**Python 的 Magic Methods 指南 【已翻译100%】**

英文原文：[A Guide to Python's Magic Methods](http://www.rafekettler.com/magicmethods.html)

标签： [Python](https://www.oschina.net/translate/tag/python)

**介绍**

本指南是数月博客的总结。主题是魔术方法。

什么是魔术方法呢？它们是面向对象Python语言中的一切。它们是你可以自定义并添加“魔法”到类中的特殊方法。它们被双下划线环绕（比如\_\_init\_\_或\_\_lt\_\_）。它们的文档也不像它所需要的那么齐备。Python的所有魔术方法都在Python文档的同一区域，但它们的使用分散，组织松散。而且文档的这部分区域中几乎没有一个示例（这很有可能是设计好的，因为在语法参考里它们都很详尽，但伴随的是枯燥的语法描述等等）。  
  
因此，为了解决Python文档中我认为的缺陷，我想提供一些更简单直白的表述——示例驱动型的Python魔术方法文档。我从每周的博客开始，现在我已经完成了，并把它们合到了一起。

我希望你能喜欢它。把它作为一个教程、复习或参考使用；它希望能成为一个Python魔术方法用户友好的指导。

**构造与初始化**

我们每个知道的最基本的“魔法”方法是\_\_init\_\_。一种让我们在初始化一个类时定义一些行为。然而当我执行 x = SomeClass(), \_\_init\_\_ 不是第一个被执行的。事实上，第一被执行的的方法是\_\_new\_\_,它会创建一个实例，然后在构造器创建时传递一些参数。在一个object的生命周期的另一端的方法是\_\_del\_\_。让我们仔细看看这3个“魔法”方法：

* \_\_new\_\_(cls, [...)
* \_\_new\_\_ 是一个类的初始化过程中第一个被执行的方法。它创建了类，然后把一些参数传递给\_\_init\_\_。\_\_new\_\_ 很少被使用，特别是当我们用一些不可变类型的子类时（像tuple ,string），我不想关心\_\_new\_\_的太多的细节，因为那是没有用的。但它有它存在的意义。更多详细的请看 [in the Python docs](http://www.python.org/download/releases/2.2/descrintro/#__new__).
* \_\_init\_\_(self, [...)
* 类的构造器，当初始构造方法被执行（例如，我们执行 x = SomeClass(10,'foo')）,\_\_init\_\_ 就会获得 10 和 ‘foo’ 作为参数。\_\_init\_\_ 在python类的定义中经常被使用
* \_\_del\_\_(self)
* 若果 \_\_new\_\_ 和 \_\_init\_\_ 形成一个类的构造函数，\_\_del\_\_ 是就是析构函数。它不实现语句 del x 的行为（这样代码就不会转换为 x.\_\_del\_\_()）。它定义了一个被垃圾回收的行为。它在类消除的时后需要做一些额外的行为时是非常有用的，就像 sockets 和 file 类。注意，当编译器还在运行，如果类还存活着，这里不能确保\_\_del\_\_一定会被执行。所以\_\_del\_\_ 不能替代一些良好的编程习惯（比如连接用完了将其关掉），事实上\_\_del\_\_很少被使用，因为它的调用是非常不稳定的；请谨慎使用！

把他们合起来后，这里就是一个 \_\_init\_\_ 和 \_\_del\_\_ 使用的例子：

from os.path import joinclass FileObject:

    '''Wrapper for file objects to make sure the file gets closed on deletion.'''

    def \_\_init\_\_(self, filepath='~', filename='sample.txt'):

        # open a file filename in filepath in read and write mode

        self.file = open(join(filepath, filename), 'r+')

    def \_\_del\_\_(self):

        self.file.close()

        del self.file

**定义自己的类中的操作**

我们使用Python的“魔法”方法最大得优势之一是它提供了一种简单的方法去定义类的行为，比如 built-in 类型。这就意味着你可以避免丑陋的，违反直觉的，非标准化的基本操作方法。在一些语言中，他们通常这样写：

if instance.equals(other\_instance):

    # do something

当让Python中也可以这么做，但是这增加了混乱和不必要的冗余。不同的类库中的相同的方法可能会用不同名字，使得使用者做了太多不必要的操作。相比之下“魔法”方法是强大的，我们可以使用它定义一个方法代替上面的例子（\_\_eq\_\_ , 在这个例子中）：

if instance == other\_instance:

    #do something

这是“魔法”方法强大用途的一部分。他们绝大部分让我们定义操作的意义，以至于我们可以使用他们在我们自己的类中就像使用built in 类型。

**魔法比较方法**

python拥有大量用于实现对象与对象之间比较的魔法方法，这些对象使用运算符进行直观比较而不是难看的方法调用。同时它也提供了一种方法去重载python默认的对象比较行为（比较引用）。这里有一个这些方法和它们做了什么事情的列表：

* \_\_cmp\_\_(self, other)
* \_\_cmp\_\_ 是比较方法里面最基本的的魔法方法。实际上它实现了所有的比较运算符（如<, ==, !=）的行为，但也许不是你想要的行为（例如，一个实例是否和另一个实例相等应该由某个条件来决定，一个实例是否大于另一个实例应该由其他的条件来决定）。当self < other时\_\_cmp\_\_应该返回一个负整数，当self == other时返回0，self > other时返回正整数。通常来说最好是定义每个你需要的比较方法而不是一次性定义所有的比较方法，但是\_\_cmp\_\_是一个消除重复性的良途，并且当你有很多比较方法需要用相类似的条件去实现的时候这能让代码变得清晰。
* \_\_eq\_\_(self, other)
* 定义相等符号的行为，==
* \_\_ne\_\_(self,other)
* 定义不等符号的行为，！=
* \_\_lt\_\_(self,other)
* 定义小于符号的行为，<
* \_\_gt\_\_(self,other)
* 定义大于符号的行为，>
* \_\_le\_\_(self,other)
* 定义小于等于符号的行为，<=
* \_\_ge\_\_(self,other)
* 定义大于等于符号的行为，>=

