8.1 改变对象的字符串显示¶

问题¶

你想改变对象实例的打印或显示输出，让它们更具可读性。

解决方案¶

要改变一个实例的字符串表示，可重新定义它的 `__str__()` 和 `__repr__()` 方法。例如：

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)

    def __str__(self):
        return '({0.x!s}, {0.y!s})'.format(self)
```

`__repr__()` 方法返回一个实例的代码表示形式，通常用来重新构造这个实例。 内置的 `repr()` 函数返回这个字符串，跟我们使用交互式解释器显示的值是一样的。 `__str__()` 方法将实例转换为一个字符串，使用 `str()` 或 `print()` 函数会输出这个字符串。比如：

```
>>> p = Pair(3, 4)
>>> p
Pair(3, 4) # __repr__() output
>>> print(p)
(3, 4) # __str__() output
>>>
```

我们在这里还演示了在格式化的时候怎样使用不同的字符串表现形式。 特别来讲，`!r` 格式化代码指明输出使用 `__repr__()` 来代替默认的 `__str__()` 。 你可以用前面的类来试着测试下：

```
>>> p = Pair(3, 4)
>>> print('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print('p is {0}'.format(p))
p is (3, 4)
>>>
```

讨论¶

自定义 `__repr__()` 和 `__str__()` 通常是很好的习惯，因为它能简化调试和实例输出。 例如，如果仅仅只是打印输出或日志输出某个实例，那么程序员会看到实例更加详细与有用的信息。

`__repr__()` 生成的文本字符串标准做法是需要让 `eval(repr(x)) == x` 为真。 如果实在不能这样子做，应该创建一个有用的文本表示，并使用 < 和 > 括起来。比如：

```
>>> f = open('file.dat')
>>> f
<_io.TextIOWrapper name='file.dat' mode='r' encoding='UTF-8'>
>>>
```

如果 `__str__()` 没有被定义，那么就会使用 `__repr__()` 来代替输出。

上面的 `format()` 方法的使用看上去很有趣，格式化代码 `{0.x}` 对应的是第1个参数的x属性。 因此，在下面的函数中，0实际上指的就是 `self` 本身：

```
def __repr__(self):
    return 'Pair({0.x!r}, {0.y!r})'.format(self)
```

作为这种实现的一个替代，你也可以使用 % 操作符，就像下面这样：

```
def __repr__(self):
    return 'Pair(%r, %r)' % (self.x, self.y)
```

8.10 使用延迟计算属性¶

问题¶

你想将一个只读属性定义成一个property，并且只在访问的时候才会计算结果。 但是一旦被访问后，你希望结果值被缓存起来，不用每次都去计算。

解决方案¶

定义一个延迟属性的一种高效方法是通过使用一个描述器类，如下所示：

```
class lazyproperty:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            value = self.func(instance)
            setattr(instance, self.func.__name__, value)
            return value
```

你需要像下面这样在一个类中使用它：

```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @lazyproperty
    def area(self):
        print('Computing area')
        return math.pi * self.radius ** 2

    @lazyproperty
    def perimeter(self):
        print('Computing perimeter')
        return 2 * math.pi * self.radius
```

下面在一个交互环境中演示它的使用：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.perimeter
Computing perimeter
25.132741228718345
>>> c.perimeter
25.132741228718345
>>>
```

仔细观察你会发现消息 `Computing area` 和 `Computing perimeter` 仅仅出现一次。

讨论¶

很多时候，构造一个延迟计算属性的主要目的是为了提升性能。 例如，你可以避免计算这些属性值，除非你真的需要它们。 这里演示的方案就是用来实现这样的效果的， 只不过它是通过以非常高效的方式使用描述器的一个精妙特性来

达到这种效果的。

正如在其他小节(如8.9小节)所讲的那样，当一个描述器被放入一个类的定义时， 每次访问属性时它的 `__get__()` 、`__set__()` 和 `__delete__()` 方法就会被触发。 不过，如果一个描述器仅仅只定义了一个 `__get__()` 方法的话，它比通常的具有更弱的绑定。 特别地，只有当被访问属性不在实例底层的字典中时 `__get__()` 方法才会被触发。

`lazyproperty` 类利用这一点，使用 `__get__()` 方法在实例中存储计算出来的值， 这个实例使用相同的名字作为它的property。 这样一来，结果值被存储在实例字典中并且以后就不需要再去计算这个property了。 你可以尝试更深入的例子来观察结果：

```
>>> c = Circle(4.0)
>>> # Get instance variables
>>> vars(c)
{'radius': 4.0}

>>> # Compute area and observe variables afterward
>>> c.area
Computing area
50.26548245743669
>>> vars(c)
{'area': 50.26548245743669, 'radius': 4.0}

>>> # Notice access doesn't invoke property anymore
>>> c.area
50.26548245743669

>>> # Delete the variable and see property trigger again
>>> del c.area
>>> vars(c)
{'radius': 4.0}
>>> c.area
Computing area
50.26548245743669
>>>
```

