8.8 子类中扩展property¶

问题¶

在子类中，你想要扩展定义在父类中的property的功能。

解决方案¶

考虑如下的代码，它定义了一个property：

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

    # Getter function
    @property
    def name(self):
        return self._name

    # Setter function
    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._name = value

    # Deleter function
    @name.deleter
    def name(self):
        raise AttributeError("Can't delete attribute")
```

下面是一个示例类，它继承自Person并扩展了 `name` 属性的功能：

```
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

接下来使用这个新类：

```
>>> s = SubPerson('Guido')
Setting name to Guido
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
Setting name to Larry
>>> s.name = 42
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 16, in name
        raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

如果你仅仅只想扩展property的某一个方法，那么可以像下面这样写：

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```

或者，你只想修改setter方法，就这么写：

```
class SubPerson(Person):
    @Person.name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)
```

讨论¶

在子类中扩展一个property可能会引起很多不易察觉的问题， 因为一个property其实是 `getter`、`setter` 和 `deleter` 方法的集合，而不是单个方法。 因此，当你扩展一个property的时候，你需要先确定你是否要重新定义所有的方法还是说只修改其中某一个。

在第一个例子中，所有的property方法都被重新定义。 在每一个方法中，使用了 `super()` 来调用父类的实现。 在 `setter` 函数中使用 `super(SubPerson, SubPerson).name.__set__(self, value)` 的语句是没有错的。 为了委托给之前定义的setter方法，需要将控制权传递给之前定义的name属性的 `__set__()` 方法。 不过，获取这个方法的唯一途径是使用类变量而不是实例变量来访问它。 这也是为什么我们要使用 `super(SubPerson, SubPerson)` 的原因。

如果你只想重定义其中一个方法，那只使用 @property 本身是不够的。比如，下面的代码就无法工作：

```
class SubPerson(Person):
    @property  # Doesn't work
    def name(self):
        print('Getting name')
        return super().name
```

如果你试着运行会发现setter函数整个消失了：

```
>>> s = SubPerson('Guido')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 5, in __init__
        self.name = name
AttributeError: can't set attribute
>>>
```

你应该像之前说过的那样修改代码：

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```

这么写后，property之前已经定义过的方法会被复制过来，而getter函数被替换。然后它就能按照期望的工作了：

```
>>> s = SubPerson('Guido')
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
>>> s.name
Getting name
'Larry'
>>> s.name = 42
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
```

```
    File "example.py", line 16, in name
        raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

在这个特别的解决方案中，我们没办法使用更加通用的方式去替换硬编码的 `Person` 类名。 如果你不知道到底是哪个基类定义了property， 那你只能通过重新定义所有property并使用 `super()` 来将控制权传递给前面的实现。

值得注意的是上面演示的第一种技术还可以被用来扩展一个描述器(在8.9小节我们有专门的介绍)。比如：

```
# A descriptor
class String:
    def __init__(self, name):
        self.name = name

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

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

# A class with a descriptor
class Person:
    name = String('name')

    def __init__(self, name):
        self.name = name

# Extending a descriptor with a property
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

最后值得注意的是，读到这里时，你应该会发现子类化 `setter` 和 `deleter` 方法其实是很简单的。 这里演示的解决方案同样适用，但是在 Python的issue页面 报告的一个bug，或许会使得将来的Python版本中出现一个更加简洁的方法。