关于作者
廖雪峰 北京 朝阳区
+加关注

定制类

阅读: 418752

看到类似`__slots__`这种形如`__xxx__`的变量或者函数名就要注意，这些在Python中是有特殊用途的。

`__slots__`我们已经知道怎么用了，`__len__()`方法我们也知道是为了能让class作用于`len()`函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

__str__

我们先定义一个`Student`类，打印一个实例：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...
>>> print(Student('Michael'))
<__main__.Student object at 0x109afb190>
```

打印出一堆`<__main__.Student object at 0x109afb190>`，不好看。

怎么才能打印得好看呢？只需要定义好`__str__()`方法，返回一个好看的字符串就可以了：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return 'Student object (name: %s)' % self.name
...
>>> print(Student('Michael'))
Student object (name: Michael)
```

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用`print`，打印出来的实例还是不好看：

```
>>> s = Student('Michael')
>>> s
<__main__.Student object at 0x109afb310>
```

这是因为直接显示变量调用的不是`__str__()`，而是`__repr__()`，两者的区别是`__str__()`返回用户看到的字符串，而`__repr__()`返回程序开发者看到的字符串，也就是说，`__repr__()`是为调试服务的。

解决办法是再定义一个`__repr__()`。但是通常`__str__()`和`__repr__()`代码都是一样的，所以，有个偷懒的写法：

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name=%s)' % self.name
    __repr__ = __str__
```

__iter__

如果一个类想被用于`for ... in`循环，类似list或tuple那样，就必须实现一个`__iter__()`方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的`__next__()`方法拿到循环的下一个值，直到遇到`StopIteration`错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器a，b

    def __iter__(self):
        return self # 实例本身就是迭代对象，故返回自己

    def __next__(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 100000: # 退出循环的条件
            raise StopIteration()
        return self.a # 返回下一个值
```

现在，试试把Fib实例作用于for循环：

```
>>> for n in Fib():
...     print(n)
...
1
1
2
3
5
...
46368
75025
```

__getitem__

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

```
>>> Fib()[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Fib' object does not support indexing
```

要表现得像list那样按照下标取出元素，需要实现`__getitem__()`方法：

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in range(n):
            a, b = b, a + b
        return a
```

现在，就可以按下标访问数列的任意一项了：

```
>>> f = Fib()
>>> f[0]
1
>>> f[1]
1
>>> f[2]
2
>>> f[3]
3
>>> f[10]
89
>>> f[100]
573147844013817084101
```

但是list有个神奇的切片方法：

```
>>> list(range(100))[5:10]
[5, 6, 7, 8, 9]
```

对于Fib却报错。原因是`__getitem__()`传入的参数可能是一个int，也可能是一个切片对象`slice`，所以要做判断：

```
class Fib(object):
    def __getitem__(self, n):
        if isinstance(n, int): # n是索引
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice): # n是切片
            start = n.start
            stop = n.stop
            if start is None:
                start = 0
            a, b = 1, 1
            L = []
            for x in range(stop):
                if x >= start:
                    L.append(a)
                a, b = b, a + b
            return L
```

现在试试Fib的切片：

```
>>> f = Fib()
>>> f[0:5]
[1, 1, 2, 3, 5]
>>> f[:10]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

但是没有对step参数作处理：

```
>>> f[:10:2]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

也没有对负数作处理，所以，要正确实现一个`__getitem__()`还是有很多工作要做的。

此外，如果把对象看成`dict`，`__getitem__()`的参数也可能是一个可以作key的object，例如`str`。

与之对应的是`__setitem__()`方法，把对象视作list或dict来对集合赋值。最后，还有一个`__delitem__()`方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

__getattr__

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。比如定义`Student`类：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'
```

调用`name`属性，没问题，但是，调用不存在的`score`属性，就有问题了：

```
>>> s = Student()
>>> print(s.name)
Michael
>>> print(s.score)
Traceback (most recent call last):
  ...
AttributeError: 'Student' object has no attribute 'score'
```

