# Python魔法方法指南

blog.csdn.net/wangjianno2/article/details/49050107

## 11.2. 构造方法

我们最为熟知的基本的魔法方法就是 *__init__* ，我们可以用它来指明一个对象初始化的行为。然而，当我们调用 *x = SomeClass()* 的时候， *__init__* 并不是第一个被调用的方法。事实上，第一个被调用的是 *__new__* ，这个方法才真正地创建了实例。当这个对象的生命周期结束的时候， *__del__* 会被调用。让我们近一步理解这三个方法：

- *__new__(cls,[...)*

  *__new__* 是对象实例化时第一个调用的方法，它只取下 *cls* 参数，并把其他参数传给 *__init__* 。 *__new__* 很少使用，但是也有它适合的场景，尤其是当类继承自一个像元组或者字符串这样不经常改变的类型的时候。我不打算深入讨论 *__new__* ，因为它并不是很有用， Python文档 中 有详细的说明。

- *__init__(self,[...])*

  类的初始化方法。它获取任何传给构造器的参数（比如我们调用 *x = SomeClass(10, 'foo')* ， *__init__* 就会接到参数 *10* 和 *'foo'* 。 *__init__*在Python的类定义中用的最多。

- *__del__(self)*

  *__new__* 和 *__init__* 是对象的构造器， *__del__* 是对象的销毁器。它并非实现了语句 *del x* (因此该语句不等同于 *x.__del__()*)。而是定义了当对象被垃圾回收时的行为。 当对象需要在销毁时做一些处理的时候这个方法很有用，比如 *socket*对象、文件对象。但是需要注意的是，当Python解释器退出但对象仍然存活的时候， *__del__* 并不会 执行。 所以养成一个手工清理的好习惯是很重要的，比如及时关闭连接。

这里有个 *__init__* 和 *__del__* 的例子:

```
from os.path import join

class FileObject:
    '''文件对象的装饰类，用来保证文件被删除时能够正确关闭。'''

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

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

## 11.3. 操作符

使用Python魔法方法的一个巨大优势就是可以构建一个拥有Python内置类型行为的对象。这意味着你可以避免使用非标准的、丑陋的方式来表达简单的操作。 在一些语言中，这样做很常见:

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

你当然可以在Python也这么做，但是这样做让代码变得冗长而混乱。不同的类库可能对同一种比较操作采用

不同的方法名称，这让使用者需要做很多没有必要的工作。运用魔法方法的魔力，我们可以定义方法 *__eq__*

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

这是魔法力量的一部分，这样我们就可以创建一个像内建类型那样的对象了！

## 11.3.1. 比较操作符

Python包含了一系列的魔法方法，用于实现对象之间直接比较，而不需要采用方法调用。同样也可以重载Python默认的比较方法，改变它们的行为。下面是这些方法的列表：

- *__cmp__(self, other)*

  *__cmp__* 是所有比较魔法方法中最基础的一个，它实际上定义了所有比较操作符的行为（<,==,!=,等等），但是它可能不能按照你需要的方式工作（例如，判断一个实例和另一个实例是否相等采用一套标准，而与判断一个实例是否大于另一实例采用另一套）。 *__cmp__* 应该在 *self < other* 时返回一个负整数，在 *self == other*时返回0，在 *self > other* 时返回正整数。最好只定义你所需要的比较形式，而不是一次定义全部。 如果你需要实现所有的比较形式，而且它们的判断标准类似，那么 *__cmp__* 是一个很好的方法，可以减少代码重复，让代码更简洁。

- *__eq__ `(self, other)*

  定义等于操作符(==)的行为。

- *__ne__(self, other)*

  定义不等于操作符(!=)的行为。

- *__lt__(self, other)*

  定义小于操作符(<)的行为。

- *__gt__(self, other)*

  定义大于操作符(>)的行为。

- *__le__(self, other)*

  定义小于等于操作符(<)的行为。

- *__ge__(self, other)*

  定义大于等于操作符(>)的行为。

举个例子，假如我们想用一个类来存储单词。我们可能想按照字典序（字母顺序）来比较单词，字符串的默认比较行为就是这样。我们可能也想按照其他规则来比较字符串，像是长度，或者音节的数量。在这个例子中，我们使用长度作为比较标准，下面是一种实现:

```
class Word(str):
    '''单词类，按照单词长度来定义比较行为'''

    def __new__(cls, word):
        # 注意，我们只能使用 __new__ ，因为str是不可变类型
        # 所以我们必须提前初始化它（在实例创建时）
        if ' ' in word:
            print "Value contains spaces. Truncating to first space."
            word = word[:word.index(' ')]
            # Word现在包含第一个空格前的所有字母
        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 *Word* 对象（ *Word('foo')* 和 *Word('bar')*)然后根据长度来比较它们。注意我们没有定义 *__eq__* 和 *__ne__* ，这是因为有时候它们会导致奇怪的结果（很明显， *Word('foo') == Word('bar')* 得到的结果会是true）。根据长度测试是否相等毫无意义，所以我们使用 *str* 的实现来比较相等。