这种方案有一个小缺陷就是计算出的值被创建后是可以被修改的。例如：

```
>>> c.area
Computing area
50.26548245743669
>>> c.area = 25
>>> c.area
25
>>>
```

如果你担心这个问题，那么可以使用一种稍微没那么高效的实现，就像下面这样：

```
def lazyproperty(func):
    name = '_lazy_' + func.__name__
    @property
    def lazy(self):
        if hasattr(self, name):
            return getattr(self, name)
        else:
            value = func(self)
            setattr(self, name, value)
            return value
    return lazy
```

如果你使用这个版本，就会发现现在修改操作已经不被允许了：

```
>>> c = Circle(4.0)
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.area = 25
```

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

然而，这种方案有一个缺点就是所有get操作都必须被定向到属性的 `getter` 函数上去。 这个跟之前简单的在实例字典中查找值的方案相比效率要低一点。 如果想获取更多关于property和可管理属性的信息，可以参考8.6小节。而描述器的相关内容可以在8.9小节找到。

8.11 简化数据结构的初始化¶

问题¶

你写了很多仅仅用作数据结构的类，不想写太多烦人的 `__init__()` 函数

解决方案¶

可以在一个基类中写一个公用的 `__init__()` 函数：

```
import math

class Structure1:
    # Class variable that specifies expected fields
    _fields = []

    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))
        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)
```

然后使你的类继承自这个基类:

```
# Example class definitions
class Stock(Structure1):
    _fields = ['name', 'shares', 'price']

class Point(Structure1):
    _fields = ['x', 'y']

class Circle(Structure1):
    _fields = ['radius']

    def area(self):
        return math.pi * self.radius ** 2
```

使用这些类的示例：

```
>>> s = Stock('ACME', 50, 91.1)
>>> p = Point(2, 3)
>>> c = Circle(4.5)
>>> s2 = Stock('ACME', 50)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "structure.py", line 6, in __init__
        raise TypeError('Expected {} arguments'.format(len(self._fields)))
TypeError: Expected 3 arguments
```

如果还想支持关键字参数，可以将关键字参数设置为实例属性：

```
class Structure2:
    _fields = []

    def __init__(self, *args, **kwargs):
        if len(args) > len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set all of the positional arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the remaining keyword arguments
        for name in self._fields[len(args):]:
            setattr(self, name, kwargs.pop(name))
```

```
        # Check for any remaining unknown arguments
        if kwargs:
            raise TypeError('Invalid argument(s): {}'.format(','.join(kwargs)))
# Example use
if __name__ == '__main__':
    class Stock(Structure2):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, price=91.1)
    s3 = Stock('ACME', shares=50, price=91.1)
    # s3 = Stock('ACME', shares=50, price=91.1, aa=1)
```

你还能将不在 `_fields` 中的名称加入到属性中去：

```
class Structure3:
    # Class variable that specifies expected fields
    _fields = []

    def __init__(self, *args, **kwargs):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the additional arguments (if any)
        extra_args = kwargs.keys() - self._fields
        for name in extra_args:
            setattr(self, name, kwargs.pop(name))

        if kwargs:
            raise TypeError('Duplicate values for {}'.format(','.join(kwargs)))

# Example use
if __name__ == '__main__':
    class Stock(Structure3):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, 91.1, date='8/2/2012')
```

讨论¶

当你需要使用大量很小的数据结构类的时候， 相比手工一个个定义 `__init__()` 方法而已，使用这种方式可以大大简化代码。

在上面的实现中我们使用了 `setattr()` 函数类设置属性值， 你可能不想用这种方式，而是想直接更新实例字典，就像下面这样：

```
class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments (alternate)
        self.__dict__.update(zip(self._fields,args))
```

尽管这也可以正常工作，但是当定义子类的时候问题就来了。 当一个子类定义了 `__slots__` 或者通过property(或描述器)来包装某个属性， 那么直接访问实例字典就不起作用了。我们上面使用 `setattr()` 会显得更通用些，因为它也适用于子类情况。

这种方法唯一不好的地方就是对某些IDE而言，在显示帮助函数时可能不太友好。比如：

```
>>> help(Stock)
Help on class Stock in module __main__:
class Stock(Structure)
...
| Methods inherited from Structure:
|
| __init__(self, *args, **kwargs)
|
...
>>>
```

可以参考9.16小节来强制在 `__init__()` 方法中指定参数的类型签名。

8.12 定义接口或者抽象基类¶

问题¶

你想定义一个接口或抽象类，并且通过执行类型检查来确保子类实现了某些特定的方法

解决方案¶

使用 `abc` 模块可以很轻松的定义抽象基类：

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxbytes=-1):
        pass

    @abstractmethod
    def write(self, data):
        pass
```

抽象类的一个特点是它不能直接被实例化，比如你想像下面这样做是不行的：

```
a = IStream() # TypeError: Can't instantiate abstract class
                # IStream with abstract methods read, write
```

抽象类的目的就是让别的类继承它并实现特定的抽象方法：