错误信息很清楚地告诉我们，没有找到`score`这个attribute。

要避免这个错误，除了可以加上一个`score`属性外，Python还有另一个机制，那就是写一个`__getattr__()`方法，动态返回一个属性。修改如下：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'

    def __getattr__(self, attr):
        if attr=='score':
            return 99
```

当调用不存在的属性时，比如`score`，Python解释器会试图调用`__getattr__(self, 'score')`来尝试获得属性，这样，我们就有机会返回`score`的值：

```
>>> s = Student()
>>> s.name
'Michael'
>>> s.score
99
```

返回函数也是完全可以的：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
```

只是调用方式要变为：

```
>>> s.age()
25
```

注意，只有在没有找到属性的情况下，才调用`__getattr__`，已有的属性，比如`name`，不会在`__getattr__`中查找。

此外，注意到任意调用如`s.abc`都会返回`None`，这是因为我们定义的`__getattr__`默认返回就是`None`。要让class只响应特定的几个属性，我们就要按照约定，抛出`AttributeError`的错误：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
        raise AttributeError('\'Student\' object has no attribute \'%s\'' % attr)
```

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

举个例子：

现在很多网站都搞REST API，比如新浪微博、豆瓣啥的，调用API的URL类似：

- http://api.server/user/friends
- http://api.server/user/timeline/list

如果要写SDK，给每个URL对应的API都写一个方法，那得累死，而且，API一旦改动，SDK也要改。

利用完全动态的`__getattr__`，我们可以写出一个链式调用：

```
class Chain(object):

    def __init__(self, path=''):
        self._path = path

    def __getattr__(self, path):
        return Chain('%s/%s' % (self._path, path))

    def __str__(self):
        return self._path

    __repr__ = __str__
```

试试：

```
>>> Chain().status.user.timeline.list
'/status/user/timeline/list'
```

这样，无论API怎么变，SDK都可以根据URL实现完全动态的调用，而且，不随API的增加而改变！

还有些REST API会把参数放到URL中，比如GitHub的API：

```
GET /users/:user/repos
```

调用时，需要把`:user`替换为实际用户名。如果我们能写出这样的链式调用：

```
Chain().users('michael').repos
```

就可以非常方便地调用API了。有兴趣的童鞋可以试试写出来。

__call__

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用`instance.method()`来调用。能不能直接在实例本身上调用呢？在Python中，答案是肯定的。

任何类，只需要定义一个`__call__()`方法，就可以直接对实例进行调用。请看示例：

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

    def __call__(self):
        print('My name is %s.' % self.name)
```

调用方式如下：

```
>>> s = Student('Michael')
>>> s() # self参数不要传入
My name is Michael.
```

`__call__()`还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个`Callable`对象，比如函数和我们上面定义的带有`__call__()`的类实例：

```
>>> callable(Student())
True
>>> callable(max)
True
>>> callable([1, 2, 3])
False
>>> callable(None)
False
>>> callable('str')
False
```

通过`callable()`函数，我们就可以判断一个对象是否是“可调用”对象。

小结

Python的class允许定义许多定制方法，可以让我们非常方便地生成特定的类。

本节介绍的是最常用的几个定制方法，还有很多可定制的方法，请参考Python的官方文档。

参考源码

special_str.py

special_iter.py

special_getitem.py

special_getattr.py

special_call.py

读后有收获可以请作者喝咖啡，读后有疑问请加群讨论：

还可以分享给朋友：

分享到微博

上一页 下一页

评论

对这章内容的一点理解

東NANIAN created at April 10, 2019 2:15 PM, Last updated at June 10, 2019 12:23 AM

实现`Chain().users('michael').repos`输出`/users/michael/repos`

无图无真相，上代码：

```
class Chain(object):
    def __init__(self, path=''):
        self.__path = path

    def __getattr__(self, path):
        return Chain('%s/%s' % (self.__path, path))
```