从上面可以看到，不需要实现所有的比较魔法方法，就可以使用丰富的比较操作。标准库还在 *functools* 模块中提供了一个类装饰器，只要我们定义 *__eq__* 和另外一个操作符（ *__gt__*, *__lt__* 等），它就可以帮我们实现比较方法。这个特性只在 Python 2.7 中可用。当它可用时，它能帮助我们节省大量的时间和精力。要使用它，只需要它*@total_ordering* 放在类的定义之上就可以了

## 11.3.2. 数值操作符

就像你可以使用比较操作符来比较类的实例，你也可以定义数值操作符的行为。固定好你的安全带，这样的操作符真的有很多。看在组织的份上，我把它们分成了五类：一元操作符，常见算数操作符，反射算数操作符（后面会涉及更多），增强赋值操作符，和类型转换操作符。

### 11.3.2.1. 一元操作符

一元操作符只有一个操作符。

- *__pos__(self)*

  实现取正操作，例如 *+some_object*。

- *__neg__(self)*

  实现取负操作，例如 *-some_object*。

- *__abs__(self)*

  实现内建绝对值函数 *abs()* 操作。

- *__invert__(self)*

  实现取反操作符 ~。

- *__round__(self， n)*

  实现内建函数 *round()* ，n 是近似小数点的位数。

- *__floor__(self)*

  实现 *math.floor()* 函数，即向下取整。

- *__ceil__(self)*

  实现 *math.ceil()* 函数，即向上取整。

- *__trunc__(self)*

  实现 *math.trunc()* 函数，即距离零最近的整数。

## 11.3.2.2. 常见算数操作符

现在，我们来看看常见的二元操作符（和一些函数），像+，-，*之类的，它们很容易从字面意思理解。

- *__add__(self, other)*

  实现加法操作。

- *__sub__(self, other)*

  实现减法操作。

- *__mul__(self, other)*

  实现乘法操作。

- *__floordiv__(self, other)*

  实现使用 *//* 操作符的整数除法。

- *__div__(self, other)*

  实现使用 */* 操作符的除法。

- *__truediv__(self, other)*

  实现 *_true_* 除法，这个函数只有使用 *from __future__ import division* 时才有作用。

- *__mod__(self, other)*

  实现 *%* 取余操作。

- *__divmod__(self, other)*

  实现 *divmod* 内建函数。

- *__pow__*

  实现 ** 操作符。

- *__lshift__(self, other)*

  实现左移位运算符 << 。

- *__rshift__(self, other)*

  实现右移位运算符 >> 。

- *__and__(self, other)*

  实现按位与运算符 *&* 。

- *__or__(self, other)*

  实现按位或运算符 | 。

- *__xor__(self, other)*

  实现按位异或运算符 ^ 。

## 11.3.2.3. 反射算数运算符

还记得刚才我说会谈到反射运算符吗？可能你会觉得它是什么高端霸气上档次的概念，其实这东西挺简单的，下面举个例子:

```
some_object + other
```

这是“常见”的加法，反射是一样的意思，只不过是运算符交换了一下位置:

```
other + some_object
```

所有反射运算符魔法方法和它们的常见版本做的工作相同，只不过是处理交换连个操作数之后的情况。绝大多数情况下，反射运算和正常顺序产生的结果是相同的，所以很可能你定义 *__radd__* 时只是调用一下 *__add__*。注意一点，操作符左侧的对象（也就是上面的 *other* ）一定不要定义（或者产生 *NotImplemented* 异常） 操作符的非反射版本。例如，在上面的例子中，只有当 *other* 没有定义 *__add__* 时 *some_object.__radd__*才会被调用。

- *__radd__(self, other)*

  实现反射加法操作。

- *__rsub__(self, other)*

  实现反射减法操作。

- *__rmul__(self, other)*

  实现反射乘法操作。

- *__rfloordiv__(self, other)*

  实现使用 *//* 操作符的整数反射除法。

- *__rdiv__(self, other)*

  实现使用 / 操作符的反射除法。

- *__rtruediv__(self, other)*

  实现 *_true_* 反射除法，这个函数只有使用 *from __future__ import division*时才有作用。

- *__rmod__(self, other)*

  实现 % 反射取余操作符。

- *__rdivmod__(self, other)*

  实现调用 *divmod(other, self)* 时 *divmod* 内建函数的操作。

- *__rpow__*

  实现 ** 反射操作符。

- *__rlshift__(self, other)*

  实现反射左移位运算符 << 的作用。

- *__rshift__(self, other)*

  实现反射右移位运算符 >> 的作用。

- *__rand__(self, other)*

  实现反射按位与运算符 *&* 。

- *__ror__(self, other)*

  实现反射按位或运算符 | 。

- *__rxor__(self, other)*

  实现反射按位异或运算符 ^ 。

## 11.3.2.4. 增强赋值运算符

Python同样提供了大量的魔法方法，可以用来自定义增强赋值操作的行为。或许你已经了解增强赋值，它融合了“常见”的操作符和赋值操作，如果你还是没听明白，看下面的例子:

```
x = 5
x += 1 # 也就是 x = x + 1
```