```
class SocketStream(IStream):
    def read(self, maxbytes=-1):
        pass

    def write(self, data):
        pass
```

抽象基类的一个主要用途是在代码中检查某些类是否为特定类型，实现了特定接口：

```
def serialize(obj, stream):
    if not isinstance(stream, IStream):
        raise TypeError('Expected an IStream')
    pass
```

除了继承这种方式外，还可以通过注册方式来让某个类实现抽象基类：

```
import io

# Register the built-in I/O classes as supporting our interface
IStream.register(io.IOBase)

# Open a normal file and type check
f = open('foo.txt')
isinstance(f, IStream) # Returns True
```

`@abstractmethod` 还能注解静态方法、类方法和 `properties` 。 你只需保证这个注解紧靠在函数定义前即可：

```
class A(metaclass=ABCMeta):
    @property
    @abstractmethod
    def name(self):
        pass

    @name.setter
    @abstractmethod
    def name(self, value):
        pass
```

```
    @classmethod
    @abstractmethod
    def method1(cls):
        pass

    @staticmethod
    @abstractmethod
    def method2():
        pass
```

讨论¶

标准库中有很多用到抽象基类的地方。`collections` 模块定义了很多跟容器和迭代器(序列、映射、集合等)有关的抽象基类。 `numbers` 库定义了跟数字对象(整数、浮点数、有理数等)有关的基类。`io` 库定义了很多跟I/O操作相关的基类。

你可以使用预定义的抽象类来执行更通用的类型检查，例如：

```
import collections

# Check if x is a sequence
if isinstance(x, collections.Sequence):
...

# Check if x is iterable
if isinstance(x, collections.Iterable):
...

# Check if x has a size
if isinstance(x, collections.Sized):
...

# Check if x is a mapping
if isinstance(x, collections.Mapping):
```

尽管ABCs可以让我们很方便的做类型检查，但是我们在代码中最好不要过多的使用它。 因为Python的本质是一门动态编程语言，其目的就是给你更多灵活性， 强制类型检查或让你代码变得更复杂，这样做无异于舍本求末。

8.13 实现数据模型的类型约束¶

问题¶

你想定义某些在属性赋值上面有限制的数据结构。

解决方案¶

在这个问题中，你需要在对某些实例属性赋值时进行检查。 所以你要自定义属性赋值函数，这种情况下最好使用描述器。

下面的代码使用描述器实现了一个系统类型和赋值验证框架：

```
# Base class. Uses a descriptor to set a value
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value


# Descriptor for enforcing types
class Typed(Descriptor):
    expected_type = type(None)

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('expected ' + str(self.expected_type))
        super().__set__(instance, value)


# Descriptor for enforcing values
class Unsigned(Descriptor):
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super().__set__(instance, value)


class MaxSized(Descriptor):
    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super().__init__(name, **opts)

    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super().__set__(instance, value)
```

这些类就是你要创建的数据模型或类型系统的基础构建模块。 下面就是我们实际定义的各种不同的数据类型：

```
class Integer(Typed):
    expected_type = int

class UnsignedInteger(Integer, Unsigned):
    pass

class Float(Typed):
    expected_type = float

class UnsignedFloat(Float, Unsigned):
    pass
```

```
class String(Typed):
    expected_type = str

class SizedString(String, MaxSized):
    pass
```

然后使用这些自定义数据类型，我们定义一个类：

```
class Stock:
    # Specify constraints
    name = SizedString('name', size=8)
    shares = UnsignedInteger('shares')
    price = UnsignedFloat('price')

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

然后测试这个类的属性赋值约束，可发现对某些属性的赋值违法了约束是不合法的：

```
>>> s.name
'ACME'
>>> s.shares = 75
>>> s.shares = -10
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 17, in __set__
        super().__set__(instance, value)
    File "example.py", line 23, in __set__
        raise ValueError('Expected >= 0')
ValueError: Expected >= 0
>>> s.price = 'a lot'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 16, in __set__
        raise TypeError('expected ' + str(self.expected_type))
TypeError: expected <class 'float'>
>>> s.name = 'ABRACADABRA'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 17, in __set__
        super().__set__(instance, value)
    File "example.py", line 35, in __set__
        raise ValueError('size must be < ' + str(self.size))
ValueError: size must be < 8
>>>
```

还有一些技术可以简化上面的代码，其中一种是使用类装饰器：

```
# Class decorator to apply constraints
def check_attributes(**kwargs):
    def decorate(cls):
        for key, value in kwargs.items():
            if isinstance(value, Descriptor):
                value.name = key
                setattr(cls, key, value)
            else:
                setattr(cls, key, value(key))
        return cls

    return decorate

# Example
@check_attributes(name=SizedString(size=8),
                  shares=UnsignedInteger,
                  price=UnsignedFloat)
class Stock:
    def __init__(self, name, shares, price):
```

```
        self.name = name
        self.shares = shares
        self.price = price
```