例如，假设一个类是一个单词模型。我们可能要按字典比较单词（按字母），这是比较字符串默认行为，但我们也可能需要基于其他一些标准来做比较，比如按长度、或字节数量等。在下面的例子中，我们将比较长度。下面是实现：

class Word(str):

    '''Class for words, defining comparison based on word length.'''

    def \_\_new\_\_(cls, word):

        # Note that we have to use \_\_new\_\_. This is because str is an immutable

        # type, so we have to initialize it early (at creation)

        if ' ' in word:

            print "Value contains spaces. Truncating to first space."

            word = word[:word.index(' ')] # Word is now all chars before first space

        return str.\_\_new\_\_(cls, word)

    def \_\_gt\_\_(self, other):

        return len(self) > len(other)

    def \_\_lt\_\_(self, other):

        return len(self) < len(other)

    def \_\_ge\_\_(self, other):

        return len(self) >= len(other)

    def \_\_le\_\_(self, other):

        return len(self) <= len(other)

现在，我们可以创建两个单词(通过使用Word('foo')和Word('bar'))然后依据长度比较它们。但要注意，我们没有定义\_\_eq\_\_和\_\_ne\_\_，因为这样会导致其他一些奇怪的行为（尤其是Word('foo')==Word('bar')会判定为true），它不是基于长度相等意义上的测量，所以我们回归到字符平等意义上的实现。

现在需要留心啦——为达到预期的比较效果你不需要为每个比较定义魔术方法。如果你只定义\_\_eq\_\_以及其他的(比如\_\_gt\_\_，\_\_lt\_\_等)，标准库已经在functools模块里为我们提供了一个类修饰器，它可以定义所有的富特性比较方法。这个特性只有在Python 2.7中才是可用的，但如果你碰巧的话这可以节省大量的时间和精力。你可以通过将@total\_ordering放置在类定义前面来使用它。

**数值魔术方法**

就如同你可以通过定义比较操作来比较你自己的类实例一样，你也可以自己定义数学运算符号的行为。好吧，先系紧你的裤腰带，深呼吸......，这些操作可多着呢。由于文章组织需要，我把这些数学“魔术方法”分为5类：单目运算操作，一般数学运算操作，满足交换律的数学运算（后面会有更多介绍），参数赋值操作和类型转换操作：

单目运算符操作与函数：

单目运算符或单目运算函数只有一个操作数： 比如取负(-2），绝对值操作等。

* \_\_pos\_\_(self)
* 实现一个取正数的操作(比如 +some\_object ，python调用\_\_pos\_\_函数)
* \_\_neg\_\_(self)
* 实现一个取负数的操作(比如 -some\_object )
* \_\_abs\_\_(self)
* 实现一个内建的abs()函数的行为
* \_\_invert\_\_(self)
* 实现一个取反操作符（～操作符）的行为。想要了解这个操作的解释，参考[the Wikipedia article on bitwise operations](http://en.wikipedia.org/wiki/Bitwise_operation#NOT).
* \_\_round\_\_(self, n)
* 实现一个内建的round（）函数的行为。 n 是待取整的十进制数.
* \_\_floor\_\_(self)
* 实现math.floor()的函数行为,比如, 把数字下取整到最近的整数.
* \_\_ceil\_\_(self)
* 实现math.ceil()的函数行为,比如, 把数字上取整到最近的整数.
* \_\_trunc\_\_(self)
* 实现math.trunc()的函数行为,比如, 把数字截断而得到整数.
* **一般算数运算**

好吧，现在我们开始介绍双目运算操作或函数，比如 +, -, \* 等等. 这些很容易自解释.

* + \_\_add\_\_(self, other)
  + 实现一个加法.
  + \_\_sub\_\_(self, other)
  + 实现一个减法.
  + \_\_mul\_\_(self, other)
  + 实现一个乘法.
  + \_\_floordiv\_\_(self, other)
  + 实现一个“//”操作符产生的整除操作（）
  + \_\_div\_\_(self, other)
  + 实现一个“/”操作符代表的除法操作.
  + \_\_truediv\_\_(self, other)
  + 实现真实除法，注意，只有当你from \_\_future\_\_ import division时才会有效
  + \_\_mod\_\_(self, other)

实现一个“%”操作符代表的取模操作.

* + \_\_divmod\_\_(self, other)
  + 实现一个内建函数divmod（）
  + \_\_pow\_\_
  + 实现一个指数操作(“\*\*”操作符）的行为
  + \_\_lshift\_\_(self, other)
  + 实现一个位左移操作（<<）的功能
  + \_\_rshift\_\_(self, other)
  + 实现一个位右移操作（>>）的功能.
  + \_\_and\_\_(self, other)
  + 实现一个按位进行与操作（&）的行为.
  + \_\_or\_\_(self, other)

实现一个按位进行或操作（|）的行为.