这些方法都应该返回左侧操作数应该被赋予的值（例如， *a* += *b* *__iadd__* 也许会返回*a* + *b* ，这个结果会被赋给 a ）,下面是方法列表：

- *__iadd__(self, other)*

  实现加法赋值操作。

- *__isub__(self, other)*

  实现减法赋值操作。

- *__imul__(self, other)*

  实现乘法赋值操作。

- *__ifloordiv__(self, other)*

  实现使用 *//=* 操作符的整数除法赋值操作。

- *__idiv__(self, other)*

  实现使用 */=* 操作符的除法赋值操作。

- *__itruediv__(self, other)*

  实现 *_true_* 除法赋值操作，这个函数只有使用 *from __future__ import division* 时才有作用。

- *__imod__(self, other)*

  实现 *%=* 取余赋值操作。

- *__ipow__*

  实现 **= 操作。

- *__ilshift__(self, other)*

  实现左移位赋值运算符 <<= 。

- *__irshift__(self, other)*

  实现右移位赋值运算符 >>= 。

- *__iand__(self, other)*

  实现按位与运算符 *&=* 。

- *__ior__(self, other)*

  实现按位或赋值运算符 |= 。

- *__ixor__(self, other)*

  实现按位异或赋值运算符 ^= 。

## 11.3.2.5. 类型转换操作符

Python也有一系列的魔法方法用于实现类似 *float()* 的内建类型转换函数的操作。它们是这些：

- *__int__(self)*

  实现到int的类型转换。

- *__long__(self)*

  实现到long的类型转换。

- *__float__(self)*

  实现到float的类型转换。

- *__complex__(self)*

  实现到complex的类型转换。

- *__oct__(self)*

  实现到八进制数的类型转换。

- *__hex__(self)*

  实现到十六进制数的类型转换。

- *__index__(self)*

  实现当对象用于切片表达式时到一个整数的类型转换。如果你定义了一个可能会用于切片操作的数值类型，你应该定义 *__index__*。

- *__trunc__(self)*

  当调用 *math.trunc(self)* 时调用该方法， *__trunc__* 应该返回 *self* 截取到一个整数类型（通常是long类型）的值。

- *__coerce__(self)*

  该方法用于实现混合模式算数运算，如果不能进行类型转换， *__coerce__* 应该返回 *None* 。反之，它应该返回一个二元组 *self* 和 *other* ，这两者均已被转换成相同的类型。

## 11.4. 类的表示

使用字符串来表示类是一个相当有用的特性。在Python中有一些内建方法可以返回类的表示，相对应的，也有一系列魔法方法可以用来自定义在使用这些内建函数时类的行为。

- *__str__(self)*

  定义对类的实例调用 *str()* 时的行为。

- *__repr__(self)*

  定义对类的实例调用 *repr()* 时的行为。 *str()* 和 *repr()* 最主要的差别在于“目标用户”。 *repr()* 的作用是产生机器可读的输出（大部分情况下，其输出可以作为有效的Python代码），而 *str()* 则产生人类可读的输出。

- *__unicode__(self)*

  定义对类的实例调用 *unicode()* 时的行为。 *unicode()* 和 *str()* 很像，只是它返回unicode字符串。注意，如果调用者试图调用 *str()* 而你的类只实现了*__unicode__()* ，那么类将不能正常工作。所有你应该总是定义 *__str__()* ，以防有些人没有闲情雅致来使用unicode。

- *__format__(self)*

  定义当类的实例用于新式字符串格式化时的行为，例如， *"Hello, 0:abc!".format(a)* 会导致调用 *a.__format__("abc")* 。当定义你自己的数值类型或字符串类型时，你可能想提供某些特殊的格式化选项，这种情况下这个魔法方法会非常有用。

- *__hash__(self)*

  定义对类的实例调用 *hash()* 时的行为。它必须返回一个整数，其结果会被用于字典中键的快速比较。同时注意一点，实现这个魔法方法通常也需要实现 *__eq__* ，并且遵守如下的规则： *a == b* 意味着 *hash(a) == hash(b)*。

- *__nonzero__(self)*

  定义对类的实例调用 *bool()* 时的行为，根据你自己对类的设计，针对不同的实例，这个魔法方法应该相应地返回True或False。

- *__dir__(self)*

  定义对类的实例调用 *dir()* 时的行为，这个方法应该向调用者返回一个属性列表。一般来说，没必要自己实现 *__dir__* 。但是如果你重定义了 *__getattr__* 或者*__getattribute__* （下个部分会介绍），乃至使用动态生成的属性，以实现类的交互式使用，那么这个魔法方法是必不可少的。

