

# Python 魔术方法指南

- 入门
- 构造和初始化
- 构造定制类
  - 用于比较的魔术方法
  - 用于数值处理的魔术方法
- 表现你的类
- 控制属性访问
- 创建定制序列
- 反射
- 可以调用的对象
- 会话管理器
- 创建描述器对象
- 持久化对象
- 总结
- 附录

## 介绍

此教程为我的数篇文章中的一个重点。主题是魔术方法。什么是魔术方法?他们是面向对象的Python的一切。他们是可以给你的类增加”magic”的特殊方法。他们总是被双下划线所包围(e.g. `__init__` 或者 `__lt__`)。然而他们的文档却远没有提供应该有的内容。Python中所有的魔术方法均在Python官方文档中有相应描述,但是对于他们的描述比较混乱而且组织比较松散。很难找到一个例子(也许他们原本打算的很好,在开始语言参考中有描述很详细,然而随之而来的确是枯燥的语法描述等等)。

所以,为了修补我认为Python文档应该修补的瑕疵,我决定给Python中的魔术方法提供一些用平淡的语言和实例驱使的文档。我在开始已经写了数篇博文,现在在这篇文章中对他们进行总结。

我希望你能够喜欢这篇文章。你可以将之当做一个教程,一个补习资料,或者一个参考。本文的目的仅仅是为Python中的魔术方法提供一个友好的教程。

## 构造和初始化

每个人都知道一个最基本的魔术方法, `__init__` 。通过此方法我们可以定义一个对象的初始操作。然而,当我调用 `x = SomeClass()` 的时候, `__init__` 并不是第一个被调用的方法。实际上,还有一个叫做 `__new__` 的方法,来构造这个实例。然后给在开始创建时候的初始化函数

 v: latest ▼

数来传递参数。在对象生命周期的另一端，也有一个 `__del__` 方法。我们现在来近距离的看一看这三个方法：

`__new__(cls, [...])` `__new__` 是在一个对象实例化的时候所调用的第一个方法。它的第一个参数是这个类，其他的参数是用来直接传递给 `__init__` 方法。`__new__` 方法相当不常用，但是它有自己的特性，特别是当继承一个不可变的类型比如一个tuple或者string。我不希望在`__new__` 上有太多细节，因为并不是很有用处，但是在 [Python文档](#) 中有详细的阐述。

`__init__(self, [...])` 此方法为类的初始化方法。当构造函数被调用的时候的任何参数都将会传给它。（比如如果我们调用 `x = SomeClass(10, 'foo')`），那么 `__init__` 将会得到两个参数10和foo。`__init__` 在Python的类定义中被广泛用到。

`__del__(self)` 如果 `__new__` 和 `__init__` 是对象的构造器的话，那么 `__del__` 就是析构器。它不实现语句 `del x`（以上代码将不会翻译为 `x.__del__()`）。它定义的是当一个对象进行垃圾回收时候的行为。当一个对象在删除的时需要更多的清洁工作的时候此方法会很有用，比如套接字对象或者是文件对象。注意，如果解释器退出的时候对象还存存在，就不能保证`__del__` 能够被执行，所以 `__del__` can't serve as a replacement for good coding practices ()~~~~~

放在一起的话，这里是一个 `__init__` 和 `__del__` 实际使用的例子。

---

```
from os.path import join

class FileObject:
    '''给文件对象进行包装从而确认在删除时文件流关闭'''

    def __init__(self, filepath='~', filename='sample.txt'):
        #读写模式打开一个文件
        self.file = open(join(filepath, filename), 'r+')

    def __del__(self):
        self.file.close()
        del self.file
```

---


## 让定制类工作起来

使用Python的魔术方法的最大优势在于他们提供了一种简单的方法来让对象可以表现的像内置类型一样。那意味着你可以避免丑陋的，违反直觉的，不标准的操作方法。在一些语言中，有一些操作很常用比如：

