目录

19 让你的模子更好用：类进阶

更新时间：2019-09-30 10:14:50

> 受苦的人，没有悲观的权利。
>
> ——尼采

类属性和类方法

之前介绍类的时候，我们学习了对象属性和对象方法。对象属性和对象方法是绑定在对象这个层次上的，也就是说需要先创建对象，然后才能使用对象的属性和方法。

即：

```
对象 = 类()

对象.属性
对象.方法()
```

除此之外，还有一种**绑定在类这个层面**的属性和方法，叫作类属性和类方法。使用类属性和类方法时，不用创建对象，直接通过类来使用。

类属性和类方法的使用方式：

```
类.属性
类.方法()
```

类属性的定义

类属性如何定义呢？

只要将属性定义在类之中方法之外即可。如下面的 属性1 和 属性2 ：

```
    属性2 = Y

    def 某方法():
        pass
```

举个例子:

```
class Char:
    letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    digits = '0123456789'
```

这里定义了类 Char ，有两个类属性，这两个类属性分别包含所有大写字母和所有数字。可以通过类名来使用这两个类属性，此时无需创建对象：

```
>>> Char.letters
' ABCDEFGHIJKLMNOPQRSTUVWXYZ '
>>> Char.digits
' 0123456789 '
```

当然，类所创建出来的对象也能使用类属性：

```
>>> char = Char()
>>> char.letters
' ABCDEFGHIJKLMNOPQRSTUVWXYZ '
>>> char.digits
' 0123456789 '
```

类方法的定义

再来看下类方法的定义方法。类方法的定义需要借助于装饰器，装饰器具体是什么后续文章中会介绍，目前只要知道用法即可。

定义类方法时，需要在方法的前面加上装饰器 @classmethod 。如下：

```
class 类:
    @classmethod
    def 类方法(cls):
        pass
```

注意与对象方法不同，类方法的第一个参数通常命名为 cls ，表示当前这个类本身。我们可以通过该参数来引用类属性，或类中其它类方法。

类方法中可以使用该类的类属性，但不能使用该类的对象属性。因为类方法隶属于类，而对象属性隶属于对象，使用类方法时可能还没有对象被创建出来。

在之前 Char 类的基础上，我们加上随机获取任意字符的类方法。代码如下：

```
class Char:
    letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    digits = '0123456789'

    @classmethod
    def random_letter(cls):
        return random.choice(cls.letters)

    @classmethod
    def random_digits(cls):
        return random.choice(cls.digits)
```

方法 random_letter() 可以从属性 letters 随机获取一个大写字母；方法 random_digits() 可以从属性 digits 随机获取一个数字。它们函数体中的 random.choice() 可从指定序列中随机获取一个元素。

```
>>> Char.random_digits()
'8'
>>> Char.random_letter()
'X'
```

扩展： import 语句不仅可用于模块的开头，也可用于模块的任意位置，如函数中。

静态方法

与类方法有点相似的是静态方法，静态方法也可直接通过类名来调用，不必先创建对象。不同在于类方法的第一个参数是类自身（ cls ），而静态方法没有这样的参数。如果方法需要和其它类属性或类方法交互，那么可以将其定义成类方法；如果方法**无需**和其它类属性或类方法交互，那么可以将其定义成静态方法。

定义静态方法时，需要在方法的前面加上装饰器 @staticmethod 。如下：

```
class 类:
    @staticmethod
    def 静态方法():
        pass
```

之前的例子中，我们可以从类属性 letters 和 digits 中随机获取字符，如果想要自己来指定字符的范围，并从中获取一个随机字符，可以再来定义一个静态方法 random_char() 。如：

```
import random

class Char:
    letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    digits = '0123456789'

    @classmethod
    def random_letter(cls):
        return random.choice(cls.letters)
```

```
        return random.choice(cls.digits)

    @staticmethod
    def random_char(string):
        if not isinstance(string, str):
            raise TypeError('需要字符串参数')

        return random.choice(string)
```

静态方法 random_char 从传入的字符串中随机挑选出一个字符。之所以定义成静态方法，是因为它无需与类属性交互。

```
>>> Char.random_char(‘imooc2019’)
‘0’
>>> Char.random_char(‘imooc2019’)
‘m’
```

私有属性、方法

类属性 letters 和 digits 是为了提供给同一个类中的类方法使用，但我们可以通过类或对象从类的外部直接访问它们。比如：

```
Char.letters
Char.digits
```

```
>>> Char.letters
’ABCDEFGHIJKLMNOPQRSTUVWXYZ’
>>> Char.digits
’0123456789’
```

有时我们不想把过多的信息暴露出去，有没有什么方法来限制属性不被类外部所访问，而是只能在类中使用？

答案是有的，我们只需要在命名上动动手脚，将属性或方法的名称用 __（两个下划线）开头即可。如：

```
import random

class Char:
    __letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    __digits = '0123456789'

    @classmethod
    def random_letter(cls):
        return random.choice(cls.__letters)

    @classmethod
```

从类外部访问这两个属性看看：

```
>>> Char.__letters
Traceback (most recent call last):
    File “ ”, line 1, in
AttributeError: type object ‘Char’ has no attribute ‘__letters’
```

```
>>> Char.__digits
Traceback (most recent call last):
    File “ ”, line 1, in
AttributeError: type object ‘Char’ has no attribute ‘__digits’
```

可以看到，修改过后的属性不能直接被访问了，解释器抛出 AttributeError 异常，提示类中没有这个属性。

但位于同一个类中的方法还是可以正常使用这些属性：

```
>>> Char.random_letter()
‘N’
>>> Char.random_digits()
‘4’
```

