[python特殊属性和方法的运用](http://blog.csdn.net/business122/article/details/7568446)

分类： [Python](http://blog.csdn.net/business122/article/category/1137872)2012-05-15 13:51 506人阅读 [评论](http://blog.csdn.net/business122/article/details/7568446#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/business122/article/details/7568446#report)

[python](http://www.csdn.net/tag/python)[class](http://www.csdn.net/tag/class)[cgi](http://www.csdn.net/tag/cgi)[解密](http://www.csdn.net/tag/%e8%a7%a3%e5%af%86)[import](http://www.csdn.net/tag/import)[file](http://www.csdn.net/tag/file)

目录[(?)[+]](http://blog.csdn.net/business122/article/details/7568446)

用\_\_dict\_\_把dict转换为对象的属性

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

1 class Messenger:

2 def \_\_init\_\_(self, \*\*kwargs):

3 self.\_\_dict\_\_ = kwargs

4

5

6 m = Messenger(info="some information", b=[’a’, ’list’])

7 m.more = 11

8 print m.info, m.b, m.more

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

*以下为转载：*

Python 用下划线作为变量前缀和后缀指定特殊变量

\_xxx 不能用’from module import \*’导入

\_\_xxx\_\_ 系统定义名字

\_\_xxx 类中的私有变量名

核心风格：避免用下划线作为变量名的开始。

因为下划线对解释器有特殊的意义，而且是内建标识符所使用的符号，我们建议程序员避免用下划线作为变量名的开始。一般来讲，变量名\_xxx被看作是“私有的”，在模块或类外不可以使用。当变量是私有的时候，用\_xxx 来表示变量是很好的习惯。因为变量名\_\_xxx\_\_对Python 来说有特殊含义，对于普通的变量应当避免这种命名风格。

“单下划线” 开始的成员变量叫做保护变量，意思是只有类对象和子类对象自己能访问到这些变量；  
“双下划线” 开始的是私有成员，意思是只有类对象自己能访问，连子类对象也不能访问到这个数据。

以单下划线开头（\_foo）的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用“from xxx import \*”而导入；以双下划线开头的（\_\_foo）代表类的私有成员；以双下划线开头和结尾的（\_\_foo\_\_）代表python里特殊方法专用的标识，如 \_\_init\_\_（）代表类的构造函数。

现在我们来总结下所有的系统定义属性和方法， 先来看下保留属性：

>>> Class1.\_\_doc\_\_ # 类型帮助信息 'Class1 Doc.'

>>> Class1.\_\_name\_\_ # 类型名称 'Class1'

>>> Class1.\_\_module\_\_ # 类型所在模块 '\_\_main\_\_'

>>> Class1.\_\_bases\_\_ # 类型所继承的基类 (<type 'object'>,)

>>> Class1.\_\_dict\_\_ # 类型字典，存储所有类型成员信息。 <dictproxy object at 0x00D3AD70>

>>> Class1().\_\_class\_\_ # 类型 <class '\_\_main\_\_.Class1'>

>>> Class1().\_\_module\_\_ # 实例类型所在模块 '\_\_main\_\_'

>>> Class1().\_\_dict\_\_ # 对象字典，存储所有实例成员信息。 {'i': 1234}

接下来是保留方法，可以把保留方法分类：

**类的基础方法**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 初始化一个实例 | x = MyClass() | x.*\_\_init\_\_*() |
| **②** | 字符串的“官方”表现形式 | *repr*(x) | x.*\_\_repr\_\_*() |
| **③** | 字符串的“非正式”值 | *str*(x) | x.*\_\_str\_\_*() |
| **④** | 字节数组的“非正式”值 | *bytes*(x) | x.*\_\_bytes\_\_*() |
| **⑤** | 格式化字符串的值 | format(x, *format\_spec*) | x.*\_\_format\_\_*(*format\_spec*) |

1. 对 \_\_init\_\_() 方法的调用发生在实例被创建 *之后* 。如果要控制实际创建进程，请使用 [\_\_new\_\_() 方法](http://www.4pang.com/2010/08/20/4%E8%83%96%E5%AD%A6python-%EF%BC%8D-%E7%89%B9%E6%AE%8A%E5%B1%9E%E6%80%A7%E5%92%8C%E6%96%B9%E6%B3%95%E7%9A%84%E6%80%BB%E7%BB%93.html#esoterica)。
2. 按照约定， \_\_repr\_\_()方法所返回的字符串为合法的 Python 表达式。
3. 在调用 print(x) 的同时也调用了 \_\_str\_\_()方法。
4. 由于 bytes 类型的引入而*从 Python 3 开始出现*。