* + \_\_xor\_\_(self, other)
  + 实现一个异或操作（^）的行为

**反射算术运算符**

你相信我说我能用一位来表示反射运算吗？可能有人会认为表示一个反射运算是大的吓人的“外国概念”，反射实际上它是非常简单的。看下面的例子：

some\_object + other

这是一个正常的加法。除了可以交换操作数以外，反射运算和加法是一样的：

other + some\_object

除了执行那种 other对像作为第一个操作数，而它自身作为第二个操作数的运算以外，所有的魔法方法做的事情与正常运算表示的意义是等价的。在大部分情况下反射运算结果和它正常的运算是等价的，所以你可以不定义\_\_radd\_\_,而是调用\_\_add\_\_等等。注意，对像（本例中的other）在运算符左边的时候，必须保证该对像没有定义（或者返回NotImplemented的）它的非反射运算符。例如，在这个例子中，some\_object.\_\_radd\_\_  只有在 other没有定义\_\_add\_\_的时候才会被调用。

* \_\_radd\_\_(self, other)
* 反射加法
* \_\_rsub\_\_(self, other)
* 反射减法的
* \_\_rmul\_\_(self, other)
* 反射除法
* \_\_rfloordiv\_\_(self, other)
* 反射地板除，使用//运算符的
* \_\_rdiv\_\_(self, other)
* 反射除法，使用/运算符的.
* \_\_rtruediv\_\_(self, other)
* 反射真除.注意只有from \_\_future\_\_ import division 的时候它才有效
* \_\_rmod\_\_(self, other)
* 反射取模运算，使用%运算符.
* \_\_rdivmod\_\_(self, other)
* 长除法，使用divmod()内置函数，当divmod(other,self)时被调用.
* \_\_rpow\_\_
* 反射乘方，使用\*\*运算符的
* \_\_rlshift\_\_(self, other)
* 反射左移，使用<<操作符.
* \_\_rrshift\_\_(self, other)
* 反射右移，使用>>操作符.
* \_\_rand\_\_(self, other)
* 反射位与，使用&操作符.
* \_\_ror\_\_(self, other)
* 反射位或，使用|操作符.
* \_\_rxor\_\_(self, other)
* 反射异或，使用^操作符.

**增量运算**

Python 还有很多种魔法方法，允许一些习惯行为被定义成增量运算。你很可能已经熟悉了增量运算，增量运算是算术运算和赋值运算的结合。如果你还不知道我在说什么，就看一下下面的例子：

x = 5x += 1 # in other words x = x + 1

每一个方法的返回值都会被赋给左边的变量。（比如，对于a += b, \_\_iadd\_\_ 可能会返回a + b, a + b会赋给变量a。） 下面是清单：

* \_\_iadd\_\_(self, other)
* 加法赋值
* \_\_isub\_\_(self, other)
* 减法赋值.
* \_\_imul\_\_(self, other)
* 乘法赋值
* \_\_ifloordiv\_\_(self, other)
* 整除赋值，地板除，相当于 //= 运算符.
* \_\_idiv\_\_(self, other)
* 除法赋值，相当于 /= 运算符.
* \_\_itruediv\_\_(self, other)
* 真除赋值，注意只有你 whenfrom \_\_future\_\_ import divisionis，才有效.
* \_\_imod\_(self, other)
* 模赋值，相当于 %= 运算符.
* \_\_ipow\_\_
* 乘方赋值，相当于 \*\*= 运算符.
* \_\_ilshift\_\_(self, other)
* 左移赋值，相当于 <<= 运算符.
* \_\_irshift\_\_(self, other)
* 左移赋值，相当于 >>= 运算符.
* \_\_iand\_\_(self, other)
* 与赋值，相当于 &= 运算符.
* \_\_ior\_\_(self, other)
* 或赋值，相当于 |= 运算符.
* \_\_ixor\_\_(self, other)
* 异或运算符，相当于 ^= 运算符.

**类型转换魔法**

Python 同样有一系列的魔法方法旨在实现内置类型的转换，比如float() 函数。它们是：

* \_\_int\_\_(self)
* 转换成整型.
* \_\_long\_\_(self)
* 转换成长整型.
* \_\_float\_\_(self)
* 转换成浮点型.
* \_\_complex\_\_(self)
* 转换成 复数型.
* \_\_oct\_\_(self)
* 转换成八进制.
* \_\_hex\_\_(self)
* 转换成十六进制.
* \_\_index\_\_(self)
* 当对象被切片时转换成int型。如果你定义了一个可能被用来做切片操作的数值型，你就应该定义\_\_index\_\_.
* \_\_trunc\_\_(self)
* 当 math.trunc(self) 使用时被调用.\_\_trunc\_\_返回自身类型的整型截取 (通常是一个长整型).
* \_\_coerce\_\_(self, other)
* 执行混合类型的运算，如果转换不能完成，应该返回None；否则，要返回一对两个元数的元组self和other, 被操作成同类型。

**表示你的类**

用一个字符串来表示一个类往往会非常有用。在Python中，有很多你可以在类定义中实施的方法来自定义内置函数的返回值以表示出你所写出的类的某些行为。

* \_\_str\_\_(self)
* 定义当 str() 被你的一个类的实例调用时所要产生的行为。
* \_\_repr\_\_(self)
* 定义 当 repr()  被你的一个类的实例调用时所要产生的行为。 str() 和 repr() 的主要区别是其目标群体。 repr() 返回的是机器可读的输出，而 str() 返回的是人类可读的。
* \_\_unicode\_\_(self)
* 定义当 unicode() 被你的一个类的实例调用时所要产生的行为。 unicode() 和 str() 很相似，但是返回的是unicode字符串。注意，如果对你的类调用 str() 然而你只定义了 \_\_unicode\_\_() ，那么其将不会工作。你应该定义 \_\_str\_\_() 来确保调用时能返回正确的值，并不是每个人都有心情去使用unicode。
* \_\_format\_\_(self, formatstr)
* 定义当你的一个类的实例被用来用新式的格式化字符串方法进行格式化时所要产生的行为。例如， "Hello, {0:abc}!".format(a) 将会导致调用 a.\_\_format\_\_("abc") 。这对定义你自己的数值或字符串类型是十分有意义的，你可能会给出一些特殊的格式化选项。
* \_\_hash\_\_(self)
* 定义当 hash()被你的一个类的实例调用时所要产生的行为。它返回一个整数，用来在字典中进行快速比较。请注意，这通常也承担着实现\_\_eq\_\_。有下面这样的规则：a == b 暗示着 hash(a) == hash(b) 。
* \_\_nonzero\_\_(self)
* 定义当 bool() 被你的一个类的实例调用时所要产生的行为。本方法应该返回True或者False，取决于你想让它返回的值。
* \_\_dir\_\_(self)
* 定义当 dir() 被你的一个类的实例调用时所要产生的行为。该方法应该返回一个属性的列表给用户，一般而言，实现 \_\_dir\_\_ 是不必要的，但是，如果你重新定义了\_\_getattr\_\_或\_\_getattribute\_\_（你将在下一节中看到）或者其它的动态生成属性，那么它对你的类的交互使用是至关重要的。
* \_\_sizeof\_\_(self)
* 定义当 sys.getsizeof() 被你的一个类的实例调用时所要产生的行为。该方法应该以字节为单位，返回你的对象的大小。这通常对于以C扩展的形式实现的Python类更加有意义，其有助于理解这些扩展。