另外一种方式是使用元类：

```
# A metaclass that applies checking
class checkedmeta(type):
    def __new__(cls, clsname, bases, methods):
        # Attach attribute names to the descriptors
        for key, value in methods.items():
            if isinstance(value, Descriptor):
                value.name = key
        return type.__new__(cls, clsname, bases, methods)

# Example
class Stock2(metaclass=checkedmeta):
    name = SizedString(size=8)
    shares = UnsignedInteger()
    price = UnsignedFloat()

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

讨论¶

本节使用了很多高级技术，包括描述器、混入类、`super()` 的使用、类装饰器和元类。 不可能在这里一一详细展开来讲，但是可以在8.9、8.18、9.19小节找到更多例子。 但是，我在这里还是要提一下几个需要注意的点。

首先，在 `Descriptor` 基类中你会看到有个 `__set__()` 方法，却没有相应的 `__get__()` 方法。 如果一个描述仅仅是从底层实例字典中获取某个属性值的话，那么没必要去定义 `__get__()` 方法。

所有描述器类都是基于混入类来实现的。比如 `Unsigned` 和 `MaxSized` 要跟其他继承自 `Typed` 类混入。 这里利用多继承来实现相应的功能。

混入类的一个比较难理解的地方是，调用 `super()` 函数时，你并不知道究竟要调用哪个具体类。 你需要跟其他类结合后才能正确的使用，也就是必须合作才能产生效果。

使用类装饰器和元类通常可以简化代码。上面两个例子中你会发现你只需要输入一次属性名即可了。

```
# Normal
class Point:
    x = Integer('x')
    y = Integer('y')

# Metaclass
class Point(metaclass=checkedmeta):
    x = Integer()
    y = Integer()
```

所有方法中，类装饰器方案应该是最灵活和最高明的。 首先，它并不依赖任何其他新的技术，比如元类。其次，装饰器可以很容易的添加或删除。

最后，装饰器还能作为混入类的替代技术来实现同样的效果;

```
# Decorator for applying type checking
def Typed(expected_type, cls=None):
    if cls is None:
        return lambda cls: Typed(expected_type, cls)
    super_set = cls.__set__

    def __set__(self, instance, value):
        if not isinstance(value, expected_type):
            raise TypeError('expected ' + str(expected_type))
```

```
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls


# Decorator for unsigned values
def Unsigned(cls):
    super_set = cls.__set__

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls


# Decorator for allowing sized values
def MaxSized(cls):
    super_init = cls.__init__

    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super_init(self, name, **opts)

    cls.__init__ = __init__

    super_set = cls.__set__

    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls


# Specialized descriptors
@Typed(int)
class Integer(Descriptor):
    pass


@Unsigned
class UnsignedInteger(Integer):
    pass


@Typed(float)
class Float(Descriptor):
    pass


@Unsigned
class UnsignedFloat(Float):
    pass


@Typed(str)
class String(Descriptor):
    pass


@MaxSized
class SizedString(String):
    pass
```

这种方式定义的类跟之前的效果一样，而且执行速度会更快。 设置一个简单的类型属性的值，装饰器方式要比之前的混入类的方式几乎快100%。 现在你应该庆幸自己读完了本节全部内容了吧？^_^

8.14 实现自定义容器¶

问题¶

你想实现一个自定义的类来模拟内置的容器类功能，比如列表和字典。但是你不确定到底要实现哪些方法。

解决方案¶

`collections` 定义了很多抽象基类，当你想自定义容器类的时候它们会非常有用。 比如你想让你的类支持迭代，那就让你的类继承 `collections.Iterable` 即可：

```
import collections
class A(collections.Iterable):
    pass
```

不过你需要实现 `collections.Iterable` 所有的抽象方法，否则会报错:

```
>>> a = A()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class A with abstract methods __iter__
>>>
```

你只要实现 `__iter__()` 方法就不会报错了(参考4.2和4.7小节)。

你可以先试着去实例化一个对象，在错误提示中可以找到需要实现哪些方法：

```
>>> import collections
>>> collections.Sequence()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Sequence with abstract methods \
__getitem__, __len__
>>>
```

下面是一个简单的示例，继承自上面Sequence抽象类，并且实现元素按照顺序存储：

```
class SortedItems(collections.Sequence):
    def __init__(self, initial=None):
        self._items = sorted(initial) if initial is not None else []

    # Required sequence methods
    def __getitem__(self, index):
        return self._items[index]

    def __len__(self):
        return len(self._items)

    # Method for adding an item in the right location
    def add(self, item):
        bisect.insort(self._items, item)


items = SortedItems([5, 1, 3])
print(list(items))
print(items[0], items[-1])
items.add(2)
print(list(items))
```

可以看到，SortedItems跟普通的序列没什么两样，支持所有常用操作，包括索引、迭代、包含判断，甚至是切片操作。

这里面使用到了 `bisect` 模块，它是一个在排序列表中插入元素的高效方式。可以保证元素插入后还保持顺序。

讨论¶

使用 `collections` 中的抽象基类可以确保你自定义的容器实现了所有必要的方法。并且还能简化类型检查。 你的自定义容器会满足大部分类型检查需要，如下所示：

