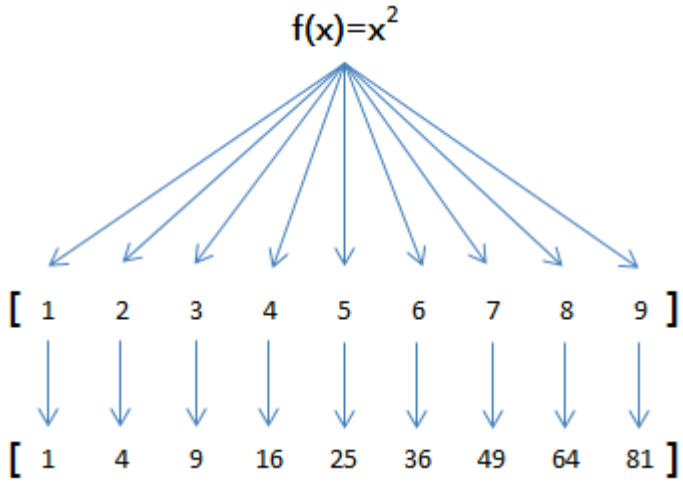
1.map()函数

map()是 Python 内置的高阶函数，它接收一个**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：



因此，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])
```

输出结果：

```
[1, 4, 9, 10, 25, 36, 49, 64, 81]
```

可以用列表替代

```
>>> print [x*x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

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。

假设用户输入的英文名字不规范，没有按照首字母大写，后续字母小写的规则，请利用 **map()**函数，把一个 list（包含若干不规范的英文名字）变成一个包含规范英文名字的 list：

输入：['adam', 'LISA', 'barT']

输出：['Adam', 'Lisa', 'Bart']

format_name(s)函数接收一个字符串，并且要返回格式化后的字符串，利用 map()函数，就可以输出新的 list。

参考代码:

```
def format_name(s):
    return s[0].upper() + s[1:].lower()
print map(format_name, ['adam', 'LISA', 'barT'])
```

2.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。**

Python 内置了求和函数 sum()，但没有求积的函数，请利用 recude()来求积：

输入：[2, 4, 5, 7, 12]

输出：2*4*5*7*12 的结果

reduce()接收的函数 f 需要两个参数，并返回一个结果，以便继续进行下一轮计算。

参考代码:

```
def prod(x, y):
    return x * y
print reduce(prod, [2, 4, 5, 7, 12])
```

3.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'

请利用 filter()过滤出 1~100 中平方根是整数的数，即结果应该是：

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

filter() 接收的函数必须判断出一个数的平方根是否是整数，而 **math.sqrt()**返回结果是浮点数。

参考代码:

```
import math
def is_sqr(x):
    r = int(math.sqrt(x))
    return r*r==x
print filter(is_sqr, range(1, 101))
```

4.自定义排序函数 sorted()

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']
```

'Zoo'排在'about'之前是因为'Z'的 ASCII 码比'a'小。

对于比较函数 cmp_ignore_case(s1, s2)，要忽略大小写比较，就是先把两个字符串都变成大写（或者都变成小写），再比较。

参考代码:

```
def cmp_ignore_case(s1, s2):
    u1 = s1.upper()
    u2 = s2.upper()
    if u1 < u2:
        return -1
    if u1 > u2:
        return 1
    return 0
print sorted(['bob', 'about', 'Zoo', 'Credit'], cmp_ignore_case)
```

5.返回函数

6.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！

7.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：

```
8.def f():
9.    print 'call f()...'
10.    # 定义函数 g:
11.    def g():
12.        print 'call g()...'
13.    # 返回函数 g:
14.    return g
```

15.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。

16.调用函数 f，我们会得到 f 返回的一个函数：

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

23.请注意区分返回函数和返回值：

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

28.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：

```
29.def calc_sum(lst):
30.    return sum(lst)
```

31.调用 calc_sum()函数时，将立刻计算并得到结果：

```
32.>>> calc_sum([1, 2, 3, 4])
33.10
```

34.但是，如果返回一个函数，就可以“延迟计算”：

```
35.def calc_sum(lst):
36.    def lazy_sum():
37.        return sum(lst)
38.    return lazy_sum
```

39.# 调用 calc_sum()并没有计算出结果，而是返回函数:

```
40.>>> f = calc_sum([1, 2, 3, 4])
41.>>> f
42.<function lazy_sum at 0x1037bfaa0>
```

43.# 对返回的函数进行调用时，才计算出结果:

```
44.>>> f()
45.10
```

46.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。

请编写一个函数 calc_prod(lst)，它接收一个 list，返回一个函数，返回函数可以计算参数的乘积。

```
def calc_prod(lst):
    def lazy_prod():
        def f(x, y):
            return x * y
        return reduce(f, lst, 1)
    return lazy_prod
f = calc_prod([1, 2, 3, 4])
print f()
```

6.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 并没有被调用，所以，此时他们并未计算 i*i，当 f1 被调用时：

```
>>> 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。

返回闭包不能引用循环变量，请改写 **count()**函数，让它正确返回能计算 1x1、2x2、3x3 的函数。

```
def f(j):
    def g():
        return j*j
    return g
```

它可以正确地返回一个闭包 g，g 所引用的变量 j 不是循环变量，因此将正常执行。

在 count 函数的循环内部，如果借助 f 函数，就可以避免引用循环变量 i。

参考代码:

```
def count():
    fs = []
    for i in range(1, 4):
        def f(j):
            def g():
                return j*j
            return g
```

```
        r = f(i)
        fs.append(r)
    return fs
f1, f2, f3 = count()
print f1(), f2(), f3()
```

7.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 $f(x)=x^2$ 时，除了定义一个 f(x)的函数外，还可以直接传入匿名函数：

```
>>> 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
```

利用匿名函数简化以下代码：

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

```
filter(is_not_empty, ['test', None, '', 'str', '  ', 'END'])
```

定义匿名函数时，没有 return 关键字，且表达式的值就是函数返回值。

参考代码:

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