8.6 创建可管理的属性¶

问题¶

你想给某个实例attribute增加除访问与修改之外的其他处理逻辑,比如类型检查或合法性验证。

解决方案¶

自定义某个属性的一种简单方法是将它定义为一个property。 例如,下面的代码定义了一个property,增加对一个属性 简单的类型检查:

```
class Person:
    def init (self, first name):
         \overline{\text{self.}} \overline{\text{first name}} = \overline{\text{first name}}
    # Getter function
    @property
    def first name(self):
        return self._first_name
    # Setter function
    @first name.setter
    def first name (self, value):
         if not isinstance (value, str):
             raise TypeError('Expected a string')
         self. first name = value
    # Deleter function (optional)
    @first name.deleter
    def first name (self):
         raise AttributeError("Can't delete attribute")
```

上述代码中有三个相关联的方法,这三个方法的名字都必须一样。 第一个方法是一个 getter 函数,它使得first_name 成为一个属性。 其他两个方法给 first_name 属性添加了 setter 和 deleter 函数。 需要强调的是只有在first_name 属性被创建后, 后面的两个装饰器 @first_name.setter 和 @first_name.deleter 才能被定义。

property的一个关键特征是它看上去跟普通的attribute没什么两样, 但是访问它的时候会自动触发 getter、setter 和 deleter 方法。例如:

```
>>> a = Person('Guido')
>>> a.first_name # Calls the getter
'Guido'
>>> a.first_name = 42 # Calls the setter
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "prop.py", line 14, in first_name
        raise TypeError('Expected a string')
TypeError: Expected a string
>>> del a.first_name
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can`t delete attribute
>>>
```

在实现一个property的时候,底层数据(如果有的话)仍然需要存储在某个地方。 因此,在get和set方法中,你会看到对 _ first_name 属性的操作,这也是实际数据保存的地方。 另外,你可能还会问为什么 _ init__() 方法中设置了 self.first_name 而不是 self._first_name 。 在这个例子中,我们创建一个property的目的就是在设置attribute的时候 进行检查。 因此,你可能想在初始化的时候也进行这种类型检查。通过设置 self.first_name ,自动调用 setter 方法,这个方法里面会进行参数的检查,否则就是直接访问 self. first_name 了。

还能在已存在的get和set方法基础上定义property。例如:

```
class Person:
   def init (self, first name):
```

```
# Getter function
def get_first_name(self):
    return self._first_name

# Setter function
def set_first_name(self, value):
    if not isinstance(value, str):
        raise TypeError('Expected a string')
    self._first_name = value

# Deleter function (optional)
def del_first_name(self):
    raise AttributeError("Can't delete attribute")

# Make a property from existing get/set methods
name = property(get_first_name, set_first_name, del_first_name)
```

讨论¶

一个property属性其实就是一系列相关绑定方法的集合。如果你去查看拥有property的类, 就会发现property本身的fget、fset和fdel属性就是类里面的普通方法。比如:

```
>>> Person.first_name.fget
<function Person.first_name at 0x1006a60e0>
>>> Person.first_name.fset
<function Person.first_name at 0x1006a6170>
>>> Person.first_name.fdel
<function Person.first_name at 0x1006a62e0>
>>>
```

通常来讲,你不会直接取调用fget或者fset,它们会在访问property的时候自动被触发。

只有当你确实需要对attribute执行其他额外的操作的时候才应该使用到property。 有时候一些从其他编程语言(比如Java) 过来的程序员总认为所有访问都应该通过getter和setter, 所以他们认为代码应该像下面这样写:

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        self._first_name = value
```

不要写这种没有做任何其他额外操作的property。 首先,它会让你的代码变得很臃肿,并且还会迷惑阅读者。 其次,它还会让你的程序运行起来变慢很多。 最后,这样的设计并没有带来任何的好处。 特别是当你以后想给普通attribute 访问添加额外的处理逻辑的时候, 你可以将它变成一个property而无需改变原来的代码。 因为访问attribute的代码还是保持原样。

Properties还是一种定义动态计算attribute的方法。 这种类型的attributes并不会被实际的存储,而是在需要的时候计算出来。比如:

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return math.pi * self.radius ** 2

    @property
```

```
def diameter(self):
    return self.radius * 2
@property
def perimeter(self):
    return 2 * math.pi * self.radius
```

在这里,我们通过使用properties,将所有的访问接口形式统一起来, 对半径、直径、周长和面积的访问都是通过属性 访问,就跟访问简单的attribute是一样的。 如果不这样做的话,那么就要在代码中混合使用简单属性访问和方法调用。 下面是使用的实例:

```
>>> c = Circle(4.0)

>>> c.radius

4.0

>>> c.area # Notice lack of ()

50.26548245743669

>>> c.perimeter # Notice lack of ()

25.132741228718345

>>>
```

尽管properties可以实现优雅的编程接口,但有些时候你还是会想直接使用getter和setter函数。例如:

```
>>> p = Person('Guido')
>>> p.get_first_name()
'Guido'
>>> p.set_first_name('Larry')
>>>
```

这种情况的出现通常是因为Python代码被集成到一个大型基础平台架构或程序中。 例如,有可能是一个Python类准备加入到一个基于远程过程调用的大型分布式系统中。 这种情况下,直接使用get/set方法(普通方法调用)而不是property或许会更容易兼容。

最后一点,不要像下面这样写有大量重复代码的property定义:

```
class Person:
   def init (self, first name, last name):
       self.first name = first name
       self.last_name = last_name
   @property
   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
   # Repeated property code, but for a different name (bad!)
   @property
   def last name(self):
       return self._last_name
   @last name.setter
   def last name (self, value):
       if not isinstance (value, str):
            raise TypeError('Expected a string')
       self. last name = value
```

重复代码会导致臃肿、易出错和丑陋的程序。好消息是,通过使用装饰器或闭包,有很多种更好的方法来完成同样的事情。可以参考8.9和9.21小节的内容。