```
>>> items = SortedItems()
>>> import collections
>>> isinstance(items, collections.Iterable)
True
>>> isinstance(items, collections.Sequence)
True
>>> isinstance(items, collections.Container)
True
>>> isinstance(items, collections.Sized)
True
>>> isinstance(items, collections.Mapping)
False
>>>
```

`collections` 中很多抽象类会为一些常见容器操作提供默认的实现， 这样一来你只需要实现那些你最感兴趣的方法即可。假设你的类继承自 `collections.MutableSequence` ，如下：

```
class Items(collections.MutableSequence):
    def __init__(self, initial=None):
        self._items = list(initial) if initial is not None else []

    # Required sequence methods
    def __getitem__(self, index):
        print('Getting:', index)
        return self._items[index]

    def __setitem__(self, index, value):
        print('Setting:', index, value)
        self._items[index] = value

    def __delitem__(self, index):
        print('Deleting:', index)
        del self._items[index]

    def insert(self, index, value):
        print('Inserting:', index, value)
        self._items.insert(index, value)

    def __len__(self):
        print('Len')
        return len(self._items)
```

如果你创建 `Items` 的实例，你会发现它支持几乎所有的核心列表方法(如append()、remove()、count()等)。 下面是使用演示：

```
>>> a = Items([1, 2, 3])
>>> len(a)
Len
3
>>> a.append(4)
Len
Inserting: 3 4
>>> a.append(2)
Len
Inserting: 4 2
>>> a.count(2)
Getting: 0
Getting: 1
Getting: 2
Getting: 3
Getting: 4
Getting: 5
2
```

```
>>> a.remove(3)
Getting: 0
Getting: 1
Getting: 2
Deleting: 2
>>>
```

本小节只是对Python抽象类功能的抛砖引玉。`numbers` 模块提供了一个类似的跟整数类型相关的抽象类型集合。 可以参考8.12小节来构造更多自定义抽象基类。

8.15 属性的代理访问¶

问题¶

你想将某个实例的属性访问代理到内部另一个实例中去，目的可能是作为继承的一个替代方法或者实现代理模式。

解决方案¶

简单来说，代理是一种编程模式，它将某个操作转移给另外一个对象来实现。 最简单的形式可能是像下面这样：

```
class A:
    def spam(self, x):
        pass

    def foo(self):
        pass


class B1:
    """简单的代理"""

    def __init__(self):
        self._a = A()

    def spam(self, x):
        # Delegate to the internal self._a instance
        return self._a.spam(x)

    def foo(self):
        # Delegate to the internal self._a instance
        return self._a.foo()

    def bar(self):
        pass
```

如果仅仅就两个方法需要代理，那么像这样写就足够了。但是，如果有大量的方法需要代理， 那么使用 `__getattr__()` 方法或许更好些：

```
class B2:
    """使用__getattr__的代理，代理方法比较多时候"""

    def __init__(self):
        self._a = A()

    def bar(self):
        pass

    # Expose all of the methods defined on class A
    def __getattr__(self, name):
        """这个方法在访问的attribute不存在的时候被调用
        the __getattr__() method is actually a fallback method
        that only gets called when an attribute is not found"""
        return getattr(self._a, name)
```

`__getattr__` 方法是在访问attribute不存在的时候被调用，使用演示：

```
b = B()
b.bar() # Calls B.bar() (exists on B)
b.spam(42) # Calls B.__getattr__('spam') and delegates to A.spam
```

另外一个代理例子是实现代理模式，例如：

```
# A proxy class that wraps around another object, but
# exposes its public attributes
class Proxy:
    def __init__(self, obj):
```

```
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        print('getattr:', name)
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)
        else:
            print('setattr:', name, value)
            setattr(self._obj, name, value)

    # Delegate attribute deletion
    def __delattr__(self, name):
        if name.startswith('_'):
            super().__delattr__(name)
        else:
            print('delattr:', name)
            delattr(self._obj, name)
```

使用这个代理类时，你只需要用它来包装下其他类即可：

```
class Spam:
    def __init__(self, x):
        self.x = x

    def bar(self, y):
        print('Spam.bar:', self.x, y)

# Create an instance
s = Spam(2)
# Create a proxy around it
p = Proxy(s)
# Access the proxy
print(p.x)  # Outputs 2
p.bar(3)  # Outputs "Spam.bar: 2 3"
p.x = 37  # Changes s.x to 37
```

通过自定义属性访问方法，你可以用不同方式自定义代理类行为(比如加入日志功能、只读访问等)。

讨论¶

代理类有时候可以作为继承的替代方案。例如，一个简单的继承如下：