我们几乎完成了对这些枯燥的魔法方法（并且没有实例）的指导。现在，我们已经提及到了一些较基本的魔法方法，到了该转移到更高级内容的时候了。

**属性访问控制**

很多用过其它语言的人抱怨Python缺乏对类真正的封装（比如没办法定义private属性和public的getter和settter)。但这不是真的啊：真相是Python通过“魔法”实现了大量的封装，而不是使用明确的方法或字段修饰符。看一下吧：

* \_\_getattr\_\_(self, name)
* 你可以定义如何处理用户试图访问一个不存在（不存在或还没创建）属性的行为。这对于捕获或者重定向一般的拼写错误非常有用，给出访问了不能访问的属性的警告（如果你愿意，你还可以推断并返回那个属性。），或者巧妙地处理一个AttributeError异常。它只有在一个不存在的属性被访问的情况下才被调用，然而，这并不是一个真正封装的方案。
* \_\_setattr\_\_(self, name, value)
* 与\_\_getattr\_\_不同,\_\_setattr\_\_是一个真正的封装方案。它允许你定义当给一个存在或不存在的属性赋值时的行为，意味着对任何属性值的改变你都可以定义一个规则。可是，你得小心使用\_\_setattr\_\_，在这个清单结尾的例子会向你说明。
* \_\_delattr\_\_
* 它与\_\_setattr\_\_非常像, 只不过是用来删除而不是设置属性。 \_\_detattr\_\_需要预防措施，就像setattr一样，当被调用时可能会引起无限递归（当\_\_delattr\_\_已经实现时，调用 del self.name 就会引起无限的递归)。
* \_\_getattribute\_\_(self, name)
* \_\_getattribute\_\_相当适合它的同伴\_\_setattr\_\_和\_\_delattr\_\_.但我却不建议你使用它。\_\_getattribute\_\_只有在新风格的类中才会被使用（所有的新风格类在Python最新的版本中，在老版本中，你可以子类化object来获得一个新风格类。它允许你定义一条规则来处理无论什么时候属性值被访问时的行为。比如类似于由于其它的伙伴犯错而引起的无限递归（这时你就可以调用基类的\_\_getattribute\_\_方法来阻止它）。它也避免了对\_\_getattr\_\_的依赖，当\_\_getattribute\_\_方法已经实现的时候，\_\_getattr\_\_只有在\_\_getattribute\_\_被明确的调用或抛出一个AttributeError异常的时候才会被调用。这个方法能被使用（毕竟，这是你的选择），但是我不推荐它，因为它很少使用并且运行的时候很难保证没有BUG。

如果定义了任何属性访问控制方法，容易产生错误。思考下面这个例子：

def \_\_setattr\_\_(self, name, value):

    self.name = value

    # since every time an attribute is assigned, \_\_setattr\_\_() is called, this

    # is recursion.

    # so this really means self.\_\_setattr\_\_('name', value). Since the method

    # keeps calling itself, the recursion goes on forever causing a crashdef \_\_setattr\_\_(self, name, value):

    self.\_\_dict\_\_[name] = value # assigning to the dict of names in the class

    # define custom behavior here

再次证明了Python的魔法方法是难以置信的强大，但强大的力量也需要强大的责任。如果你不想运行时中断你的代码，那了解如何适当地使用魔法方法就非常重要啦。

我们从Python中定制的属性访问中学到了什么？它们不是被轻易使用的。事实上，它有点过分强大并且违反直觉。但它们存在的原因是用来止痒的：Python不阻止你制造遭糕东西，但可能会让它变的困难。自由是最重要的东西，所以你可做任何你想做的事情。这里有一个例子，展示了一些特殊的属性访问控制行为。（注意我们使用super，因为不是所有的类都有\_\_dict\_\_属性）：

class AccessCounter(object):

    '''A class that contains a value and implements an access counter.    The counter increments each time the value is changed.'''

    def \_\_init\_\_(self, val):

        super(AccessCounter, self).\_\_setattr\_\_('counter', 0)

        super(AccessCounter, self).\_\_setattr\_\_('value', val)

    def \_\_setattr\_\_(self, name, value):

        if name == 'value':

            super(AccessCounter, self).\_\_setattr\_\_('counter', self.counter + 1)

        # Make this unconditional.

        # If you want to prevent other attributes to be set, raise AttributeError(name)

        super(AccessCounter, self).\_\_setattr\_\_(name, value)

    def \_\_delattr\_\_(self, name):

        if name == 'value':

            super(AccessCounter, self).\_\_setattr\_\_('counter', self.counter + 1)

        super(AccessCounter, self).\_\_delattr\_\_(name)]

**自定义序列**

有很多办法能让你的Python类使用起来就像内置的序列（dict,tuple,list,string等）。Python里有一些目前我最喜欢的办法，因为它们给你的控制到了荒谬的程度并且神奇地使得大量的全局函数优雅地工作在你类的实例当中。但是在深入讲这些好东西之前，我们先介绍下需求。

