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，求和的函数式这样定义的：

```
def calc_sum(*args):
    ax = 0
    for n in args:
        ax = ax + n
    return ax
```

但是，如果不需要立即求和，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 0x02183C00>
```

调用函数f时，才是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sun时，相关参数和变量都保存在返回的函数中，这种成为“闭包(Closure)”的程序结构拥有极大地威力。

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
```

闭包

注意到返回的函数在其内部引用了局部变量args，所以，当一个函数返回一个函数后，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即执行，而是知道调用了f()才执行。

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

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, f2, f3 = count()
>>> 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各函数都返回时，它们所引用的变量i已经变成了3，因此最终和结果为9。

返回闭包时牢记一点就是：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的当前的值，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))
    return fs

>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9
>>>
```