```
class A:
    def spam(self, x):
        print('A.spam', x)
    def foo(self):
        print('A.foo')

class B(A):
    def spam(self, x):
        print('B.spam')
        super().spam(x)
    def bar(self):
        print('B.bar')
```

使用代理的话，就是下面这样：

```
class A:
    def spam(self, x):
        print('A.spam', x)
    def foo(self):
        print('A.foo')

class B:
```

```
    def __init__(self):
        self._a = A()
    def spam(self, x):
        print('B.spam', x)
        self._a.spam(x)
    def bar(self):
        print('B.bar')
    def __getattr__(self, name):
        return getattr(self._a, name)
```

当实现代理模式时，还有些细节需要注意。 首先，`__getattr__()` 实际是一个后备方法，只有在属性不存在时才会调用。 因此，如果代理类实例本身有这个属性的话，那么不会触发这个方法的。 另外，`__setattr__()` 和 `__delattr__()` 需要额外的魔法来区分代理实例和被代理实例 `_obj` 的属性。 一个通常的约定是只代理那些不以下划线 _ 开头的属性(代理类只暴露被代理类的公共属性)。

还有一点需要注意的是，`__getattr__()` 对于大部分以双下划线(__)开始和结尾的属性并不适用。 比如，考虑如下的类：

```
class ListLike:
    """__getattr__对于双下划线开始和结尾的方法是不能用的，需要一个个去重定义"""

    def __init__(self):
        self._items = []

    def __getattr__(self, name):
        return getattr(self._items, name)
```

如果是创建一个ListLike对象，会发现它支持普通的列表方法，如append()和insert()， 但是却不支持len()、元素查找等。例如：

```
>>> a = ListLike()
>>> a.append(2)
>>> a.insert(0, 1)
>>> a.sort()
>>> len(a)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: object of type 'ListLike' has no len()
>>> a[0]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'ListLike' object does not support indexing
>>>
```

为了让它支持这些方法，你必须手动的实现这些方法代理：

```
class ListLike:
    """__getattr__对于双下划线开始和结尾的方法是不能用的，需要一个个去重定义"""

    def __init__(self):
        self._items = []

    def __getattr__(self, name):
        return getattr(self._items, name)

    # Added special methods to support certain list operations
    def __len__(self):
        return len(self._items)

    def __getitem__(self, index):
        return self._items[index]

    def __setitem__(self, index, value):
        self._items[index] = value

    def __delitem__(self, index):
        del self._items[index]
```

11.8小节还有一个在远程方法调用环境中使用代理的例子。

8.16 在类中定义多个构造器¶

问题¶

你想实现一个类，除了使用 `__init__()` 方法外，还有其他方式可以初始化它。

解决方案¶

为了实现多个构造器，你需要使用到类方法。例如：

```
import time

class Date:
    """方法一：使用类方法"""
    # Primary constructor
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    # Alternate constructor
    @classmethod
    def today(cls):
        t = time.localtime()
        return cls(t.tm_year, t.tm_mon, t.tm_mday)
```

直接调用类方法即可，下面是使用示例：

```
a = Date(2012, 12, 21) # Primary
b = Date.today() # Alternate
```

讨论¶

类方法的一个主要用途就是定义多个构造器。它接受一个 `class` 作为第一个参数(cls)。 你应该注意到了这个类被用来创建并返回最终的实例。在继承时也能工作的很好：

```
class NewDate(Date):
    pass

c = Date.today() # Creates an instance of Date (cls=Date)
d = NewDate.today() # Creates an instance of NewDate (cls=NewDate)
```

8.17 创建不调用init方法的实例¶

问题¶

你想创建一个实例，但是希望绕过执行 `__init__()` 方法。

解决方案¶

可以通过 `__new__()` 方法创建一个未初始化的实例。例如考虑如下这个类：

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

下面演示如何不调用 `__init__()` 方法来创建这个Date实例：

```
>>> d = Date.__new__(Date)
>>> d
<__main__.Date object at 0x1006716d0>
>>> d.year
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: 'Date' object has no attribute 'year'
>>>
```

结果可以看到，这个Date实例的属性year还不存在，所以你需要手动初始化：

```
>>> data = {'year':2012, 'month':8, 'day':29}
>>> for key, value in data.items():
...     setattr(d, key, value)
...
>>> d.year
2012
>>> d.month
8
>>>
```

讨论¶

当我们在反序列对象或者实现某个类方法构造函数时需要绕过 `__init__()` 方法来创建对象。 例如，对于上面的Date来讲，有时候你可能会像下面这样定义一个新的构造函数 `today()` ：

```
from time import localtime

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        d = cls.__new__(cls)
        t = localtime()
        d.year = t.tm_year
        d.month = t.tm_mon
        d.day = t.tm_mday
        return d
```

同样，在你反序列化JSON数据时产生一个如下的字典对象：