---

```
if instance.equals(other_instance):
    # do something
```

---

在Python中你可以这样。但是这会让人迷惑且产生不必要的冗余。相同的操作因为  [v: latest](#) 会使用不同的名字，这样会产生不必要的工作。然而有了魔术方法的力量，我们可以定义一个方法（本例中为 `__eq__`），就说明了我们的意思：

```
if instance == other_instance:
    #do something
```

这只是魔术方法的功能的一小部分。它让你可以定义符号的含义所以我们可以我们的类中使用。就像内置类型一样。

## 用于比较的魔术方法

Python对实现对象的比较，使用魔术方法进行了大的逆转，使他们非常直观而不是笨拙的方法调用。而且还提供了一种方法可以重写Python对对象比较的默认行为(通过引用)。以下是这些方法和他们的作用。

`__cmp__(self, other)` `__cmp__` 是最基本的用于比较的魔术方法。它实际上实现了所有的比较符号(<,<=,>,>=,!=,etc.)，但是它的表现并不会总是如你所愿(比如，当一个实例与另一个实例相等是通过一个规则来判断，而一个实例大于另外一个实例是通过另外一个规则来判断)。如果 `self < other` 的话 `__cmp__` 应该返回一个负数，当 `self == other` 的时候会返回0，而当 `self > other` 的时候会返回正数。通常最好的一种方式是去分别定义每一个比较符号而不是一次性将他们定义。但是 `__cmp__` 方法是你想要实现所有的比较符号而一个保持清楚明白的一个好的方法。

`__eq__(self, other)` 定义了等号的行为，`=`。

`__ne__(self, other)` 定义了不等号的行为，`≠`。

`__lt__(self, other)` 定义了小于号的行为，`<`。

`__gt__(self, other)` 定义了大于等于号的行为，`≥`。

举一个例子，创建一个类来表现一个词语。我们也许会想要比较单词的字典序(通过字母表)，通过默认的字符串比较的方法就可以实现，但是我们也想要通过一些其他的标准来实现，比如单词长度或者音节数量。在这个例子中，我们来比较长度实现。以下是实现代码：

```
class Word(str):
    '''存储单词的类，定义比较单词的几种方法'''

    def __new__(cls, word):
        # 注意我们必须要用到__new__方法，因为str是不可变类型
        # 所以我們必須在創建的時候將它初始化
        if ' ' in word:
            print "Value contains spaces. Truncating to first space."
            word = word[:word.index(' ')] #单词是第一个空格之前的所有字符
        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)
```

 v: latest ▾

```
def __le__(self, other):
    return len(self) ≤ len(other)
```

现在，我们创建两个 Words 对象(通过使用 Word('foo') 和 Word('bar')) 然后通过长度来比较它们。注意，我们没有定义 `__eq__` 和 `__ne__` 方法。这是因为将会产生一些怪异的结果(比如 Word('foo') = Word('bar') 将会返回true)。这对于测试基于长度的比较不是很有意义。所以我们退回去，用 `str` 内置来进行比较。

现在你知道你不必定义每一个比较的魔术方法从而进行丰富的比较。标准库中很友好的在 `functiontools` 中提供给我们一个类的装饰器定义了所有的丰富的比较函数。如果你只是定义 `__eq__` 和另外一个(e.g. `__gt__`, `__lt__`, etc.) 这个特性仅仅在Python 2.7中存在，但是你如果有机会碰到的话，那么将会节省大量的时间和工作量。你可以通过在你定义类前放置 `@total_ordering` 来使用。

## 数值处理的魔术方法

如同你在通过比较符来比较类的实例的时候来创建很多方法，你也可以定义一些数值符号的特性。系紧你的安全带，来吧，这里有很多内容。为了组织方便，我将会把数值处理的方法来分成五类：一元操作符，普通算数操作符，反射算数操作符(之后会详细说明)，增量赋值，和类型转换。

### 一元操作符和函数

仅仅有一个操作位的一元操作符和函数。比如绝对值，负等。

`__pos__(self)` 实现正号的特性(比如 `+some_object`)