到这里，我们基本上已经结束了魔法方法指南中无聊并且例子匮乏的部分。既然我们已经介绍了较为基础的魔法方法，是时候涉及更高级的内容了。

## 11.5. 访问控制

很多从其他语言转向Python的人都抱怨Python的类缺少真正意义上的封装（即没办法定义私有属性然后使用公有的getter和setter）。然而事实并非如此。实际上Python不是通过显式定义的字段和方法修改器，而是通过魔法方法实现了一系列的封装。

*__getattr__(self, name)*

当用户试图访问一个根本不存在（或者暂时不存在）的属性时，你可以通过这个魔法方法来定义类的行为。这个可以用于捕捉错误的拼写并且给出指引，使用废弃属性时给出警告（如果你愿意，仍然可以计算并且返回该属性），以及灵活地处理AttributeError。只有当试图访问不存在的属性时它才会被调用，所以这不能算是一个真正的封装的办法。

*__setattr__(self, name, value)*

和 *__getattr__* 不同， *__setattr__* 可以用于真正意义上的封装。它允许你自定义某个属性的赋值行为，不管这个属性存在与否，也就是说你可以对任意属性的任何变化都定义自己的规则。然后，一定要小心使用 *__setattr__* ，这个列表最后的例子中会有所展示。

*__delattr__(self, name)*

这个魔法方法和 *__setattr__* 几乎相同，只不过它是用于处理删除属性时的行为。和*_setattr__* 一样，使用它时

也需要多加小心，防止产生无限递归（在 *__delattr__* 的实现中调用 *del self.name* 会导致无限递归）。

*__getattribute__(self, name)*

` __getattribute__` 看起来和上面那些方法很合得来，但是最好不要使用它。*__getattribute__* 只能用于新式类。在最新版的Python中所有的类都是新式类，在老版Python中你可以通过继承 *object* 来创建新式类。*__getattribute__* 允许你自定义属性被访问时的行为，它也同样可能遇到无限递归问题（通过调用基类的*__getattribute__* 来避免）。*__getattribute__* 基本上可以替代 *__getattr__* 。只有当它被实现，并且显式地被调用，或者产生 *AttributeError* 时它才被使用。 这个魔法方法可以被使用（毕竟，选择权在你自己），我不推荐你使用它，因为它的使用范围相对有限（通常我们想要在赋值时进行特殊操作，而不是取值时），而且实现这个方法很容易出现Bug。

自定义这些控制属性访问的魔法方法很容易导致问题，考虑下面这个例子:

```
def __setattr__(self, name. value):
    self.name = value
    # 因为每次属性幅值都要调用 __setattr__()，所以这里的实现会导致递归
    # 这里的调用实际上是 self.__setattr('name', value)。因为这个方法一直
    # 在调用自己，因此递归将持续进行，直到程序崩溃

def __setattr__(self, name, value):
    self.__dict__[name] = value # 使用 __dict__ 进行赋值
    # 定义自定义行为
```

再次重申，Python的魔法方法十分强大，能力越强责任越大，了解如何正确的使用魔法方法更加重要。

到这里，我们对Python中自定义属性存取控制有了什么样的印象？它并不适合轻度的使用。实际上，它有些过分强大，而且违反直觉。然而它之所以存在，是因为一个更大的原则：Python不指望让杜绝坏事发生，而是想办法让做坏事变得困难。自由是至高无上的权利，你真的可以随心所欲。下面的例子展示了实际应用中某些特殊的属性访问方法（注意我们之所以使用 *super* 是因为不是所有的类都有 *__dict__* 属性）:

```
class AccessCounter(object):
    ''' 一个包含了一个值并且实现了访问计数器的类
    每次值的变化都会导致计数器自增'''

    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)
        # 使计数器自增变成不可避免
        # 如果你想阻止其他属性的赋值行为
        # 产生 AttributeError(name) 就可以了
        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)
```

## 11.6. 自定义序列

有许多办法可以让你的Python类表现得像是内建序列类型（字典，元组，列表，字符串等）。这些魔法方式是目前为止我最喜欢的。它们给了你难以置信的控制能力，可以让你的类与一系列的全局函数完美结合。在了解激动人心的内容之前，首先你需要掌握一些预备知识。

## 11.6.1. 预备知识

既然讲到创建自己的序列类型，就不得不说一说协议了。协议类似某些语言中的接口，里面包含的是一些必须实现的方法。在Python中，协议完全是非正式的，也不需要显式的声明，事实上，它们更像是一种参考标准。

为什么我们要讲协议？因为在Python中实现自定义容器类型需要用到一些协议。首先，不可变容器类型有如下协议：想实现一个不可变容器，你需要定义 *__len__* 和*__getitem__* (后面会具体说明）。可变容器的协议除了上面提到的两个方法之外，还需要定义 *__setitem__* 和 *__delitem__* 。最后，如果你想让你的对象可以迭代，你需要定义 *__iter__* ，这个方法返回一个迭代器。迭代器必须遵守迭代器协议，需要定义*__iter__* （返回它自己）和 *next* 方法。