```
data = { 'year': 2012, 'month': 8, 'day': 29 }
```

如果你想将它转换成一个Date类型实例，可以使用上面的技术。

当你通过这种非常规方式来创建实例的时候，最好不要直接去访问底层实例字典，除非你真的清楚所有细节。 否则的话，如果这个类使用了 `__slots__` 、properties 、descriptors 或其他高级技术的时候代码就会失效。 而这时候使用 `setattr()` 方法会让你的代码变得更加通用。

8.18 利用Mixins扩展类功能¶

问题¶

你有很多有用的方法，想使用它们来扩展其他类的功能。但是这些类并没有任何继承的关系。 因此你不能简单的将这些方法放入一个基类，然后被其他类继承。

解决方案¶

通常当你想自定义类的时候会碰上这些问题。可能是某个库提供了一些基础类， 你可以利用它们来构造你自己的类。

假设你想扩展映射对象，给它们添加日志、唯一性设置、类型检查等等功能。下面是一些混入类：

```
class LoggedMappingMixin:
    """
    Add logging to get/set/delete operations for debugging.
    """
    __slots__ = ()  # 混入类都没有实例变量，因为直接实例化混入类没有任何意义

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return super().__getitem__(key)

    def __setitem__(self, key, value):
        print('Setting {} = {!r}'.format(key, value))
        return super().__setitem__(key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return super().__delitem__(key)


class SetOnceMappingMixin:
    '''
    Only allow a key to be set once.
    '''
    __slots__ = ()

    def __setitem__(self, key, value):
        if key in self:
            raise KeyError(str(key) + ' already set')
        return super().__setitem__(key, value)


class StringKeysMappingMixin:
    '''
    Restrict keys to strings only
    '''
    __slots__ = ()

    def __setitem__(self, key, value):
        if not isinstance(key, str):
            raise TypeError('keys must be strings')
        return super().__setitem__(key, value)
```

这些类单独使用起来没有任何意义，事实上如果你去实例化任何一个类，除了产生异常外没任何作用。 它们是用来通过多继承来和其他映射对象混入使用的。例如：

```
class LoggedDict(LoggedMappingMixin, dict):
    pass

d = LoggedDict()
d['x'] = 23
print(d['x'])
del d['x']
```

```
from collections import defaultdict

class SetOnceDefaultDict(SetOnceMappingMixin, defaultdict):
    pass


d = SetOnceDefaultDict(list)
d['x'].append(2)
d['x'].append(3)
# d['x'] = 23  # KeyError: 'x already set'
```

这个例子中，可以看到混入类跟其他已存在的类(比如dict、defaultdict和OrderedDict)结合起来使用，一个接一个。 结合后就能发挥正常功效了。

讨论¶

混入类在标准库中很多地方都出现过，通常都是用来像上面那样扩展某些类的功能。 它们也是多继承的一个主要用途。比如，当你编写网络代码时候， 你会经常使用 `socketserver` 模块中的 `ThreadingMixIn` 来给其他网络相关类增加多线程支持。 例如，下面是一个多线程的XML-RPC服务：

```
from xmlrpc.server import SimpleXMLRPCServer
from socketserver import ThreadingMixIn
class ThreadedXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
    pass
```

同时在一些大型库和框架中也会发现混入类的使用，用途同样是增强已存在的类的功能和一些可选特征。

对于混入类，有几点需要记住。首先是，混入类不能直接被实例化使用。 其次，混入类没有自己的状态信息，也就是说它们并没有定义 `__init__()` 方法，并且没有实例属性。 这也是为什么我们在上面明确定义了 `__slots__ = ()` 。

还有一种实现混入类的方式就是使用类装饰器，如下所示：

```
def LoggedMapping(cls):
    """第二种方式：使用类装饰器"""
    cls_getitem = cls.__getitem__
    cls_setitem = cls.__setitem__
    cls_delitem = cls.__delitem__

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return cls_getitem(self, key)

    def __setitem__(self, key, value):
        print('Setting {} = {!r}'.format(key, value))
        return cls_setitem(self, key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return cls_delitem(self, key)

    cls.__getitem__ = __getitem__
    cls.__setitem__ = __setitem__
    cls.__delitem__ = __delitem__
    return cls


@LoggedMapping
class LoggedDict(dict):
    pass
```

这个效果跟之前的是一样的，而且不再需要使用多继承了。参考9.12小节获取更多类装饰器的信息， 参考8.13小节查看更多混入类和类装饰器的例子。

8.19 实现状态对象或者状态机¶

问题¶

你想实现一个状态机或者是在不同状态下执行操作的对象，但是又不想在代码中出现太多的条件判断语句。

解决方案¶

在很多程序中，有些对象会根据状态的不同来执行不同的操作。比如考虑如下的一个连接对象：

```
class Connection:
    """普通方案，好多个判断语句，效率低下~~"""

    def __init__(self):
        self.state = 'CLOSED'

    def read(self):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('reading')

    def write(self, data):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('writing')

    def open(self):
        if self.state == 'OPEN':
            raise RuntimeError('Already open')
        self.state = 'OPEN'

    def close(self):
        if self.state == 'CLOSED':
            raise RuntimeError('Already closed')
        self.state = 'CLOSED'
```

这样写有很多缺点，首先是代码太复杂了，好多的条件判断。其次是执行效率变低， 因为一些常见的操作比如read()、write()每次执行前都需要执行检查。

一个更好的办法是为每个状态定义一个对象：