**需求**

在讨论在Python中创建你自己的序列也是时候谈谈协议了。在其他语言中协议有点类似于接口，因为你必须实现一系列的方法。然而，在Python中协议是完全不正式的，不需要显式的声明去实现它，它更像是一种指导原则。

为什么我们要谈论协议呢？因为在Python中实现自定义容器类型涉及到这些协议的使用。首先，有一些协议用于定义不变容器：为了实现一个不变窗口，你只需定义\_\_len\_\_和\_\_getitem\_\_方法（接下来会细说）。不变容器的协议要求所有的类加上一个 \_\_setitem\_\_和\_\_delitem\_\_方法。最后，如果你想让你的容器支持遍历，你必须定义\_\_iter\_\_方法，它返回一个iterator。这个iterator必须遵守iterator的协议，它要求iterator类里面有\_\_iter\_\_方法(返回自身)和next方法。

**容器后的魔法**

不需要再等待了，这里就是容器所使用的一些魔法方法。

* \_\_len\_\_(self)
* 返回容器的长度。对于可变和不可变容器的协议，这都是其中的一部分。
* \_\_getitem\_\_(self, key)
* 定义当某一项被访问时，使用self[key]所产生的行为。这也是不可变容器和可变容器协议的一部分。如果键的类型错误将产生TypeError；如果key没有合适的值则产生KeyError。
* \_\_setitem\_\_(self, key, value)
* 定义当一个条目被赋值时，使用self[nkey] = value所产生的行为。这也是协议的一部分。而且，在相应的情形下也会产生KeyError和TypeError。
* \_\_delitem\_\_(self, key)
* 定义当某一项被删除时所产生的行为。（例如del self[key]）。这只是可变容器协议的一部分。当你使用一个无效的键时必须抛出适当的异常。
* \_\_iter\_\_(self)
* 返回一个容器迭代器，很多情况下会返回迭代器，尤其是当内置的iter()方法被调用的时候，以及当使用for x in container:方式循环的时候。迭代器是它们本身的对象，它们必须定义返回self的\_\_iter\_\_方法。
* \_\_reversed\_\_(self)
* 实现当reversed()被调用时的行为。应该返回序列反转后的版本。仅当序列可以是有序的时候实现它，例如对于列表或者元组。
* \_\_contains\_\_(self, item)
* 定义了调用in和not in来测试成员是否存在的时候所产生的行为。你可能会问为什么这个不是序列协议的一部分？因为当\_\_contains\_\_没有被定义的时候，Python会迭代这个序列，并且当找到需要的值时会返回True。
* \_\_missing\_\_(self, key)
* 其在dict的子类中被使用。它定义了当一个不存在字典中的键被访问时所产生的行为。（例如，如果我有一个字典d，当"george"不是字典中的key时，使用了d["george"]，此时d["george"]将会被调用）。

**一个例子**

对于我们的例子, 让我们看看一个列表，它实现了一些功能结构，你可能在其他在其他程序中用到 (例如Haskell).

class FunctionalList:

    '''A class wrapping a list with some extra functional magic, like head,    tail, init, last, drop, and take.'''

    def \_\_init\_\_(self, values=None):

        if values is None:

            self.values = []

        else:

            self.values = values

    def \_\_len\_\_(self):

        return len(self.values)

    def \_\_getitem\_\_(self, key):

        # if key is of invalid type or value, the list values will raise the error

        return self.values[key]

    def \_\_setitem\_\_(self, key, value):

        self.values[key] = value

    def \_\_delitem\_\_(self, key):

        del self.values[key]

    def \_\_iter\_\_(self):

        return iter(self.values)

    def \_\_reversed\_\_(self):

        return FunctionalList(reversed(self.values))

    def append(self, value):

        self.values.append(value)

    def head(self):

        # get the first element

        return self.values[0]

    def tail(self):

        # get all elements after the first

        return self.values[1:]

    def init(self):

        # get elements up to the last

        return self.values[:-1]

    def last(self):

        # get last element

        return self.values[-1]

    def drop(self, n):

        # get all elements except first n

        return self.values[n:]

    def take(self, n):

        # get first n elements

        return self.values[:n]

这样你拥有了它，如何实现自己的序列的，有点用的例子。当然，也有更有用的应用程序的自定义序列，但在标准库中,已经有相当多的实现（包括电池，对吧？），像Counter，OrderedDict，和NamedTuple。

**反射**

你也可以控制怎么使用内置在函数sisinstance()和issubclass()方法 反射定义魔法方法. 这个魔法方法是:

* \_\_instancecheck\_\_(self, instance)
* 检查对象是否是您定义的类的一个实例(例.isinstance(instance, class).
* \_\_subclasscheck\_\_(self, subclass)
* 检查类是否是你定义类的子类 (例.issubclass(subclass, class)).

这些魔法方法的用例看起来很小, 并且确实非常实用. 我不想花太多时间在反射魔法方法上，因为它们不是非常重要, 但是它们反应了关于面向对象程序上一些重要的东西在Python上,并且总的来说Python: 总是一个简单的方法去找某些事情, 即使是没有必要的. 这些魔法方法可能看起来不是很有用, 但是一旦你需要它们，你会感到庆幸它们的存在 (并且为自己阅读了本指南高兴!).