`__neg__(self)` 实现负号的特性(比如 `-some_object`)

`__abs__(self)` 实现内置 `abs()` 函数的特性。

`__invert__(self)` 实现 `~` 符号的特性。为了说明这个特性。你可以查看 [Wikipedia中的这篇文章](#)

### 普通算数操作符

现在我们仅仅覆盖了普通的二进制操作符：`+`, `-`, `*`和类似符号。这些符号大部分来说都浅显易懂。

`__add__(self, other)` 实现加法。 `__sub__(self, other)` 实现减法。 `__mul__(self, other)` 实现乘法。 `__floordiv__(self, other)` 实现 `//` 符号实现的整数除法。 `__div__(self, other)` 实现 `/` 符号实现的除法。 `__truediv__(self, other)` 实现真除法。注意只有只用 `from __future__ import division` 的时候才会起作用。 `__mod__(self, other)` 实现取模算法 `%` `__divmod__(self, other)` 实现内置 `divmod()` 算法 `__pow__` 实现使用 `**` 的指数运算

`__lshift__(self, other)` 实现使用 `<<` 的按位左移动 `__rshift__(self, other)` 实现使用 `>>` 的按位右移动 `__and__(self, other)` 实现使用 `&` 的按位与 `__or__(self, other)` 实现使用 `|` 的按位或 `__xor__(self, other)` 实现使用 `^` 的按位异或

## 反运算

下面我将会讲解一些反运算的知识。有些概念你可能会认为恐慌或者是陌生。但是实际上非常简单。以下是一个例子：

---

```
some_object + other
```

---

这是一个普通的加法运算，反运算是相同的，只是把操作数调换了位置：

---

```
other + some_object
```

---

所以，除了当与其他对象操作的时候自己会成为第二个操作数之外，所有的这些魔术方法都与普通的操作是相同的。大多数情况下，反运算的结果是与普通运算相同的。所以你可以将 `__radd__` 与 `__add__` 等价。

`__radd__(self, other)` 实现反加 `__rsub__(self, other)` 实现反减 `__rmul__(self, other)` 实现反乘 `__rfloordiv__(self, other)` 实现 `//` 符号的反除 `__rdiv__(self, other)` 实现 `/` 符号的反除 `__rtruediv__(self, other)` 实现反真除，只有当 `from __future__ import division` 的时候会起作用 `__rmod__(self, other)` 实现 `%` 符号的反取模运算 `__rdivmod__(self, other)` 当 `divmod(other, self)` 被调用时，实现内置 `divmod()` 的反运算 `__rpow__` 实现 `**` 符号的反运算 `__rlshift__(self, other)` 实现 `<<` 符号的反左位移 `__rrshift__(self, other)` 实现 `>>` 符号的反右位移 `__rand__(self, other)` 实现 `&` 符号的反与运算 `__ror__(self, other)` 实现 `|` 符号的反或运算 `__xor__(self, other)` 实现 `^` 符号的反异或运算


## 增量赋值

Python也有大量的魔术方法可以来定制增量赋值语句。你也许对增量赋值已经很熟悉，它将操作符与赋值结合起来。如果你仍然不清楚我在说什么的话，这里有一个例子：

---

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

---

`__iadd__(self, other)` 实现赋值加法 `__isub__(self, other)` 实现赋值减法 `__imul__(self, other)` 实现赋值乘法 `__ifloordiv__(self, other)` 实现 `//=` 的赋值地板除 `__idiv__(self, other)` 实现符号 `/=` 的赋值除 `__itruediv__(self, other)` 实现赋值真除，只有使用  [v: latest](#) `__future__ import division` 的时候才能使用 `__imod__(self, other)` 实现符号 `%=` 的赋值取模 `__ipow__` 实现符号 `**=` 的赋值幂运算 `__ilshift__(self, other)` 实现符号 `<=<` 的赋值位左移