## 11.6.2. 容器背后的魔法方法

- *__len__(self)*

  返回容器的长度，可变和不可变类型都需要实现。

- *__getitem__(self, key)*

  定义对容器中某一项使用 *self[key]* 的方式进行读取操作时的行为。这也是可变和不可变容器类型都需要实现的一个方法。它应该在键的类型错误式产生 *TypeError*异常，同时在没有与键值相匹配的内容时产生 *KeyError* 异常。

- *__setitem__(self, key)*

  定义对容器中某一项使用 *self[key]* 的方式进行赋值操作时的行为。它是可变容器类型必须实现的一个方法，同样应该在合适的时候产生 *KeyError* 和 *TypeError* 异常。

- *__iter__(self, key)*

  它应该返回当前容器的一个迭代器。迭代器以一连串内容的形式返回，最常见的是使用 *iter()* 函数调用，以及在类似 *for x in container:* 的循环中被调用。迭代器是他们自己的对象，需要定义 *__iter__* 方法并在其中返回自己。

- *__reversed__(self)*

  定义了对容器使用 *reversed()* 内建函数时的行为。它应该返回一个反转之后的序列。当你的序列类是有序时，类似列表和元组，再实现这个方法，

- *__contains__(self, item)*

  *__contains__* 定义了使用 *in* 和 *not in* 进行成员测试时类的行为。你可能好奇为什么这个方法不是序列协议的一部分，原因是，如果 *__contains__* 没有定义，Python就会迭代整个序列，如果找到了需要的一项就返回 *True* 。

- *__missing__(self ,key)*

  *__missing__* 在字典的子类中使用，它定义了当试图访问一个字典中不存在的键时的行为（目前为止是指字典的实例，例如我有一个字典 *d* ， *“george”* 不是字典中的一个键，当试图访问 *d[“george’]* 时就会调用 *d.__missing__(“george”)*）。

## 11.6.3. 一个例子

让我们来看一个实现了一些函数式结构的列表，可能在其他语言中这种结构更常见（例如Haskell）：

```
class FunctionalList:
    '''一个列表的封装类，实现了一些额外的函数式
    方法，例如head, tail, init, last, drop和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):
        # 如果键的类型或值不合法，列表会返回异常
        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):
        # 取得第一个元素
        return self.values[0]

    def tail(self):
        # 取得除第一个元素外的所有元素
        return self.valuse[1:]

    def init(self):
        # 取得除最后一个元素外的所有元素
        return self.values[:-1]

    def last(self):
        # 取得最后一个元素
        return self.values[-1]

    def drop(self, n):
        # 取得除前n个元素外的所有元素
        return self.values[n:]

    def take(self, n):
        # 取得前n个元素
        return self.values[:n]
```

就是这些，一个（微不足道的）有用的例子，向你展示了如何实现自己的序列。当然啦，自定义序列有更大的用处，而且绝大部分都在标准库中实现了（Python是自带电池的，记得吗？），
像 *Counter* , *OrderedDict* 和 *NamedTuple* 。

## 11.7. 反射

你可以通过定义魔法方法来控制用于反射的内建函数 *isinstance* 和 *issubclass* 的行为。下面是对应的魔法方法：

- *__instancecheck__(self, instance)*

  检查一个实例是否是你定义的类的一个实例（例如 *isinstance(instance, class)*）。

- *__subclasscheck__(self, subclass)*

  检查一个类是否是你定义的类的子类（例如 *issubclass(subclass, class)*）。

这几个魔法方法的适用范围看起来有些窄，事实也正是如此。我不会在反射魔法方法上花费太多时间，因为相比其他魔法方法它们显得不是很重要。但是它们展示了在Python中进行面向对象编程（或者总体上使用Python进行编程）时很重要的一点：不管做什么事情，都会有一个简单方法，不管它常用不常用。这些魔法方法可能看起来没那么有用，但是当你真正需要用到它们的时候，你会感到很幸运，因为它们还在那儿（也因为你阅读了这本指南！）

## 11.8. 抽象基类

请参考 http://docs.python.org/2/library/abc.html.

## 11.9. 可调用的对象

你可能已经知道了，在Python中，函数是一等的对象。这意味着它们可以像其他任何对象一样被传递到函数和方法中，这是一个十分强大的特性。

Python中一个特殊的魔法方法允许你自己类的对象表现得像是函数，然后你就可以"调用"它们，把它们传递到使用函数做参数的函数中，等等等等。这是另一个强大而且方便的特性，让使用Python编程变得更加幸福。

> *__call__(self, [args...])*
>
> 允许类的一个实例像函数那样被调用。本质上这代表了 *x()* 和 *x.__call__()* 是相同的。注意 *__call__* 可以有多个参数，这代表你可以像定义其他任何函数一样，定义 *__call__* ，喜欢用多少参数就用多少。

*__call__* 在某些需要经常改变状态的类的实例中显得特别有用。"调用"这个实例来改变它的状态，是一种更加符合直觉，也更加优雅的方法。一个表示平面上实体的类是一个不错的例子:

```
class Entity:
        '''表示一个实体的类，调用它的实例
        可以更新实体的位置'''

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

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

# 11.10. 上下文管理器

在Python 2.5中引入了一个全新的关键词，随之而来的是一种新的代码复用方法——*with*声明。上下文管理的概念在Python中并不是全新引入的（之前它作为标准库的一部分实现），直到PEP 343被接受，它才成为一种一级的语言结构。可能你已经见过这种写法了:

```
with open('foo.txt') as bar:
    # 使用bar进行某些操作
```

当对象使用 *with* 声明创建时，上下文管理器允许类做一些设置和清理工作。上下文管理器的行为由下面两个魔法方法所定义：

- *__enter__(self)*

  定义使用 *with* 声明创建的语句块最开始上下文管理器应该做些什么。注意*__enter__* 的返回值会赋给 *with* 声明的目标，也就是 *as* 之后的东西。

- *__exit__(self, exception_type, exception_value, traceback)*

  定义当 *with* 声明语句块执行完毕（或终止）时上下文管理器的行为。它可以用来处理异常，进行清理，或者做其他应该在语句块结束之后立刻执行的工作。如果语句块顺利执行， *exception_type* , *exception_value* 和 *traceback* 会是 *None* 。否则，你可以选择处理这个异常或者让用户来处理。如果你想处理异常，确保*__exit__* 在完成工作之后返回 *True* 。如果你不想处理异常，那就让它发生吧。

对一些具有良好定义的且通用的设置和清理行为的类，*__enter__* 和 *__exit__* 会显得特别有用。你也可以使用这几个方法来创建通用的上下文管理器，用来包装其他对象。下面是一个例子:

```
class Closer:
    '''一个上下文管理器，可以在with语句中
    使用close()自动关闭对象'''

    def __init__(self, obj):
        self.obj = obj

    def __enter__(self, obj):
        return self.obj # 绑定到目标

    def __exit__(self, exception_type, exception_value, traceback):
        try:
                self.obj.close()
        except AttributeError: # obj不是可关闭的
                print 'Not closable.'
                return True # 成功地处理了异常
```

这是一个 *Closer* 在实际使用中的例子，使用一个FTP连接来演示（一个可关闭的socket):

```
>>> from magicmethods import Closer
>>> from ftplib import FTP
>>> with Closer(FTP('ftp.somesite.com')) as conn:
...         conn.dir()
...
# 为了简单，省略了某些输出
>>> conn.dir()
# 很长的 AttributeError 信息，不能使用一个已关闭的连接
>>> with Closer(int(5)) as i:
...         i += 1
...
Not closable.
>>> i
6
```

看到我们的包装器是如何同时优雅地处理正确和不正确的调用了吗？这就是上下文管理器和魔法方法的力量。Python标准库包含一个 *contextlib* 模块，里面有一个上下文管理器 *contextlib.closing()* 基本上和我们的包装器完成的是同样的事情（但是没有包含任何当对象没有close()方法时的处理）。

## 11.11. 创建描述符对象

描述符是一个类，当使用取值，赋值和删除 时它可以改变其他对象。描述符不是用来单独使用的，它们需要被一个拥有者类所包含。描述符可以用来创建面向对象数据库，以及创建某些属性之间互相依赖的类。描述符在表现具有不同单位的属性，或者需要计算的属性时显得特别有用（例如表现一个坐标系中的点的类，其中的距离原点的距离这种属性）。

要想成为一个描述符，一个类必须具有实现 *__get__* , *__set__* 和 *__delete__* 三个方法中至少一个。

让我们一起来看一看这些魔法方法：

- *__get__(self, instance, owner)*

  定义当试图取出描述符的值时的行为。 *instance* 是拥有者类的实例， *owner* 是拥有者类本身。

- *__set__(self, instance, owner)*

  定义当描述符的值改变时的行为。 *instance* 是拥有者类的实例， *value* 是要赋给描述符的值。

- *__delete__(self, instance, owner)*

  定义当描述符的值被删除时的行为。 *instance* 是拥有者类的实例

现在，来看一个描述符的有效应用：单位转换:

```
class Meter(object):
    '''米的描述符。'''

    def __init__(self, value=0.0):
        self.value = float(value)
    def __get__(self, instance, owner):
        return self.value
    def __set__(self, instance, owner):
            self.value = float(value)

class Foot(object):
    '''英尺的描述符。'''