**可调用对象**

你也许已经知道，在Python中，方法是最高级的对象。这意味着他们也可以被传递到方法中，就像其他对象一样。这是一个非常惊人的特性。

在Python中，一个特殊的魔法方法可以让类的实例的行为表现的像函数一样，你可以调用它们，将一个函数当做一个参数传到另外一个函数中等等。这是一个非常强大的特性，其让Python编程更加舒适甜美。

* \_\_call\_\_(self, [args...])
* 允许一个类的实例像函数一样被调用。实质上说，这意味着 x() 与 x.\_\_call\_\_() 是相同的。注意 \_\_call\_\_ 的参数可变。这意味着你可以定义\_\_call\_\_ 为其他你想要的函数，无论有多少个参数。

\_\_call\_\_ 在那些类的实例经常改变状态的时候会非常有效。“调用”这个实例是一种改变这个对象状态的直接和优雅的做法。比如这样一个例子，一个类表示了一个实体在飞机上的位置：

class Entity:

    '''表示一个实体的类。调用该类以更新实体的位置。'''

    def \_\_init\_\_(self, size, x, y):

        self.x, self.y = x, y

        self.size = size

    def \_\_call\_\_(self, x, y):

        '''Change the position of the entity.'''

        self.x, self.y = x, y

    # snip...

**会话管理器**

在Python 2.5中，为了代码重用而新定义了一个关键字with，其也就带来了一种with语句。会话管理在Python中并不罕见（之前是作为库的一部分而实现的），不过直到[PEP 343](http://www.python.org/dev/peps/pep-0343/)被接受后，其就作为了一种一级语言结构。你也许在之前看到过这样的语句：

with open('foo.txt') as bar:

    # 执行一些针对bar的操作

会话管理器通过包装一个with语句来设置和清理相应对象的行为。会话管理器的行为通过两个魔方方法来决定：

* \_\_enter\_\_(self)
* 定义了当使用with语句的时候，会话管理器在块被初始创建事要产生的行为。请注意，\_\_enter\_\_的返回值与with语句的目标或者as后的名字绑定。
* \_\_exit\_\_(self, exception\_type, exception\_value, traceback)
* 定义了当一个代码块被执行或者终止后，会话管理器应该做什么。它可以被用来处理异常、执行清理工作或做一些代码块执行完毕之后的日常工作。如果代码块执行成功，exception\_type，exception\_value，和traceback将会为None。否则，你可以选择处理这个异常或者是直接交给用户处理。如果你想处理这个异常的话，请确保\_\_exit\_\_在所有语句结束之后返回True。如果你想让异常被会话管理器处理的话，那么就让其产生该异常。

\_\_enter\_\_和\_\_exit\_\_对于那些定义良好以及有普通的启动和清理行为的类是很有意义的。你也可以使用这些方法来创建一般的可以包装其它对象的会话管理器。下面是一个例子：

class Closer:

    '''通过with语句和一个close方法来关闭一个对象的会话管理器。'''

    def \_\_init\_\_(self, obj):

        self.obj = obj

    def \_\_enter\_\_(self):

        return self.obj # bound to target

    def \_\_exit\_\_(self, exception\_type, exception\_val, trace):

        try:

           self.obj.close()

        except AttributeError: # obj isn't closable

           print 'Not closable.'

           return True # exception handled successfully

下面是一个实际使用Closer的例子，使用一个FTP连接来证明（一个可关闭的套接字）：

>>> from magicmethods import Closer

>>> from ftplib import FTP

>>> with Closer(FTP('ftp.somesite.com')) as conn:

...     conn.dir()

...

>>> conn.dir()

>>> with Closer(int(5)) as i:

...     i += 1

...

Not closable.

>>> i

6

看到我们的包装器如何友好地处理恰当和不不恰当的行为了吗？这是会话管理器和魔法方法的强大功能。请注意，Python标准库包括了一个叫作 [contextlib](http://docs.python.org/library/contextlib.html) 的模块，其包含了一个会话管理器，contextlib.closing()完成了类似的功能（当一个对象没有close()方法时则没有任何处理）。

**抽象基类**

见http://docs.python.org/2/library/abc.html。

**创建描述器对象**

描述器是通过获取、设置以及删除的时候被访问的类。当然也可以改变其它的对象。描述器并不是独立的。相反，它意味着被一个所有者类持有。当创建面向对象的数据库或者类，里面含有相互依赖的属相时，描述器将会非常有用。一种典型的使用方法是用不同的单位表示同一个数值，或者表示某个数据的附加属性（比如坐标系上某个点包含了这个点到原点的距离信息）。

为了成为一个描述器，一个类必须至少有\_\_get\_\_，\_\_set\_\_，\_\_delete\_\_方法被实现，让我们看看这些魔法方法：

* \_\_get\_\_(self, instance, owner)
* 定义了当描述器的值被取得的时候的行为。instance是拥有该描述器对象的一个实例。owner是拥有者本身。
* \_\_set\_\_(self, instance, value)
* 定义了当描述器的值被改变的时候的行为。instance是拥有该描述器类的一个实例。value是要设置的值。
* \_\_delete\_\_(self, instance)
* 定义了当描述器的值被删除的时候的行为。instance是拥有该描述器对象的一个实例。

下面是一个描述器的实例：单位转换。

class Meter(object):

    '''对于”米“的描述器。'''

    def \_\_init\_\_(self, value=0.0):

        self.value = float(value)

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

        return self.value

    def \_\_set\_\_(self, instance, value):

        self.value = float(value)class Foot(object):

    '''对于”英尺“的描述器。'''

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

        return instance.meter \* 3.2808

    def \_\_set\_\_(self, instance, value):

        instance.meter = float(value) / 3.2808class Distance(object):

    '''用米和英寸来表示两个描述器之间的距离。'''

    meter = Meter()

    foot = Foot()

**复制**

有时候，尤其是当你在处理可变对象时，你可能想要复制一个对象，然后对其做出一些改变而不希望影响原来的对象。这就是Python的copy所发挥作用的地方。然而（幸运的是），Python的模块并不是“感性”的，所以我们没必要担心一个基于Linux的机器会突然开始工作，但是我们确实需要告诉Python如何高效地复制一些东西。