**行为方式与迭代器类似的类**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 遍历某个序列 | *iter*(seq) | seq.*\_\_iter\_\_*() |
| **②** | 从迭代器中获取下一个值 | *next*(seq) | seq.*\_\_next\_\_*() |
| **③** | 按逆序创建一个迭代器 | *reversed*(seq) | seq.*\_\_reversed\_\_()* |

1. 无论何时创建迭代器都将调用 \_\_iter\_\_()方法。这是用初始值对迭代器进行初始化的绝佳之处。
2. 无论何时从迭代器中获取下一个值都将调用 \_\_next\_\_()方法。
3. \_\_reversed\_\_() 方法并不常用。它以一个现有序列为参数，并将该序列中所有元素从尾到头以逆序排列生成一个新的迭代器。

Sample：

**Python代码  [收藏代码](javascript:void())**

1. **class** Itr(object):
2. **def** \_\_init\_\_(self):
3. self.result = ['a', 'b', 'c', 'd']
4. self.i = iter(self.result)

7. **def** \_\_call\_\_(self):
8. res = next(self.i)
9. **print**("\_\_call\_\_ called, which would return ", res)
10. **return** res

13. **def** \_\_iter\_\_(self):
14. **print**("\_\_iter\_\_ called")
15. **return** iter(self.result)

18. itr = Itr()
19. # i1必须是callable的，否则无法返回callable-iterator
20. i1 = iter(itr, 'c')
21. **print**("i1 = ", i1)
22. # i2只需要类实现\_\_iter\_\_函数即可返回
23. i2 = iter(itr)
24. **print**("i2 = ", i2)
26. **for** i **in** i1:
27. **print**(i)
29. **for** i **in** i2:
30. **print**(i)

输出结果是：

**Python代码  [收藏代码](javascript:void())**

1. i1 =  <callable\_iterator object at 0x1349710>
2. \_\_iter\_\_ called
3. i2 =  <list\_iterator object at 0x133a090>
4. \_\_call\_\_ called, which would **return**  a
5. a
6. \_\_call\_\_ called, which would **return**  b
7. b
8. \_\_call\_\_ called, which would **return**  c
9. a
10. b
11. c
12. d

可以看到传入两个参数的i1的类型是一个callable\_iterator，它每次在调用的时候，都会调用\_\_cal\_\_函数，并且最后到c就停止了。   
而i2就简单的多，itr把自己类中一个容器的迭代器返回就可以了。   
  
有朋友可能不满意，对上面的例子只是为了介绍iter()函数传两个参数的功能而写，如果真正想写一个iterator的类，需要使用\_\_next\_\_函数。这个函数每次返回一个值就可以实现迭代了。

**Python代码  [收藏代码](javascript:void())**

1. **class** Next(object):
2. **def** \_\_init\_\_(self, data = 1):
3. self.data = data

6. **def** \_\_iter\_\_(self):
7. **return** self

10. **def** \_\_next\_\_(self):
11. **print**("\_\_next\_\_ called")
12. **if** self.data > 5:
13. **raise** StopIteration
14. **else**:
15. self.data += 1
16. **return** self.data
17. **for** i **in** Next(3):
18. **print**(i)

输出结果是：

**Python代码  [收藏代码](javascript:void())**

1. \_\_next\_\_ called
2. 4
3. \_\_next\_\_ called
4. 5
5. \_\_next\_\_ called
6. 6
7. \_\_next\_\_ called

很简单把。唯一需要注意下的就是\_\_next\_\_中必须控制iterator的结束条件，不然就死循环了。

**计算属性**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 获取一个计算属性（无条件的） | x.my\_property | x.*\_\_getattribute\_\_*(*'my\_property'*) |
| **②** | 获取一个计算属性（后备） | x.my\_property | x.*\_\_getattr\_\_*(*'my\_property'*) |
| **③** | 设置某属性 | x.my\_property = value | x.*\_\_setattr\_\_*(*'my\_property'*,*value*) |
| **④** | 删除某属性 | del x.my\_property | x.*\_\_delattr\_\_*(*'my\_property'*) |
| **⑤** | 列出所有属性和方法 | dir(x) | x.*\_\_dir\_\_*() |