`__irshift__(self, other)` 实现符号 `>>=` 的赋值位右移 `__iand__(self, other)` 实现符号 `&=` 的赋值位与 `__ior__(self, other)` 实现符号 `|=` 的赋值位或 `__ixor__(self, other)` 实现符号 `^=` 的赋值位异或

## 类型转换魔术方法

Python也有很多的魔术方法来实现类似 `float()` 的内置类型转换特性。 `__int__(self)` 实现整形的强制转换 `__long__(self)` 实现长整形的强制转换 `__float__(self)` 实现浮点型的强制转换 `__complex__(self)` 实现复数的强制转换 `__oct__(self)` 实现八进制的强制转换 `__hex__(self)` 实现二进制的强制转换 `__index__(self)` 当对象是被应用在切片表达式中时, 实现整形强制转换, 如果你定义了一个可能在切片时用到的定制的数值型, 你应该定义 `__index__` (详见PEP357) `__trunc__(self)` 当使用 `math.trunc(self)` 的时候被调用。 `__trunc__` 应该返回数值被截取成整形(通常为长整形)的值 `__coerce__(self, other)` 实现混合模式算数。如果类型转换不可能的话, 那么 `__coerce__` 将会返回 `None` , 否则他将对 `self` 和 `other` 返回一个长度为2的tuple, 两个为相同的类型。

## 表现你的类

如果有一个字符串来表示一个类将会非常有用。在Python中, 有很多方法可以实现类定义内置的一些函数的返回值。 `__str__(self)` 定义当 `str()` 调用的时候的返回值 `__repr__(self)` 定义 `repr()` 被调用的时候的返回值。 `str()` 和 `repr()` 的主要区别在于 `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()` 调用的时候的返回值。 `unicode()` 和 `str()` 很相似, 但是返回的是unicode字符串。注意, 如a果对你的类调用 `str()` 然而你只定义了 `__unicode__()` , 那么将不会工作。你应该定义 `__str__()` 来确保调用时能返回正确的值。

`__hash__(self)` 定义当 `hash()` 调用的时候的返回值, 它返回一个整形, 用来在字典中进行快速比较 `__nonzero__(self)` 定义当 `bool()` 调用的时候的返回值。本方法应该返回True或者False, 取决于你想让它返回的值。

## 控制属性访问

许多从其他语言转到Python的人会抱怨它缺乏类的真正封装。(没有办法定义私有变量, 然后定义公共的getter和setter)。Python其实可以通过魔术方法来完成封装。我们来看一下:

`__getattr__(self, name)` 你可以定义当用户试图获取一个不存在的属性时的行为。这适用于对普通拼写错误的获取和重定向, 对获取一些不建议的属性时候给出警告(如果你愿意你也可以计算并且给出一个值)或者处理一个 `AttributeError` 。只有当调用不存在的属性的时候会被返回。然而, 这不是一个封装的解决方案。 `__setattr__(self, name, value)` 与 `__getattr__` 不同, `__setattr__` 是一个封装的解决方案。无论属性是否存在, 它都允许你定义对 `v: latest` 值行为, 以为这你可以对属性的值进行个性定制。但是你必须对使用 `__setattr__` 特别小心。

之后我们会详细阐述。 `__delattr__` 与 `__setattr__` 相同，但是功能是删除一个属性而不是设置他们。注意与 `__setattr__` 相同，防止无限递归现象发生。（在实现 `__delattr__` 的时候调用 `del self.name` 即会发生） `__getattribute__(self, name)` `__getattribute__` 与它的同伴 `__setattr__` 和 `__delattr__` 配合非常好。但是我不建议使用它。只有在新类型类定义中才能使用 `__getattribute__`（在最新版本Python中所有的类都是新类型，在老版本中你可以通过继承 `object` 来制作一个新类。这样你可以定义一个属性值的访问规则。有时也会产生一些递归现象。（这时候你可以调用基类的 `__getattribute__` 方法来防止此现象的发生。）它可以消除对 `__getattr__` 的使用，如果它被明确调用或者一个 `AttributeError` 被抛出，那么当实现 `__getattribute__` 之后才能被调用。此方法是否被使用其实最终取决于你的选择。）我不建议使用它因为它的使用几率较小（我们在取得一个值而不是设置一个值的时候有特殊的行为是非常罕见的。）而且它不能避免会出现bug。