* \_\_copy\_\_(self)
* 定义了当对你的类的实例调用copy.copy()时所产生的行为。copy.copy()返回了你的对象的一个*浅拷贝*——这意味着，当实例本身是一个新实例时，它的所有数据都被引用了——例如，当一个对象本身被复制了，它的数据仍然是被引用的（因此，对于浅拷贝中数据的更改仍然可能导致数据在原始对象的中的改变）。
* \_\_deepcopy\_\_(self, memodict={})
* 定义了当对你的类的实例调用copy.deepcopy()时所产生的行为。copy.deepcopy()返回了你的对象的一个*深拷贝——*对象和其数据都被拷贝了。memodict是对之前被拷贝的对象的一个缓存——这优化了拷贝过程并且阻止了对递归数据结构拷贝时的无限递归。当你想要进行对一个单独的属性进行深拷贝时，调用copy.deepcopy()，并以memodict为第一个参数。

这些魔法方法的使用例子都是什么？答案和以往一样，当你需要进行和默认行为相比，更细粒度的控制时使用这些方法。例如，你想要复制一个对象，其中以字典的形式（其可能会很大）存储了一个缓存，那么对缓存进行复制可能是没有意义的——如果当该缓存可以在内存中被多个实例共享，那么对其进行复制就确实是没意义的。

**Pickling 序列化你的对象**

如果你打算与其他python发烧友交换数据，那你一定应该听说过pickling。Pickling是一个将Python数据结构进行序列化的工具，它对于存储、重新取回一个对象这类工作来说真是难以置信的有用。但它也是一些担心和误解的源头。

Pickling是如此的重要，以至于它不仅仅拥有自己的模块（pickling），而且还有自己的协议和“魔术”方法。但首先，我们先简单地介绍一下Pickling如何序列化已存在的类型（如果你已经知道这些了，那么请自行飘过）。

**Pickling: 赶快到盐水中泡泡**

(译者注：pickle是用来直接保存Python对象的模块，在英文中有“腌制”的意思)

让我们深入挖掘pickling方法。假设你想保存一个字典并在之后检索它：你可以把它写入一个文件中，小心确保其有正确的语法，之后用exec()或者读取文件来检索它。但这很有可能是相当危险的：如果你将重要数据保存在纯文本中，它可能会损坏或者发生各种各样的改变，有些会让你的程序崩溃，有些甚至会在你的电脑上运行恶意代码。因此，我们应该使用 pickle方法：

import pickle

data = {'foo': [1, 2, 3],

        'bar': ('Hello', 'world!'),

        'baz': True}

jar = open('data.pkl', 'wb')

pickle.dump(data, jar) # write the pickled data to the file jar

jar.close()

几个小时之后，我们希望找回这些数据，现在我们只需unpickle它：

import pickle

pkl\_file = open('data.pkl', 'rb') # connect to the pickled data

data = pickle.load(pkl\_file) # load it into a variable

print data

pkl\_file.close()

发生了什么？正如你所想的那样，我们现在找回了data。

现在，我们要注意一点：pickle并不完美。被pickle序列化的文件很容易被意外或是有意损坏。pickle模块可能比一般的纯文本文件要来的安全，但它仍然可能会被利用去运行恶意代码。而且它在各个Python版本之间是不兼容的，所以不要传送pkl文件并妄想其他人可以打开它。但是，pickle确实是处理缓存和其他序列化任务的强有力工具。

**用Pickle序列化你的对象**

pickle模块不仅可以用于内建类型，它还可以以用于序列化任何遵循pickle协议的类。pickle协议为Python对象定义了四个可选的方法，你可以重载这些方法来定义它们的行为（这和C扩展有些不同，但这不在我们的讨论范围之内）：

* \_\_getinitargs\_\_(self)
* 如果你想在你的类被unpickle的时候执行\_\_init\_\_方法，你可以重载\_\_getinitargs\_\_方法，它会返回一个元组，包含你想传给\_\_init\_\_方法的参数。注意，这种方法只适用于旧式的Python类型（译者注：区别于2.2中引入的新式类）。
* \_\_getnewargs\_\_(self)
* 对于新式类，在unpickle的时候你可以决定传给\_\_new\_\_方法的参数。以上方法可以返回一个包含你想传给\_\_new\_\_方法的参数元组。
* \_\_getstate\_\_(self)
* 除了储存\_\_dict\_\_中的原来的那些变量，你可以自定义使用pickle序列化对象时想要储存的额外属性。这些属性将在你unpickle文件时被\_\_setstate\_\_方法使用。
* \_\_setstate\_\_(self, state)
* 当文件被unpickle时，其中保存的对象属性不会直接被写入对象的\_\_dict中，而是会被传入这个方法。这个方法和\_\_getstate\_\_是配套的：当他们都被定义了的时候，你可以任意定义对象被序列化存储时的状态。
* \_\_reduce\_\_(self)
* 当你定义扩展类（使用C语言实现的Python扩展类）时，可以通过实现\_\_reduce\_\_函数来控制pickle的数据。如果\_\_reduce\_\_()方法被定义了，在一个对象被pickle时它将被调用。如果它返回一个字符串，那么pickle在将在全局空间中搜索对应名字的对象进行pickle；它还可以返回一个元组，包含2-5个元素： 一个可以用来重建该对象的可调用对象，一个包含有传给该可调用对象参数的元组，传给\_\_setstate\_\_方法的参数（可选），一个用于待pickle对象列表的迭代器（译者注：这些对象会被append到原来对象的后面）（可选）调用对象，一个包含有传给该可调用对象参数的元组，传给\_\_setstate\_\_方法的参数（可选），一个用于待pickle对象列表的迭代器（译者注：这些对象会被append到原来对象的后面）（可选），一个用于待pickle的字典的迭代器（可选）。
* \_\_reduce\_ex\_\_(self)
* \_\_reduce\_ex\_\_是为兼容性而设计的。如果它被实现了，\_\_reduce\_ex\_\_将会取代\_\_reduce\_\_在pickle时被执行。\_\_reduce\_\_可以同时被实现以支持那些不支持\_\_reduce\_ex\_\_的老版本pickling API。