1. 如果某个类定义了 \_\_getattribute\_\_() 方法，在 *每次引用属性或方法名称时*Python 都调用它（特殊方法名称除外，因为那样将会导致讨厌的无限循环）。
2. 如果某个类定义了 \_\_getattr\_\_() 方法，Python 将只在正常的位置查询属性时才会调用它。如果实例 *x* 定义了属性*color*，x.color 将 *不会* 调用x.\_\_getattr\_\_('color')；而只会返回*x.color*已定义好的值。
3. 无论何时给属性赋值，都会调用 \_\_setattr\_\_()方法。
4. 无论何时删除一个属性，都将调用 \_\_delattr\_\_()方法。
5. 如果定义了 \_\_getattr\_\_() 或 \_\_getattribute\_\_() 方法， \_\_dir\_\_() 方法将非常有用。通常，调用 dir(x) 将只显示正常的属性和方法。如果\_\_getattr()\_\_方法动态处理*color* 属性，dir(x) 将不会将 *color* 列为可用属性。可通过覆盖 \_\_dir\_\_() 方法允许将*color* 列为可用属性，对于想使用你的类但却不想深入其内部的人来说，该方法非常有益。

Sample：

python中\_\_get\_\_,\_\_getattr\_\_,\_\_getattribute\_\_的区别  
\_\_get\_\_,\_\_getattr\_\_和\_\_getattribute都是访问属性的方法，但不太相同。  
object.\_\_getattr\_\_(self, name)  
当一般位置找不到attribute的时候，会调用getattr，返回一个值或AttributeError异常。  
  
object.\_\_getattribute\_\_(self, name)  
无条件被调用，通过实例访问属性。如果class中定义了\_\_getattr\_\_()，则\_\_getattr\_\_()不会被调用（除非显示调用或引发AttributeError异常）  
  
object.\_\_get\_\_(self, instance, owner)  
如果class定义了它，则这个class就可以称为descriptor。owner是所有者的类，instance是访问descriptor的实例，如果不是通过实例访问，而是通过类访问的话，instance则为None。（descriptor的实例自己访问自己是不会触发\_\_get\_\_，而会触发\_\_call\_\_，只有descriptor作为其它类的属性才有意义。）（所以下文的d是作为C2的一个属性被调用）

class C(object):

a = 'abc'

def \_\_getattribute\_\_(self, \*args, \*\*kwargs):

print("\_\_getattribute\_\_() is called")

return object.\_\_getattribute\_\_(self, \*args, \*\*kwargs)

# return "haha"

def \_\_getattr\_\_(self, name):

print("\_\_getattr\_\_() is called ")

return name + " from getattr"

def \_\_get\_\_(self, instance, owner):

print("\_\_get\_\_() is called", instance, owner)

return self

def foo(self, x):

print(x)

class C2(object):

d = C()

if \_\_name\_\_ == '\_\_main\_\_':

c = C()

c2 = C2()

print(c.a)

print(c.zzzzzzzz)

c2.d

print(c2.d.a)

输出结果是：

\_\_getattribute\_\_() is called

abc

\_\_getattribute\_\_() is called

\_\_getattr\_\_() is called

zzzzzzzz from getattr

\_\_get\_\_() is called <\_\_main\_\_.C2 object at 0x16d2310> <class '\_\_main\_\_.C2'>

\_\_get\_\_() is called <\_\_main\_\_.C2 object at 0x16d2310> <class '\_\_main\_\_.C2'>

\_\_getattribute\_\_() is called

abc

小结：可以看出，每次通过**实例**访问属性，都会经过\_\_getattribute\_\_函数。而当属性不存在时，仍然需要访问\_\_getattribute\_\_，不过接着要访问\_\_getattr\_\_。这就好像是一个异常处理函数。  
每次访问descriptor（即实现了\_\_get\_\_的类），都会先经过\_\_get\_\_函数。  
  
需要注意的是，当使用类访问不存在的变量是，不会经过\_\_getattr\_\_函数。而descriptor不存在此问题，只是把instance标识为none而已。

