8.9 创建新的类或实例属性¶

问题¶

你想创建一个新的拥有一些额外功能的实例属性类型，比如类型检查。

解决方案¶

如果你想创建一个全新的实例属性，可以通过一个描述器类的形式来定义它的功能。下面是一个例子：

```
# Descriptor attribute for an integer type-checked attribute
class Integer:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]
```

一个描述器就是一个实现了三个核心的属性访问操作(get, set, delete)的类， 分别为 `__get__()` 、 `__set__()` 和 `__delete__()` 这三个特殊的方法。 这些方法接受一个实例作为输入，之后相应的操作实例底层的字典。

为了使用一个描述器，需将这个描述器的实例作为类属性放到一个类的定义中。例如：

```
class Point:
    x = Integer('x')
    y = Integer('y')

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

当你这样做后，所有对描述器属性(比如x或y)的访问会被 `__get__()` 、 `__set__()` 和 `__delete__()` 方法捕获到。例如：

```
>>> p = Point(2, 3)
>>> p.x # Calls Point.x.__get__(p,Point)
2
>>> p.y = 5 # Calls Point.y.__set__(p, 5)
>>> p.x = 2.3 # Calls Point.x.__set__(p, 2.3)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "descrip.py", line 12, in __set__
        raise TypeError('Expected an int')
TypeError: Expected an int
>>>
```

作为输入，描述器的每一个方法会接受一个操作实例。 为了实现请求操作，会相应的操作实例底层的字典(__dict__属性)。 描述器的 `self.name` 属性存储了在实例字典中被实际使用到的key。

讨论¶

描述器可实现大部分Python类特性中的底层魔法， 包括 `@classmethod` 、 `@staticmethod` 、 `@property` ，甚至是

`__slots__` 特性。

通过定义一个描述器，你可以在底层捕获核心的实例操作(get, set, delete)，并且可完全自定义它们的行为。 这是一个强大的工具，有了它你可以实现很多高级功能，并且它也是很多高级库和框架中的重要工具之一。

描述器的一个比较困惑的地方是它只能在类级别被定义，而不能为每个实例单独定义。因此，下面的代码是无法工作的：

```
# Does NOT work
class Point:
    def __init__(self, x, y):
        self.x = Integer('x') # No! Must be a class variable
        self.y = Integer('y')
        self.x = x
        self.y = y
```

同时，`__get__()` 方法实现起来比看上去要复杂得多：

```
# Descriptor attribute for an integer type-checked attribute
class Integer:

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
```

`__get__()` 看上去有点复杂的原因归结于实例变量和类变量的不同。 如果一个描述器被当做一个类变量来访问，那么 `instance` 参数被设置成 `None` 。 这种情况下，标准做法就是简单的返回这个描述器本身即可(尽管你还可以添加其他的自定义操作)。例如：

```
>>> p = Point(2,3)
>>> p.x # Calls Point.x.__get__(p, Point)
2
>>> Point.x # Calls Point.x.__get__(None, Point)
<__main__.Integer object at 0x100671890>
>>>
```

描述器通常是那些使用到装饰器或元类的大型框架中的一个组件。同时它们的使用也被隐藏在后面。 举个例子，下面是一些更高级的基于描述器的代码，并涉及到一个类装饰器：

```
# Descriptor for a type-checked attribute
class Typed:
    def __init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('Expected ' + str(self.expected_type))
        instance.__dict__[self.name] = value
    def __delete__(self, instance):
        del instance.__dict__[self.name]

# Class decorator that applies it to selected attributes
def typeassert(**kwargs):
    def decorate(cls):
        for name, expected_type in kwargs.items():
            # Attach a Typed descriptor to the class
            setattr(cls, name, Typed(name, expected_type))
        return cls
    return decorate
```

```
# Example use
@typeassert(name=str, shares=int, price=float)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

最后要指出的一点是，如果你只是想简单的自定义某个类的单个属性访问的话就不用去写描述器了。 这种情况下使用8.6小节介绍的property技术会更加容易。 当程序中有很多重复代码的时候描述器就很有用了 (比如你想在你代码的很多地方使用描述器提供的功能或者将它作为一个函数库特性)。