（译者注：这段说的不是非常清楚，感兴趣可以去看文档，一般来说只要使用上一节中的方法就足够了，注意在反序列化之前要先有对象的定义，否则会出错）

**一个例子**

我们以Slate为例，这一段记录一个值以及这个值是何时被写入的程序，但是，这个Slate有一点特殊的地方，就是当前值不会被保存。

import time

class Slate:

    '''Class to store a string and a changelog, and forget its value when    pickled.'''

    def \_\_init\_\_(self, value):

        self.value = value

        self.last\_change = time.asctime()

        self.history = {}

    def change(self, new\_value):

        # Change the value. Commit last value to history

        self.history[self.last\_change] = self.value

        self.value = new\_value

        self.last\_change = time.asctime()

    def print\_changes(self):

        print 'Changelog for Slate object:'

        for k, v in self.history.items():

            print '%s\t %s' % (k, v)

    def \_\_getstate\_\_(self):

        # Deliberately do not return self.value or self.last\_change.

        # We want to have a "blank slate" when we unpickle.

        return self.history

    def \_\_setstate\_\_(self, state):

        # Make self.history = state and last\_change and value undefined

        self.history = state

        self.value, self.last\_change = None, None

**总结**

这份指南的目的是希望为所有人带来一些知识，即使你是Python大牛或者精通面向对象开发。如果你是一个Python初学者，阅读这篇文章之后，你已经获得了编写丰富，优雅，灵活的类的知识基础了。如果你是一个有一些经验的Python程序员，你可能会发现一些能让你写的代码更简洁的方法。如果你是一个曾经使用过Python的程序员，该文可能会帮助你知晓一些新的概念和方法以及帮助你减少编写代码量的方式。如果你是一个Python专家，该文会帮助你想起来一些你已经遗忘的只是，或者一些你还没听说过的新功能。不惯你现在有多少经验，我希望这次对于Python特殊方法的旅程是真正的一次神奇之旅。（双关语的感觉真是棒！）

**附录 1: 如何调用Magic Method**

一些magic method已经映射到自带的方法(built-in functions)；这种情况下如何调用他们是显而易见的。然而，在其他情况下，调用它们就不那么容易了。本附录致力于展示能够调用magic method的一些不被察觉的语法。

|  |  |  |
| --- | --- | --- |
| **Magic Method** | **何时被调用(例子)** | **Explanation** |

|  |  |  |
| --- | --- | --- |
| \_\_new\_\_(cls [,...]) | instance = MyClass(arg1, arg2) | \_\_new\_\_ is called on instance creation |
| \_\_init\_\_(self [,...]) | instance = MyClass(arg1, arg2) | \_\_init\_\_ is called on instance creation |
| \_\_cmp\_\_(self, other) | self == other, self > other, etc. | Called for any comparison |
| \_\_pos\_\_(self) | +self | Unary plus sign |
| \_\_neg\_\_(self) | -self | Unary minus sign |
| \_\_invert\_\_(self) | ~self | Bitwise inversion |
| \_\_index\_\_(self) | x[self] | Conversion when object is used as index |
| \_\_nonzero\_\_(self) | bool(self) | Boolean value of the object |
| \_\_getattr\_\_(self, name) | self.name # name doesn't exist | Accessing nonexistent attribute |
| \_\_setattr\_\_(self, name, val) | self.name = val | Assigning to an attribute |
| \_\_delattr\_\_(self, name) | del self.name | Deleting an attribute |
| \_\_getattribute\_\_(self, name) | self.name | Accessing any attribute |
| \_\_getitem\_\_(self, key) | self[key] | Accessing an item using an index |
| \_\_setitem\_\_(self, key, val) | self[key] = val | Assigning to an item using an index |
| \_\_delitem\_\_(self, key) | del self[key] | Deleting an item using an index |
| \_\_iter\_\_(self) | for x in self | Iteration |
| \_\_contains\_\_(self, value) | value in self,value not in self | Membership tests using in |
| \_\_call\_\_(self [,...]) | self(args) | "Calling" an instance |
| \_\_enter\_\_(self) | with self as x: | with statement context managers |
| \_\_exit\_\_(self, exc, val, trace) | with self as x: | with statement context managers |
| \_\_getstate\_\_(self) | pickle.dump(pkl\_file, self) | Pickling |
| \_\_setstate\_\_(self) | data = pickle.load(pkl\_file) | Pickling |

希望这个表能够解决你可能会遇到的哪个语法调用哪个magic method的问题。

[**附录 2: Python 3中的改动**](http://www.rafekettler.com/magicmethods.html?utm_campaign=Manong_Weekly_Issue_14&utm_medium=EDM&utm_source=Manong_Weekly#appendix2)

这里我们列举出一些Python 3与2.x在对象模型上主要的的不同之处。

* 因为Python 3中string和unicode直接已经没有差别，\_\_unicode\_\_已经不存在了，并且\_\_bytes\_\_(它的行为与\_\_str\_\_和\_\_unicode\_\_类似)成为新的自带方法来构造byte数组。
* 因为Python 3里面的division默认变成了true division，\_\_div\_\_在Python3中不存在了。
* \_\_coerce\_\_被去除掉了是因为它与其他magic method冗余并且造成了行为混淆。
* \_\_cmp\_\_被去除掉了是因为它与其他magic method冗余。
* \_\_nonzero\_\_被重命名为\_\_bool\_\_