**行为方式与函数类似的类**

可以让类的实例变得可调用——就像函数可以调用一样——通过定义 \_\_call\_\_() 方法。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 像调用函数一样“调用”一个实例 | my\_instance() | my\_instance.*\_\_call\_\_*() |

zipfile 模块 通过该方式定义了一个可以使用给定密码*解密* *经加密* *zip* 文件的类。该 zip*解密* 算法需要在解密的过程中保存状态。通过将解密器定义为类，使我们得以在 decryptor 类的单个实例中对该状态进行维护。状态在\_\_init\_\_() 方法中进行初始化，如果文件 *经加密* 则进行更新。但由于该类像函数一样“可调用”，因此可以将实例作为map() 函数的第一个参数传入，代码如下：

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

# excerpt from zipfile.py

class \_ZipDecrypter:

def \_\_init\_\_(self, pwd):

self.key0 = 305419896

self.key1 = 591751049

self.key2 = 878082192

for p in pwd:

self.\_UpdateKeys(p)

def \_\_call\_\_(self, c):

assert isinstance(c, int)

k = self.key2 | 2 c = c ^ (((k \* (k^1)) >> & 255)

self.\_UpdateKeys(c)

return c

zd = \_ZipDecrypter(pwd)

bytes = zef\_file.read(12)

h = list(map(zd, bytes[0:12]))

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

1. \_ZipDecryptor 类维护了以三个旋转密钥形式出现的状态，该状态稍后将在 \_UpdateKeys()方法中更新（此处未展示）。
2. 该类定义了一个 \_\_call\_\_() 方法，使得该类可像函数一样调用。在此例中，\_\_call\_\_()对 zip 文件的单个字节进行解密，然后基于经解密的字节对旋转密码进行更新。
3. *zd* 是 \_ZipDecryptor 类的一个实例。变量 *pwd* 被传入 \_\_init\_\_()方法，并在其中被存储和用于首次旋转密码更新。
4. 给出 zip 文件的头 12 个字节，将这些字节映射给 *zd* 进行解密，实际上这将导致调用 \_\_call\_\_() 方法 12 次，也就是 更新内部状态并返回结果字节 12 次。

**行为方式与序列类似的类**

如果类作为一系列值的容器出现——也就是说如果对某个类来说，是否“包含”某值是件有意义的事情——那么它也许应该定义下面的特殊方法已，让它的行为方式与序列类似。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 序列的长度 | *len*(seq) | seq.*\_\_len\_\_*() |
| **②** | 了解某序列是否包含特定的值 | x in seq | seq.*\_\_contains\_\_*(*x*) |

[cgi 模块](http://docs.python.org/3.1/library/cgi.html) 在其FieldStorage 类中使用了这些方法，该类用于表示提交给动态网页的所有表单字段或查询参数。

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

# A script which responds to http://example.com/search?q=cgi

import cgi fs = cgi.FieldStorage()

if 'q' in fs:

do\_search()

# An excerpt from cgi.py that explains how that works class FieldStorage: . . .

def \_\_contains\_\_(self, key):

if self.list is None:

raise TypeError('not indexable')

return any(item.name == key for item in self.list)

def \_\_len\_\_(self):

return len(self.keys())

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

一旦创建了 cgi.FieldStorage 类的实例，就可以使用 “in” 运算符来检查查询字符串中是否包含了某个特定参数。

1. 而 \_\_contains\_\_()方法是令该魔法生效的主角。
2. 如果代码为 if 'q' in fs，Python 将在 *fs* 对象中查找 \_\_contains\_\_() 方法，而该方法在cgi.py 中已经定义。'q' 的值被当作*key* 参数传入\_\_contains\_\_()方法。
3. 同样的 FieldStorage 类还支持返回其长度，因此可以编写代码 len(*fs*) 而其将调用FieldStorage 的\_\_len\_\_()方法，并返回其识别的查询参数个数。
4. self.keys() 方法检查 self.list is None 是否为真值，因此 \_\_len\_\_ 方法无需重复该错误检查。

