7.12 访问闭包中定义的变量¶

问题¶

你想要扩展函数中的某个闭包，允许它能访问和修改函数的内部变量。

解决方案¶

通常来讲，闭包的内部变量对于外界来讲是完全隐藏的。 但是，你可以通过编写访问函数并将其作为函数属性绑定到闭包上来实现这个目的。例如：

```
def sample():
    n = 0
    # Closure function
    def func():
        print('n=', n)

    # Accessor methods for n
    def get_n():
        return n

    def set_n(value):
        nonlocal n
        n = value

    # Attach as function attributes
    func.get_n = get_n
    func.set_n = set_n
    return func
```

下面是使用的例子:

```
>>> f = sample()
>>> f()
n= 0
>>> f.set_n(10)
>>> f()
n= 10
>>> f.get_n()
10
>>>
```

讨论¶

为了说明清楚它如何工作的，有两点需要解释一下。首先，`nonlocal` 声明可以让我们编写函数来修改内部变量的值。其次，函数属性允许我们用一种很简单的方式将访问方法绑定到闭包函数上，这个跟实例方法很像(尽管并没有定义任何类)。

还可以进一步的扩展，让闭包模拟类的实例。你要做的仅仅是复制上面的内部函数到一个字典实例中并返回它即可。例如：

```
import sys
class ClosureInstance:
    def __init__(self, locals=None):
        if locals is None:
            locals = sys._getframe(1).f_locals

        # Update instance dictionary with callables
        self.__dict__.update((key,value) for key, value in locals.items()
                             if callable(value) )
    # Redirect special methods
    def __len__(self):
        return self.__dict__['__len__']()
```

```
# Example use
def Stack():
    items = []
    def push(item):
        items.append(item)

    def pop():
        return items.pop()

    def __len__():
        return len(items)

    return ClosureInstance()
```

下面是一个交互式会话来演示它是如何工作的：

```
>>> s = Stack()
>>> s
<__main__.ClosureInstance object at 0x10069ed10>
>>> s.push(10)
>>> s.push(20)
>>> s.push('Hello')
>>> len(s)
3
>>> s.pop()
'Hello'
>>> s.pop()
20
>>> s.pop()
10
>>>
```

有趣的是，这个代码运行起来会比一个普通的类定义要快很多。你可能会像下面这样测试它跟一个类的性能对比：

```
class Stack2:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def __len__(self):
        return len(self.items)
```

如果这样做，你会得到类似如下的结果：

```
>>> from timeit import timeit
>>> # Test involving closures
>>> s = Stack()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
0.9874754269840196
>>> # Test involving a class
>>> s = Stack2()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
1.0707052160287276
>>>
```

结果显示，闭包的方案运行起来要快大概8%，大部分原因是因为对实例变量的简化访问， 闭包更快是因为不会涉及到额外的self变量。

Raymond Hettinger对于这个问题设计出了更加难以理解的改进方案。不过，你得考虑下是否真的需要在你代码中这样做， 而且它只是真实类的一个奇怪的替换而已，例如，类的主要特性如继承、属性、描述器或类方法都是不能用的。并且你要做一些其他的工作才能让一些特殊方法生效(比如上面 `ClosureInstance` 中重写过的 `__len__()` 实现。)

最后，你可能还会让其他阅读你代码的人感到疑惑，为什么它看起来不像一个普通的类定义呢？ (当然，他们也想知

道为什么它运行起来会更快)。尽管如此，这对于怎样访问闭包的内部变量也不失为一个有趣的例子。

总体上讲，在配置的时候给闭包添加方法会有更多的实用功能， 比如你需要重置内部状态、刷新缓冲区、清除缓存或其他的反馈机制的时候。