
```

上面的语句看上去还是比较绕，再展开一下:

```
log_decorator = log('DEBUG')
my_func = log_decorator(my_func)
```

上面的语句又相当于:

```
log_decorator = log('DEBUG')
@log_decorator
def my_func():
    pass
```

```
#所以，带参数的log函数首先返回一个decorator函数，再让这个decorator函数接收my_func并返回新函数：
def log(prefix):
    def log_decorator(f):
        def wrapper(*args, **kw):
            print '[%s] %s()...' % (prefix, f.__name__)
            return f(*args, **kw)
        return wrapper
    return log_decorator

@log('DEBUG')
def test():
    pass
print test()
```

执行结果：

```
[DEBUG] test()...
None
```

对于这种3层嵌套的decorator定义，你可以先把它拆开：

```
# 标准decorator:
def log_decorator(f):
    def wrapper(*args, **kw):
        print '[%s] %s()...' % (prefix, f.__name__)
        return f(*args, **kw)
    return wrapper
return log_decorator

# 返回decorator:
def log(prefix):
    return log_decorator(f)
```

拆开以后会发现，调用会失败，因为在3层嵌套的decorator定义中，最内层的wrapper引用了最外层的参数prefix，所以，把一个闭包拆成普通的函数调用会比较困难。不支持闭包的编程语言要实现同样的功能就需要更多的代码。

任务 上一节的@performance只能打印秒，请给 @performace 增加一个参数，允许传入's'或'ms'：

```
import time
def performance(unit):
  def j(f):
      def g(*args, **kw):
          t_start = time.time()
          r = f(*args, **kw)
```

```
            t_end = time.time()
            t = t_end - t_start
            if unit == 'ms':
                t = t * 1000
            print 'call %s () in %s %s' % (f.__name__,t, unit)
            return r
        return g
    return j

@performance('ms')
def factorial(n):
    return reduce(lambda x,y: x*y,range(1,n+1))

print factorial(10)
```

12 完善decorator

在没有decorator的情况下，打印函数名：

```
def f1(x):
    pass
print f1.__name__
#输出：f1
```

有decorator的情况下，再打印函数名：

```
def log(f):
    def wrapper(*args, **kw):
        print 'call...'
        return f(*args, **kw)
    return wrapper
@log
def f2(x):
    pass
print f2.__name__
#输出：wrapper


#可见，由于decorator返回的新函数函数名已经不是'f2'，
#而是@log内部定义的'wrapper'。这对于那些依赖函数名的代码就会失效。
#decorator还改变了函数的__doc__等其它属性。
#如果要让调用者看不出一个函数经过了@decorator的\改造"，
#就需要把原函数的一些属性复制到新函数中：
def log(f):
    def wrapper(*args, **kw):
```

```
        print 'call...'
        return f(*args, **kw)
    wrapper.__name__ = f.__name__
    wrapper.__doc__ = f.__doc__
    return wrapper
```

这样写decorator很不方便，因为我们也很难把原函数的所有必要属性都一个一个复制到新函数上，所以Python内置的functools可以用来自动化完成这个“复制”的任务：

```
import functools
def log(f):
    @functools.wraps(f)
    def wrapper(*args, **kw):
        print 'call...'
        return f(*args, **kw)
    return wrapper
```

最后需要指出，由于我们把原函数签名改成了(*args, **kw)，因此，无法获得原函数的原始参数信息。即便我们采用固定参数来装饰只有一个参数的函数：

```
def log(f):
    @functools.wraps(f)
    def wrapper(x):
        print 'call...'
        return f(x)
    return wrapper
```

也可能改变原函数的参数名，因为新函数的参数名始终是 'x'，原函数定义的参数名不一定叫 'x'。

任务

```
import time, functools
def performance(unit):
    def perf_decorator(f):
        @functools.wraps(f)
        def wrapper(*args, **kw):
            t1 = time.time()
            r = f(*args, **kw)
            t2 = time.time()
            t = (t2 - t1) * 1000 if unit=='ms' else (t2 - t1)
            print 'call %s() in %f %s' % (f.__name__, t, unit)
            return r
        return wrapper
```

```
    return perf_decorator

@performance('ms')
def factorial(n):
    return reduce(lambda x,y: x*y, range(1, n+1))
print factorial.__name__
```

13 偏函数

当一个函数有很多参数时，调用者就需要提供多个参数。如果减少参数个数，就可以简化调用者的负担。

```
#比如，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')
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)
5349
>>> int('12345', 16)
74565


#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，我们想到，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，我们转换二进制就非常方便了：
>>> int2('1000000')
64
>>> int2('1010101')
85
#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，functools.partial可以把一个参数多的函数变成一个参数少的新函数，少的参数需要在创建时指定默认值，这样，新函数调用的难度就降低了。

任务 在第7节中，我们在sorted这个高阶函数中传入自定义排序函数就可以实现忽略大小写排序。请用functools.partial把这个复杂调用变成一个简单的函数:

```
import functools
def f(x):
    return x.lower()

sorted_ignore_case = functools.partial(sorted,key=f)
print(sorted_ignore_case(['bob','about','Zoo','Credit']))
```