**行为方式与字典类似的类**

在前一节的基础上稍作拓展，就不仅可以对 “in” 运算符和 len() 函数进行响应，还可像全功能字典一样根据键来返回值。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 通过键来获取值 | x[key] | x.*\_\_getitem\_\_*(*key*) |
| **②** | 通过键来设置值 | x[key] = value | x.*\_\_setitem\_\_*(*key*, *value*) |
| **③** | 删除一个键值对 | del x[key] | x.*\_\_delitem\_\_*(*key*) |
| **④** | 为缺失键提供默认值 | x[nonexistent\_key] | x.*\_\_missing\_\_*(*nonexistent\_key*) |

[cgi 模块](http://docs.python.org/3.1/library/cgi.html) 的[FieldStorage 类](http://www.4pang.com/2010/08/20/4%E8%83%96%E5%AD%A6python-%EF%BC%8D-%E7%89%B9%E6%AE%8A%E5%B1%9E%E6%80%A7%E5%92%8C%E6%96%B9%E6%B3%95%E7%9A%84%E6%80%BB%E7%BB%93.html#acts-like-list-example) 同样定义了这些特殊方法，也就是说可以像下面这样编码：

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

# A script which responds to http://example.com/search?q=cgi import cgi fs = cgi.FieldStorage()

if 'q' in fs:

do\_search(fs['q'])

# An excerpt from cgi.py that shows how it works class FieldStorage: . . .

def \_\_getitem\_\_(self, key):

if self.list is None:

raise TypeError('not indexable')

found = []

for item in self.list:

if item.name == key:

found.append(item)

if not found:

raise KeyError(key)

if len(found) == 1:

return found[0]

else: return found

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

1. *fs* 对象是 cgi.FieldStorage 类的一个实例，但仍然可以像 fs['q']这样估算表达式。
2. fs['q'] 将 *key* 参数设置为 'q' 来调用 \_\_getitem\_\_() 方法。然后它将在其内部维护的查询参数列表 (*self.list*) 中查找一个.name 与给定键相符的字典项。

**可比较的类**

我将此内容从前一节中拿出来使其单独成节，是因为“比较”操作并不局限于数字。许多数据类型都可以进行比较——字符串、列表，甚至字典。如果要创建自己的类，且对象之间的比较有意义，可以使用下面的特殊方法来实现比较。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 相等 | x == y | x.*\_\_eq\_\_*(*y*) |
| **②** | 不相等 | x != y | x.*\_\_ne\_\_*(*y*) |
| **③** | 小于 | x < y | x.*\_\_lt\_\_*(*y*) |
| **④** | 小于或等于 | x <= y | x.*\_\_le\_\_*(*y*) |
| **⑤** | 大于 | x > y | x.*\_\_gt\_\_*(*y*) |
| **⑥** | 大于或等于 | x >= y | x.*\_\_ge\_\_*(*y*) |
| **⑦** | 布尔上上下文环境中的真值 | if x: | x.*\_\_bool\_\_*() |

☞如果定义了 \_\_lt\_\_() 方法但没有定义 \_\_gt\_\_() 方法，Python 将通过经交换的算子调用\_\_lt\_\_() 方法。然而，Python 并不会组合方法。例如，如果定义了\_\_lt\_\_() 方法和 \_\_eq()\_\_ 方法，并试图测试是否 x <= y，Python 不会按顺序调用\_\_lt\_\_() 和\_\_eq()\_\_ 。它将只调用\_\_le\_\_()方法。

**可序列化的类**

Python 支持 [任意对象的序列化和反序列化](http://control.blog.sina.com.cn/admin/article/serializing.html)。（多数 Python 参考资料称该过程为 “pickling” 和 “unpickling”）。该技术对与将状态保存为文件并在稍后恢复它非常有意义。所有的[内置数据类型](http://control.blog.sina.com.cn/admin/article/native-datatypes.html) 均已支持 pickling 。如果创建了自定义类，且希望它能够 pickle，阅读[pickle 协议](http://docs.python.org/3.1/library/pickle.html) 了解下列特殊方法何时以及如何被调用。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 自定义对象的复制 | copy.copy(x) | x.*\_\_copy\_\_*() |
| **②** | 自定义对象的深度复制 | copy.deepcopy(x) | x.*\_\_deepcopy\_\_*() |
| **③** | 在 pickling 之前获取对象的状态 | pickle.dump(x, *file*) | x.*\_\_getstate\_\_*() |
| **④** | 序列化某对象 | pickle.dump(x, *file*) | x.*\_\_reduce\_\_*() |
| **⑤** | 序列化某对象（新 pickling 协议） | pickle.dump(x, *file*, *protocol\_version*) | x.*\_\_reduce\_ex\_\_*(*protocol\_version*) |
| **⑥** | 控制 unpickling 过程中对象的创建方式 | x = pickle.load(*file*) | x.*\_\_getnewargs\_\_*() |
| **⑦** | 在 unpickling 之后还原对象的状态 | x = pickle.load(*file*) | x.*\_\_setstate\_\_*() |