在进行属性访问控制定义的时候你可能会很容易的引起一个错误。考虑下面的例子。

---

```
def __setattr__(self, name, value):
    self.name = value
    #每当属性被赋值的时候， ``__setattr__()`` 会被调用，这样就造成了递归调用。
    #这意味这会调用 ``self.__setattr__('name', value)``，每次方法会调用自己。这样会造成程

def __setattr__(self, name, value):
    self.__dict__[name] = value #给类中的属性名分配值
    #定制特有属性
```

---

Python的魔术方法非常强大，然而随之而来的则是责任。了解正确的方法去使用非常重要。

所以我们对于定制属性访问权限了解了多少呢。它不应该被轻易的使用。实际上，它非常大。但是它存在的原因是:Python 不会试图将一些不好的东西变得不可能，而是让它们难以实现。自由是至高无上的，所以你可以做任何你想做的。以下是一个特别的属性控制的例子（我们使用 `super` 因为不是所有的类都有 `__dict__` 属性）：

---

```
class AccessCounter:
    '''一个包含计数器的控制权限的类每当值被改变时计数器会加一'''

    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)
        #如果你不想让其他属性被访问的话，那么可以抛出 AttributeError(name) 异常
        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)]
```

---

 v: latest ▾

## 创建定制的序列

有很多方法让你的Python类行为可以像内置的序列(dict, tuple, list, string等等)。这是目前为止我最喜欢的魔术方法,因为它给你很搞的控制权限而且让很多函数在你的类实例上工作的很出色。但是在开始之前,需要先讲一些必须条件。

### 必须条件

现在我们开始讲如何在Python中创建定制的序列,这个时候该讲一讲协议。协议(Protocols)与其他语言中的接口很相似。它给你很多你必须定义的方法。然而在Python中的协议是很不正式的,不需要明确声明实现。事实上,他们更像一种指南。

我们为什么现在讨论协议?因为如果要定制容器类型的话需要用到这些协议。首先,实现不变容器的话有一个协议:实现不可变容器,你只能定义 `__len__` 和 `__getitem__` (一会会讲更多)。可变容器协议则需要所有不可变容器的所有另外还需要 `__setitem__` 和 `__delitem__`。最终,如果你希望你的对象是可迭代的话,你需要定义 `__iter__` 会返回一个迭代器。迭代器必须遵循迭代器协议,需要有 `__iter__` (返回它本身) 和 `next`。

### 容器后的魔法

这些是容器使用的魔术方法。`__len__(self)` 返回容器长度。对于可变不可变容器都需要有的协议的一部分。`__getitem__(self, key)` 定义当一个条目被访问时,使用符号 `self[key]`。这也是不可变容器和可变容器都要有的协议的一部分。如果键的类型错误和 `KeyError` 或者没有合适的值。那么应该抛出适当的 `TypeError` 异常。`__setitem__(self, key, value)` 定义当一个条目被赋值时的行为,使用 `self[key] = value`。这也是可变容器和不可变容器协议中都要有的一部分。`__delitem__(self, key)` 定义当一个条目被删除时的行为(比如 `del self[key]`)。这只是可变容器协议中的一部分。当使用一个无效的键时应该抛出适当的异常。`__iter__(self)` 返回一个容器的迭代器。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,或者当使用 `for x in container` 方式循环的时候。迭代器是他们本身的对象,他们必须定义返回 `self` 的 `__iter__` 方法。`__reversed__(self)` 实现当 `reversed()` 被调用时的行为。应该返回列表的反转版本。`__contains__(self, item)` 当调用 `in` 和 `not in` 来测试成员是否存在时候 `__contains__` 被定义。你问为什么这个不是序列协议的一部分?那是因为当 `__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`。`__concat__(self, other)` 最终,你可以通过 `__concat__` 来定义当用其他的来连接两个序列时候的行为。当 `+` 操作符被调用时候会返回一个 `self` 和 `other.__concat__` 被调用后的结果产生的新序列。