像这样以 __（两个下划线）开头的属性我们称为**私有属性**。顾名思义，它是类所私有的，不能在类外部使用。

上述是以类属性作为示例，该规则对类方法、对象属性、对象方法同样适用。只需在名称前加上 __（两个下划线）即可。

我们也可以使用 _（一个下划线）前缀来声明某属性或方法是私有的，但是这种形式只是一种使用者间的约定，并不在解释器层面作限制。如：

```
class Char:
    _letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    _digits = '0123456789'
```

上面的 _letters 和 _digits 也可看作私有属性，只不过是约定上的私有，通过名称前缀 _（一个下滑线）向使用者告知这是私有的。但你如果非要使用，依然可以用。

```
>>> Char._letters
’ABCDEFGHIJKLMNOPQRSTUVWXYZ’
```

特殊方法

类中以 __ 开头并以 __ 结尾的方法是特殊方法，特殊方法有特殊的用途。它们可以直接调用，也可以通过一些内置函数或操作符来间接调用，如之前学习过的 __init__() 、__next__() 。

特殊方法很多，在这里我们简单例举几个：

- __init__()

  __init__() 是非常典型的一个特殊方法，它用于对象的初始化。在实例化类的过程中，被自动调用。

- __next__()

  在迭代器章节中我们讲过，对迭代器调用 next() 函数，便能生成下一个值。这个过程的背后，next() 调用了迭代器的 __next__() 方法。

- __len__()

  你可能会好奇，为什么调用 len() 函数时，便能返回一个容器的长度？原因就是容器类中实现了 __len__() 方法，调用 len() 函数时将自动调用容器的 __len__() 方法。

- __str__()

  在使用 print() 函数时将自动调用类的 __str__() 方法。如：

```
class A:
    def __str__(self):
        return '这是 A 的对象'
```

```
>>> a = A()
>>> print(a)
这是 A 的对象`
```

- __getitem__()

  诸如列表、元素、字符串这样的序列，我们可以通过索引的方式来获取其中的元素，这背后便是 __getitem__() 在起作用。

  'abc'[2] 即等同于 'abc'.__getitem__(2) 。

```
>>> ‘abc’[2]
‘c’
```

继承

如果想基于一个现有的类，获取其全部能力，并以此扩展出一个更强大的类，此时可以使用类的**继承**。被继承的类叫作父类（或基类），继承者叫作子类（或派生类）。

定义时，子类名称的后面加上括号并写入父类。如下：

```
class 父类:
    父类的实现

class 子类(父类):
        子类的实现
```

例如：

```
class A:
    def __init__(self):
        self.apple = 'apple'

    def have(self):
        print('I hava an', self.apple)

class B(A):
        def who(self):
                print('I am an object of B')
```

```
>>> b = B()

>>> b.who()
I am an object of B

>>> b.apple
’ apple ’

>>> b.have()
I hava an apple
```

可以看到，虽然类 B 中什么都没定义，但由于 B 继承自 A，所以它拥有 A 的属性和方法。

子类 B 中当然也可以定义自己的属性。

```
class B(A):
        def __init__(self):
                super().__init__()
                self.banana = 'banana'
```

```
>>> b.banana
’ banana ’
```

我们在 B 中定义 __init__() 方法，并在其中定义了 B 自己的属性 banana 。

super().__init__() 这一句代码是什么作用？由于我们在子类中定义了 __init__() 方法，这会导致子类无法再获取父类的属性，加上这行代码就能在子类初始化的同时初始化父类。super() 用在类的方法中时，返回父类对象。

子类中出现和父类同名的方法会怎么样？答案是子类会覆盖父类的同名方法。

```
class A:
    def __init__(self):
        self.apple = 'apple'

    def have(self):
        print('I hava an', self.apple)

class B(A):
        def __init__(self):
                super().__init__()
                self.banana = 'banana'

        def have(self):
                print('I hava an', self.banana)
```

```
>>> b = B()
>>> b.have()
I hava an banana
```

继承链

子类可以继承父类，同样的，父类也可以继承它自己的父类，如此一层一层继承下去。

```
class A:
        def have(self):
                print('I hava an apple')

class B(A):
        pass

class C(B):
        pass
```

```
>>> c = C()
>>> c.have()
I hava an apple
```

其实 A 也有继承，它继承自 object 。任何类的根源都是 object 类。如果一个类没有指定所继承的类，那么它默认继承 object 。

A 中也可以显式指明其继承于 object ：

```
class A(object):
    def have(self):
        print('I hava an apple')
```

如果想要判断一个类是否是另一个类的子类，可以使用内置函数 issubclass() 。用法如下：

```
>>> issubclass(C, A)
True
>>> issubclass(B, A)
True
>>> issubclass(C, B)
True
```

多继承

子类可以同时继承多个父类，这样它便拥有了多份能力。

定义时，子类名称后面加上括号并写入多个父类。如下：

```
class A:
    def get_apple(self):
        return 'apple'

class B:
    def get_banana(self):
        return 'banana'

class C(A, B):
    pass
```

```
>>> c = C()
>>> c.get_apple()
 'apple'
>>> c.get_banana()
 'banana'
```

此时 C 便同时拥有了 A 和 B 的能力。

← 18 把盒子升级为豪宅：函数进阶

20 从小独栋升级为别墅区：函数式编程 →

目录

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论

千学不如一看，千看不如一练