\* 要重建序列化对象，Python 需要创建一个和被序列化的对象看起来一样的新对象，然后设置新对象的所有属性。\_\_getnewargs\_\_() 方法控制新对象的创建过程，而\_\_setstate\_\_() 方法控制属性值的还原方式。

**可在 with 语块中使用的类**

with 语块定义了 [运行时刻上下文环境](http://www.python.org/doc/3.1/library/stdtypes.html#typecontextmanager)；在执行 with 语句时将“进入”该上下文环境，而执行该语块中的最后一条语句将“退出”该上下文环境。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 在进入 with 语块时进行一些特别操作 | with x: | x.*\_\_enter\_\_*() |
| **②** | 在退出 with 语块时进行一些特别操作 | with x: | x.*\_\_exit\_\_*() |

以下是 with *file* 习惯用法 的运作方式：

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

# excerpt from io.py:

def \_checkClosed(self, msg=None):

'''Internal: raise an ValueError if file is closed '''

if self.closed:

raise ValueError('I/O operation on closed file.' if msg is None else msg)

def \_\_enter\_\_(self):

'''Context management protocol. Returns self.'''

self.\_checkClosed()

return self

def \_\_exit\_\_(self, \*args):

'''Context management protocol. Calls close()'''

self.close()

[复制代码](http://blog.csdn.net/business122/article/details/7568446)

1. 该文件对象同时定义了一个 \_\_enter\_\_() 和一个 \_\_exit\_\_() 方法。该 \_\_enter\_\_() 方法检查文件是否处于打开状态；如果没有， \_checkClosed()方法引发一个例外。
2. \_\_enter\_\_() 方法将始终返回 *self* —— 这是 with语块将用于调用属性和方法的对象
3. 在 with 语块结束后，文件对象将自动关闭。怎么做到的？在 \_\_exit\_\_() 方法中调用了 self.close() .

☞该 \_\_exit\_\_() 方法将总是被调用，哪怕是在 with 语块中引发了例外。实际上，如果引发了例外，该例外信息将会被传递给\_\_exit\_\_() 方法。查阅[With 状态上下文环境管理器](http://www.python.org/doc/3.1/reference/datamodel.html#with-statement-context-managers) 了解更多细节。

**真正神奇的东西**

如果知道自己在干什么，你几乎可以完全控制类是如何比较的、属性如何定义，以及类的子类是何种类型。

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **目的** | **所编写代码** | **Python 实际调用** |
| **①** | 类构造器 | x = MyClass() | x.*\_\_new\_\_*() |
| **②** | 类析构器 | del x | x.*\_\_del\_\_*() |
| **③** | 只定义特定集合的某些属性 |  | x.*\_\_slots\_\_*() |
| **④** | 自定义散列值 | hash(x) | x.*\_\_hash\_\_*() |
| **⑤** | 获取某个属性的值 | x.color | type(x).*\_\_dict\_\_*['color'].\_\_get\_\_(x, type(x)) |
| **⑥** | 设置某个属性的值 | x.color = 'PapayaWhip' | type(x).*\_\_dict\_\_*['color'].\_\_set\_\_(x, 'PapayaWhip') |
| **⑦** | 删除某个属性 | del x.color | type(x).*\_\_dict\_\_*['color'].\_\_del\_\_(x) |
| **⑧** | 控制某个对象是否是该对象的实例 your class | isinstance(x, MyClass) | MyClass.*\_\_instancecheck\_\_*(x) |
| **⑨** | 控制某个类是否是该类的子类 | issubclass(C, MyClass) | MyClass.*\_\_subclasscheck\_\_*(C) |
| **⑩** | 控制某个类是否是该抽象基类的子类 | issubclass(C, MyABC) | MyABC.*\_\_subclasshook\_\_*(C) |