### 一个例子

在我们的例子中,让我们看一看你可能在其他语言中 用到的函数构造语句的实现 (Haskell)。

 v: latest ▾



```

class Functionallist:
    '''一个封装了一些附加魔术方法比如 head, tail, init, last, drop, 和take的列表类。'''
    ...

    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):
        #如果键的类型或者值无效, 列表值将会抛出错误
        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 reversed(self.values)

    def append(self, value):
        self.values.append(value)

    def head(self):
        return self.values[0]

    def tail(self):
        return self.values[1:]

    def init(self):
        #返回一直到末尾的所有元素
        return self.values[:-1]

    def last(self):
        #返回末尾元素
        return self.values[-1]

    def drop(self, n):
        #返回除前n个外的所有元素
        return self.values[n:]

    def take(self, n):
        #返回前n个元素
        return self.values[:n]

```

## 反射

你可以通过魔术方法控制使用 `isinstance()` 和 `issubclass()` 内置方法的反射行为。这些魔术方法是:

`__instancecheck__(self, instance)`

 v: latest ▾

检查一个实例是不是你定义的类的实例

```
__subclasscheck__(self, subclass)
```

检查一个类是不是你定义的类的子类

这些方法的用例似乎很少，这也许是真的。我不会花更多的时间在这些魔术方法上因为他们并不是很重要，但是他们的确反应了Python 中的面向对象编程的一些基本特性:非常容易的去做一些事情，即使并不是很必须。这些魔术方法看起来并不是很有用，但是当你需要的时候你会很高兴有这种特性。

## 可以调用的对象

你也许已经知道，在Python中，方法也是一种高等的对象。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，一个特殊的魔术方法可以让类的实例的行为表现的像函数一样，你可以调用他们，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__` 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。用一个实例来表达最好不过了：

---

```
class Entity:
    '''调用实体来改变实体的位置。'''

    def __init__(self, size, x, y):
        self.x, self.y = x, y
        self.size = size

    def __call__(self, x, y):
        '''改变实体的位置'''
        self.x, self.y = x, y
```

---

## 会话管理

在Python 2.5中，为了代码利用定义了一个新的关键词 `with` 语句。会话控制在Python中不罕见(之前是作为库的一部分被实现)，直到 [PEP343](#) 被添加后。它被成为一级语言结构。你也许之前看到这样的语句：

---

```
with open('foo.txt') as bar:
    # perform some action with bar
```

---

 v: latest ▾

会话控制器通过包装一个 `with` 语句来设置和清理行为。会话控制器的行为通过两个魔术方法来定义：`__enter__(self)` 定义当使用 `with` 语句的时候会话管理器应该初始块被创建的时候的行为。注意 `__enter__` 的返回值被 `with` 语句的目标或者 `as` 后的名字绑定。

`__exit__(self, exception_type, exception_value, traceback)` 定义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，清除工作或者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` 将会是 `None`。否则的话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，确认 `__exit__` 在所有结束之后会返回 `True`。如果你想让异常被会话管理器处理的话，那么就这样处理。

`__enter` 和 `__exit__` 对于明确有定义好的和日常行为的设置和清洁工作的类很有帮助。你也可以使用这些方法来创建一般的可以包装其他对象的会话管理器。以下是一个例子。