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

class Distance(object):
    '''用于描述距离的类，包含英尺和米两个描述符。'''
    meter = Meter()
    foot = Foot()
```

# 11.12. 拷贝

有些时候，特别是处理可变对象时，你可能想拷贝一个对象，改变这个对象而不影响原有的对象。这时就需要用到Python的 *copy* 模块了。然而（幸运的是），Python模块并不具有感知能力， 因此我们不用担心某天基于Linux的机器人崛起。但是我们的确需要告诉Python如何有效率地拷贝对象。

- *__copy__(self)*

  定义对类的实例使用 *copy.copy()* 时的行为。 *copy.copy()* 返回一个对象的浅拷贝，这意味着拷贝出的实例是全新的，然而里面的数据全都是引用的。也就是说，对象本身是拷贝的，但是它的数据还是引用的（所以浅拷贝中的数据更改会影响原对象）。

- *__deepcopy__(self, memodict=)*

  定义对类的实例使用 *copy.deepcopy()* 时的行为。 *copy.deepcopy()* 返回一个对象的深拷贝，这个对象和它的数据全都被拷贝了一份。 *memodict* 是一个先前拷贝对象的缓存，它优化了拷贝过程，而且可以防止拷贝递归数据结构时产生无限递归。当你想深拷贝一个单独的属性时，在那个属性上调用 *copy.deepcopy()* ，使用*memodict* 作为第一个参数。

这些魔法方法有什么用武之地呢？像往常一样，当你需要比默认行为更加精确的控制时。例如，如果你想拷贝一个对象，其中存储了一个字典作为缓存（可能会很大），拷贝缓存可能是没有意义的。如果这个缓存可以在内存中被不同实例共享，那么它就应该被共享。

# 11.13. Pickling

如果你和其他的Python爱好者共事过，很可能你已经听说过Pickling了。Pickling是Python数据结构的序列化过程，当你想存储一个对象稍后再取出读取时，Pickling会显得十分有用。然而它同样也是担忧和混淆的主要来源。

Pickling是如此的重要，以至于它不仅仅有自己的模块（ *pickle* ），还有自己的协议和魔法方法。首先，我们先来简要的介绍一下如何pickle已存在的对象类型（如果你已经知道了，大可跳过这部分内容）。

## 11.13.1. Pickling : 小试牛刀

我们一起来pickle吧。假设你有一个字典，你想存储它，稍后再取出来。你可以把它的内容写入一个文件，小心翼翼地确保使用了正确地格式，要把它读取出来，你可以使用*exec()* 或处理文件输入。但是这种方法并不可靠：如果你使用纯文本来存储重要数据，数据很容易以多种方式被破坏或者修改，导致你的程序崩溃，更糟糕的情况下，还可能在你的计算机上运行恶意代码。因此，我们要pickle它:

```
import pickle

data = {'foo': [1,2,3],
        'bar': ('Hello', 'world!'),
        'baz': True}
jar = open('data.pkl', 'wb')
pickle.dump(data, jar) # 将pickle后的数据写入jar文件
jar.close()
```

过了几个小时，我们想把它取出来，我们只需要反pickle它:

```
import pickle

pkl_file = open('data.pkl', 'rb') # 与pickle后的数据连接
data = pickle.load(pkl_file) # 把它加载进一个变量
print data
pkl_file.close()
```

将会发生什么？正如你期待的，它就是我们之前的 *data* 。

现在，还需要谨慎地说一句： pickle并不完美。Pickle文件很容易因为事故或被故意的破坏掉。Pickling或许比纯文本文件安全一些，但是依然有可能被用来运行恶意代码。而且它还不支持跨Python版本，所以不要指望分发pickle对象之后所有人都能正确地读取。然而不管怎么样，它依然是一个强有力的工具，可以用于缓存和其他类型的持久化工作。

## 11.13.2. Pickle你的对象

Pickle不仅仅可以用于内建类型，任何遵守pickle协议的类都可以被pickle。Pickle协议有四个可选方法，可以让类自定义它们的行为（这和C语言扩展略有不同，那不在我们的讨论范围之内）。

- *__getinitargs__(self)*

  如果你想让你的类在反pickle时调用 *__init__* ，你可以定义*__getinitargs__(self)* ，它会返回一个参数元组，这个元组会传递给 *__init__*。注意，这个方法只能用于旧式类。

- *__getnewargs__(self)*

  对新式类来说，你可以通过这个方法改变类在反pickle时传递给 *__new__* 的参数。这个方法应该返回一个参数元组。

- *__getstate__(self)*

  你可以自定义对象被pickle时被存储的状态，而不使用对象的 *__dict__* 属性。 这个状态在对象被反pickle时会被 *__setstate__* 使用。

- *__setstate__(self)*

  当一个对象被反pickle时，如果定义了 *__setstate__* ，对象的状态会传递给这个魔法方法，而不是直接应用到对象的 *__dict__* 属性。这个魔法方法和*__getstate__* 相互依存：当这两个方法都被定义时，你可以在Pickle时使用任何方法保存对象的任何状态。

- *__reduce__(self)*

  当定义扩展类型时（也就是使用Python的C语言API实现的类型），如果你想pickle它们，你必须告诉Python如何pickle它们。 *__reduce__* 被定义之后，当对象被Pickle时就会被调用。它要么返回一个代表全局名称的字符串，Pyhton会查找它并pickle，要么返回一个元组。这个元组包含2到5个元素，其中包括：一个可调用的对象，用于重建对象时调用；一个参数元素，供那个可调用对象使用；被传递给*__setstate__* 的状态（可选）；一个产生被pickle的列表元素的迭代器（可选）；一个产生被pickle的字典元素的迭代器（可选）；

- *__reduce_ex__(self)*

  *__reduce_ex__* 的存在是为了兼容性。如果它被定义，在pickle时*__reduce_ex__* 会代替 *__reduce__* 被调用。 *__reduce__* 也可以被定义，用于不支持 *__reduce_ex__* 的旧版pickle的API调用。

## 11.13.3. 一个例子

我们的例子是 *Slate* ，它会记住它的值曾经是什么，以及那些值是什么时候赋给它的。然而 每次被pickle时它都会变成空白，因为当前的值不会被存储:

```
import time

class Slate:
    '''存储一个字符串和一个变更日志的类
    每次被pickle都会忘记它当前的值'''

    def __init__(self, value):
        self.value = value
        self.last_change = time.asctime()
        self.history = {}

    def change(self, new_value):
        # 改变当前值，将上一个值记录到历史
        self.history[self.last_change] = self.value
        self.value = new_value)
        self.last_change = time.asctime()

    def print_change(self):
        print 'Changelog for Slate object:'
        for k,v in self.history.items():
            print '%s\t %s' % (k,v)

    def __getstate__(self):
        # 故意不返回self.value或self.last_change
        # 我们想在反pickle时得到一个空白的slate
        return self.history

    def __setstate__(self):
        # 使self.history = slate，last_change
        # 和value为未定义
        self.history = state
        self.value, self.last_change = None, None
```

## 11.14. 总结

这本指南的目标是使所有阅读它的人都能有所收获，无论他们有没有使用Python或者进行面向对象编程的经验。如果你刚刚开始学习Python，你会得到宝贵的基础知识，了解如何写出具有丰富特性的，优雅而且易用的类。如果你是中级的Python程序员，你或许能掌握一些新的概念和技巧，以及一些可以减少代码行数的好办法。如果你是专家级别的Python爱好者，你又重新复习了一遍某些可能已经忘掉的知识，也可能顺便了解了一些新技巧。无论你的水平怎样，我希望这趟遨游Python特殊方法的旅行，真的对你产生了魔法般的效果（实在忍不住不说最后这个双关）。

## 11.15. 附录1：如何调用魔法方法

一些魔法方法直接和内建函数对应，这种情况下，如何调用它们是显而易见的。然而，另外的情况下，调用魔法方法的途径并不是那么明显。这个附录旨在展示那些不那么明显的调用魔法方法的语法。

| 魔法方法 | 什么时候被调用 | 解释 |
|---|---|---|
| __new__(cls [,...]) | instance = MyClass(arg1, arg2) | __new__在实例创建时调用 |
| __init__(self [,...]) | instance = MyClass(arg1,arg2) | __init__在实例创建时调用 |
| __cmp__(self) | self == other, self > other 等 | 进行比较时调用 |
| __pos__(self) | +self | 一元加法符号 |
| __neg__(self) | -self | 一元减法符号 |
| __invert__(self) | ~self | 按位取反 |
| __index__(self) | x[self] | 当对象用于索引时 |
| __nonzero__(self) | bool(self) | 对象的布尔值 |
| __getattr__(self, name) | self.name #name不存在 | 访问不存在的属性 |
| __setattr__(self, name) | self.name = val | 给属性赋值 |
| __delattr_(self, name) | del self.name | 删除属性 |
| __getattribute__(self,name) | self.name | 访问任意属性 |
| __getitem__(self, key) | self[key] | 使用索引访问某个元素 |
| __setitem__(self, key) | self[key] = val | 使用索引给某个元素赋值 |

| 魔法方法 | 什么时候被调用 | 解释 |
| --- | --- | --- |
| __delitem__(self, key) | del self[key] | 使用索引删除某个对象 |
| __iter__(self) | for x in self | 迭代 |
| __contains__(self, value) | value in self, value not in self | 使用in进行成员测试 |
| __call__(self [,...]) | self(args) | “调用”一个实例 |
| __enter__(self) | with self as x: | with声明的上下文管理器 |
| __exit__(self, exc, val, trace) | with self as x: | with声明的上下文管理器 |
| __getstate__(self) | pickle.dump(pkl_file, self) | Pickling |
| __setstate__(self) | data = pickle.load(pkl_file) | Pickling |

## 11.16. 附录2：Python 3中的变化

在这里，我们记录了几个在对象模型方面 Python 3 和 Python 2.x 之间的主要区别。

- Python 3中string和unicode的区别不复存在，因此*__unicode__* 被取消了，*__bytes__* 加入进来（与 Python 2.7 中的 *__str__* 和 *__unicode__* 行为类似），用于新的创建字节数组的内建方法。
- Python 3中默认除法变成了 true 除法，因此*__div__* 被取消了。
- *__coerce__* 被取消了，因为和其他魔法方法有功能上的重复，以及本身行为令人迷惑。
- *__cmp__* 被取消了，因为和其他魔法方法有功能上的重复。
- *__nonzero__* 被重命名成 *__bool__* 。

转载资料来源于：

http://pyzh.readthedocs.org/en/latest/python-magic-methods-guide.html