Python property 介绍

简介

最近看 Effective Python 第四章元类及属性。其中经常出现`@property`装饰器。因此总结一下。我理解`@property`的一个比较直观的好处是可以创建只读的属性，这样可以防止属性呗随意更改。

Property 是 Built-in Functions

> *参考 Python3.8 Built-in Functions*

```
class property(fget=None, fset=None, fdel=None, doc=None)
"""
Return a property attribute.

fget is a function for getting an attribute value. fset is a
function for setting an attribute value. fdel is a function for
deleting an attribute value. And doc creates a docstring for
the attribute.
"""
```

注意`fget, fset, fdel`都是函数，用来读取属性，对属性赋值和删除属性。`doc`用于创建该属性的文档。

直接使用时，如下所示

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

```

```
    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

测试如下，`doc` 用于 `help(C.x)`。实例化之后`x`绑定`c._x`这个属性。同时`getx, setx, delx`分别对应`fget, fsset, fdel`可以通过`help(C.x)`查看对应的文档。

```
print(C.x)
c = C()
print(c.x)
help(C.x)
"""
<property object at 0x7f8fcea06180>
None
Help on property:

    I'm the 'x' property.
"""
```

Property 也可用作装饰器

注意 property 对象包含`getter, setter, deleter`方法，可以被用于装饰器，需要注意的是，`setter deleter`方法的名称必须与相关属性保持一致。

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter # 次数调用方法
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Property 实现只读的属性

定义`Resistor`电阻器类:

```
class Resistor:
    def __init__(self, ohms):
        self._ohms = ohms

    @property
    def ohms(self):
        return self._ohms
```

由于没有设定`setter, deleter`当外界想要对属性赋值时会报错。这样可以实现只读的属性。

```
resistor = Resistor(5)
print(resistor.ohms)
resistor.ohms = 5
"""
5
Traceback (most recent call last):
  File "/demo.py", line 28, in <module>
    resistor.ohms = 5
AttributeError: can't set attribute
"""
```

Property 可以做属性的类型和数值验证

这是 Effecive Python里面的内容，比较有意思。定义电阻器类和其派生类:

```
class Resistor:
    def __init__(self, ohms):
        self.ohms = ohms
        self.voltage = 0
        self.current = 0

class BoundedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
```

```
        if ohms <= 0:
            raise ValueError('{} ohms must be > 0'.format(ohms))
        self._ohms = ohms
```

首先定义一个有效的类，传入无效值

```
r3 = BoundedResistance(1e3)
r3.ohms = 0
"""
Traceback (most recent call last):
  File "/demo.py", line 39, in <module>
    r3.ohms = 0
  File "/demo.py", line 35, in ohms
    raise ValueError('{} ohms must be > 0'.format(ohms))
ValueError: 0 ohms must be > 0
"""
```

当产生实例时，我理解的流程是这样：

1. `super().__init__(ohms)` 在`Resistor`中执行`self.ohms = ohms`
2. 父类中的`self.ohms = ohms` 使得子类的`@ohms.setter`执行，此时实例中有`self._ohms = ohms`
3. 最后生成的实例中，有三个属性`self._ohms, self.current, self.voltage`

验证如下，只能说 python 真的是太动态了。

```
r3 = BoundedResistance(1e3)
print(r3.__dict__)
"""
{'_ohms': 1000.0, 'voltage': 0, 'current': 0}
"""
```

书中还指出直接给构造器传入无效值的时候，也会引发异常。

```
BoundedResistance(-5)
```

因为在执行`Resistor.__init__` 中`self.ohms = -5` 时，会使 `@ohms.setter` 得以执行，所以在对象构造完毕之前，程序会通过 `setter` 做一步验证。

property 还可以防止属性遭到更改

```
class FixedResistance(Resistor):
    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if hasattr(self, '_ohms'):
            raise AttributeError("Can't set attribute")
        self._ohms = ohms
```

测试如下：

```
r4 = FixedResistance(1e3)
print(r4.ohms)
print(r4._ohms) # 打印 protected attribute
r4.ohms = 2e3
"""1000.0
1000.0
Traceback (most recent call last):
  File "/demo.py", line 54, in <module>
    r4.ohms = 2e3
  File "/demo.py", line 47, in ohms
    raise AttributeError("Can't set attribute")
AttributeError: Can't set attribute
"""
```

第一次构造出实例的时候，子类生成了一个属性`_ohms`后面重复赋值的时候会报错。

`hasattr(object, str)`判断对象是否有对应的属性。