---

```
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
```

---

以下是一个使用 `Closer` 的例子，使用一个FTP链接来证明(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
```

---

你已经看到了我们的包装器如何静默的处理适当和不适当的使用行为。这是会话管理器和魔术方法的强大功能。

 v: latest ▾

## 创建对象的描述器

描述器是通过得到，设置，删除的时候被访问的类。当然也可以修改其他的对象。描述器并不是鼓励的，他们注定被一个所有者类所持有。当创建面向对象的数据库或者类，里面含有相互依赖的属性时，描述器将会非常有用。一种典型的使用方法是使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远点的距离信息)。

为了构建一个描述器，一个类必须有至少 `__get__` 或者 `__set__` 其中一个，并且 `__delete__` 被实现。让我们看看这些魔术方法。`__get__(self, instance, owner)` 定义当描述器的值被取得的时候的行为，`instance` 是拥有者对象的一个实例。`owner` 是拥有者类本身。`__set__(self, instance, value)` 定义当描述器值被改变时候的行为。`instance` 是拥有者类的一个实例 `value` 是要设置的值。`__delete__(self, instance)` 定义当描述器的值被删除的行为。`instance` 是拥有者对象的实例。 以下是一个描述器的实例:单位转换。

---

```
class Meter(object):
    '''Descriptor for a meter.'''

    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):
    '''Descriptor for a foot.'''

    def __get__(self, instance, owner):
        return instance.meter * 3.2808
    def __set__(self, instance, value):
        instance.meter = float(value) / 3.2808

class Distance(object):
    '''Class to represent distance holding two descriptors for feet and meters.'''
    meter = Meter()
    foot = Foot()
```

---

## 储存你的对象

如果你接触过其他的 Pythoner，你可能已经听说过 Pickle 了，Pickle 是用来序列化 Python 数据结构的模块，在你需要暂时存储一个对象的时候（比如缓存），这个模块非常的有用，不过这同时也是隐患的诞生地。

序列化数据是一个非常重要的功能，所以他不仅仅拥有相关的模块（Pickle，cPickle），还有自己的协议以及魔术方法，不过首先，我们先讨论下关于序列化内建数据结构的方法。

### Pickling: 简单例子

 v: latest ▾

让我们深入研究 Pickle，比如说你现在需要临时储存一个字典，你可以把它写入到一个文件里，并且要小心翼翼的确保格式正确，之后再用 `exec()` 或者处理文件输入来恢复数据，实际上这是很不安全的，如果你使用文本存储了一些重要的数据，任何方式的改变都可能会影响到你的程序，轻则程序崩溃，重则被恶意程序利用，所以，让我们用 Pickle 代替这种方式：

---

```
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()
```

---

嗯，过了几个小时之后，我们需要用到它了，只需把它 `unpickle` 了就行了：

---

```
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()
```

---

正如你期望的，数据原封不动的回来了！

同时要给你一句忠告： `pickle` 并不是很完美， `Pickle` 文件很容易被不小心或者故意损坏， `Pickle` 文件比纯文本文件要稍微安全一点，但是还是可以被利用运行恶意程序。 `Pickle` 不是跨版本兼容的（译注：最近刚好在《Python Cookbook》上看到相关讨论，书中描述的 `Pickle` 是跨版本兼容的，此点待验证），所以尽量不要去分发 `Pickle` 过的文本，因为别人并不一定能够打开。不过在做缓存或者其他需要序列化数据的时候， `Pickle` 还是很有用处的。

## 序列化你自己的对象

`Pickle` 并不是只支持内建数据结果，任何遵循 `Pickle` 协议的类都可以， `Pickle` 协议为 Python 对象规定了4个可选方法来自定义 `Pickle` 行为（对于 C 扩展的 `cPickle` 模块会有一些不同，但是这并不在我们的讨论范围内）：

```
__getinitargs__(self)
```

如果你希望在逆序列化的同时调用 `__init__`，你可以定义 `__getinitargs__` 方法，这个方法应该返回一系列你想被 `__init__` 调用的参数，注意这个方法只对老样式的类起作用。

```
__getnewargs__(self)
```

 v: latest ▾



对于新式的类，你可以定义任何在重建对象时候传递到 `__new__` 方法中的参数。这个方法也应该返回一系列的被 `__new__` 调用的参数。

