9.8 将装饰器定义为类的一部分¶

问题¶

你想在类中定义装饰器，并将其作用在其他函数或方法上。

解决方案¶

在类里面定义装饰器很简单，但是你首先要确认它的使用方式。比如到底是作为一个实例方法还是类方法。 下面我们用例子来阐述它们的不同：

```
from functools import wraps

class A:
    # Decorator as an instance method
    def decorator1(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 1')
            return func(*args, **kwargs)
        return wrapper

    # Decorator as a class method
    @classmethod
    def decorator2(cls, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 2')
            return func(*args, **kwargs)
        return wrapper
```

下面是一使用例子：

```
# As an instance method
a = A()
@a.decorator1
def spam():
    pass
# As a class method
@A.decorator2
def grok():
    pass
```

仔细观察可以发现一个是实例调用，一个是类调用。

讨论¶

在类中定义装饰器初看上去好像很奇怪，但是在标准库中有很多这样的例子。 特别的，`@property` 装饰器实际上是一个类，它里面定义了三个方法 `getter()`, `setter()`, `deleter()` ,每一个方法都是一个装饰器。例如：

```
class Person:
    # Create a property instance
    first_name = property()

    # Apply decorator methods
    @first_name.getter
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value
```

它为什么要这么定义的主要原因是各种不同的装饰器方法会在关联的 `property` 实例上操作它的状态。 因此，任何时候只要你碰到需要在装饰器中记录或绑定信息，那么这不失为一种可行方法。

在类中定义装饰器有个难理解的地方就是对于额外参数 `self` 或 `cls` 的正确使用。 尽管最外层的装饰器函数比如 `decorator1()` 或 `decorator2()` 需要提供一个 `self` 或 `cls` 参数， 但是在两个装饰器内部被创建的 `wrapper()` 函数并不需要包含这个 `self` 参数。 你唯一需要这个参数是在你确实要访问包装器中这个实例的某些部分的时候。其他情况下都不用去管它。

对于类里面定义的包装器还有一点比较难理解，就是在涉及到继承的时候。 例如，假设你想让在A中定义的装饰器作用在子类B中。你需要像下面这样写：

```
class B(A):
    @A.decorator2
    def bar(self):
        pass
```

也就是说，装饰器要被定义成类方法并且你必须显式的使用父类名去调用它。 你不能使用 `@B.decorator2` ，因为在方法定义时，这个类B还没有被创建。