```
__getstate__(self)
```

你可以自定义当对象被序列化时返回的状态，而不是使用 `__dict__` 方法，当逆序列化对象的时候，返回的状态将会被 `__setstate__` 方法调用。

```
__setstate__(self, state)
```

在对象逆序列化的时候，如果 `__setstate__` 定义过的话，对象的状态将被传给它而不是传给 `__dict__`。这个方法是和 `__getstate__` 配对的，当这两个方法都被定义的时候，你就可以完全控制整个序列化与逆序列化的过程了。

## 例子

我们以 `Slate` 为例，这是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，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
```

---


 v: latest ▾

## 结论

这份指南的希望为所有人都能带来一些知识，即使你是 Python 大牛或者对于精通于面向对象开发。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 Python 大牛，可能会帮助你想起来一些你已经遗忘的知识，或者一些你还没听说过的新功能。不管你现在有多少经验，我希望这次对于 Python 特殊方法的旅程能够带给你一些帮助（用双关语真的很不错 XD）（译注：这里的双关在于标题为 Magic Methods 这里是 神奇的旅程，不过由于中英语序的问题，直译略显头重脚轻，所以稍微变化了下意思，丢掉了双关的含义）。

## 附录：如何调用魔术方法

一些魔术方法直接和内建函数相对，在这种情况下，调用他们的方法很简单，但是，如果是另外一种不是特别明显的调用方法，这个附录介绍了很多并不是很明显的魔术方法的调用形式。

魔术方法	调用方式	解释
<code>__new__(cls [,...])</code>	<code>instance = MyClass(arg1, arg2)</code>	<code>__new__</code> 在创建实例的时候被调用
<code>__init__(self [,...])</code>	<code>instance = MyClass(arg1, arg2)</code>	<code>__init__</code> 在创建实例的时候被调用
<code>__cmp__(self, other)</code>	<code>self == other, self &gt; other, 等。</code>	在比较的时候调用
<code>__pos__(self)</code>	<code>+self</code>	一元加运算符
<code>__neg__(self)</code>	<code>-self</code>	一元减运算符
<code>__invert__(self)</code>	<code>~self</code>	取反运算符
<code>__index__(self)</code>	<code>x[self]</code>	对象被作为索引使用的时候
<code>__nonzero__(self)</code>	<code>bool(self)</code>	对象的布尔值
<code>__getattr__(self, name)</code>	<code>self.name</code> # <code>name</code> 不存在	访问一个不存在的属性时
<code>__setattr__(self, name, val)</code>	<code>self.name = val</code>	对一个属性赋值时
<code>__delattr__(self, name)</code>	<code>del self.name</code>	删除一个属性时
<code>__getattribute__(self, name)</code>	<code>self.name</code>	访问任何属性时
<code>__getitem__(self, key)</code>	<code>self[key]</code>	使用索引访问元素时
<code>__setitem__(self, key, val)</code>	<code>self[key] = val</code>	对某个索引值赋值时
<code>__delitem__(self, key)</code>	<code>del self[key]</code>	删除某个索引值时
<code>__iter__(self)</code>	<code>for x in self</code>	迭代时
<code>__contains__(self, value)</code>	<code>value in self, value not in self</code>	使用 <code>in</code> 操作测试关系时
<code>__concat__(self, value)</code>	<code>self + other</code>	连接两个对象时
<code>__call__(self [,...])</code>	<code>self(args)</code>	“调用”对象时
<code>__enter__(self)</code>	<code>with self as x:</code>	<code>with</code> 语句环境管  v: latest ▾
<code>__exit__(self, exc, val, trace)</code>	<code>with self as x:</code>	<code>with</code> 语句环境管理

魔术方法	调用方式	解释
<code>__getstate__(self)</code>	<code>pickle.dump(pk1_file, self)</code>	序列化
<code>__setstate__(self)</code>	<code>data = pickle.load(pk1_file)</code>	序列化

希望这个表格对你对于什么时候应该使用什么方法这个问题有所帮助。