```
class Connection1:
    """新方案—对每个状态定义一个类"""

    def __init__(self):
        self.new_state(ClosedConnectionState)

    def new_state(self, newstate):
        self._state = newstate
        # Delegate to the state class

    def read(self):
        return self._state.read(self)

    def write(self, data):
        return self._state.write(self, data)

    def open(self):
        return self._state.open(self)

    def close(self):
        return self._state.close(self)


# Connection state base class
class ConnectionState:
```

```
    @staticmethod
    def read(conn):
        raise NotImplementedError()

    @staticmethod
    def write(conn, data):
        raise NotImplementedError()

    @staticmethod
    def open(conn):
        raise NotImplementedError()

    @staticmethod
    def close(conn):
        raise NotImplementedError()


# Implementation of different states
class ClosedConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        raise RuntimeError('Not open')

    @staticmethod
    def write(conn, data):
        raise RuntimeError('Not open')

    @staticmethod
    def open(conn):
        conn.new_state(OpenConnectionState)

    @staticmethod
    def close(conn):
        raise RuntimeError('Already closed')


class OpenConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        print('reading')

    @staticmethod
    def write(conn, data):
        print('writing')

    @staticmethod
    def open(conn):
        raise RuntimeError('Already open')

    @staticmethod
    def close(conn):
        conn.new_state(ClosedConnectionState)
```

下面是使用演示：

```
>>> c = Connection()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>> c.read()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 10, in read
        return self._state.read(self)
    File "example.py", line 43, in read
        raise RuntimeError('Not open')
RuntimeError: Not open
>>> c.open()
>>> c._state
<class '__main__.OpenConnectionState'>
>>> c.read()
reading
```

```
>>> c.write('hello')
writing
>>> c.close()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>>
```

讨论¶

如果代码中出现太多的条件判断语句的话，代码就会变得难以维护和阅读。 这里的解决方案是将每个状态抽取出来定义成一个类。

这里看上去有点奇怪，每个状态对象都只有静态方法，并没有存储任何的实例属性数据。 实际上，所有状态信息都只存储在 `Connection` 实例中。 在基类中定义的 `NotImplementedError` 是为了确保子类实现了相应的方法。 这里你或许还想使用8.12小节讲解的抽象基类方式。

设计模式中有一种模式叫状态模式，这一小节算是一个初步入门！

8.2 自定义字符串的格式化¶

问题¶

你想通过 `format()` 函数和字符串方法使得一个对象能支持自定义的格式化。

解决方案¶

为了自定义字符串的格式化，我们需要在类上面定义 `__format__()` 方法。例如：

```
_formats = {
    'ymd' : '{d.year}-{d.month}-{d.day}',
    'mdy' : '{d.month}/{d.day}/{d.year}',
    'dmy' : '{d.day}/{d.month}/{d.year}'
    }

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __format__(self, code):
        if code == '':
            code = 'ymd'
        fmt = _formats[code]
        return fmt.format(d=self)
```

现在 `Date` 类的实例可以支持格式化操作了，如同下面这样：

```
>>> d = Date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, 'mdy')
'12/21/2012'
>>> 'The date is {:ymd}'.format(d)
'The date is 2012-12-21'
>>> 'The date is {:mdy}'.format(d)
'The date is 12/21/2012'
>>>
```

讨论¶

`__format__()` 方法给Python的字符串格式化功能提供了一个钩子。 这里需要着重强调的是格式化代码的解析工作完全由类自己决定。因此，格式化代码可以是任何值。 例如，参考下面来自 `datetime` 模块中的代码：

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d,'%A, %B %d, %Y')
'Friday, December 21, 2012'
>>> 'The end is {:%d %b %Y}. Goodbye'.format(d)
'The end is 21 Dec 2012. Goodbye'
>>>
```

对于内置类型的格式化有一些标准的约定。 可以参考 string模块文档 说明。

8.20 通过字符串调用对象方法¶

问题¶

你有一个字符串形式的方法名称，想通过它调用某个对象的对应方法。

解决方案¶

最简单的情况，可以使用 `getattr()` ：

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Point({!r:},{!r:})'.format(self.x, self.y)

    def distance(self, x, y):
        return math.hypot(self.x - x, self.y - y)


p = Point(2, 3)
d = getattr(p, 'distance')(0, 0)  # Calls p.distance(0, 0)
```

另外一种方法是使用 `operator.methodcaller()` ，例如：

```
import operator
operator.methodcaller('distance', 0, 0)(p)
```

当你需要通过相同的参数多次调用某个方法时，使用 `operator.methodcaller` 就很方便了。 比如你需要排序一系列的点，就可以这样做：

```
points = [
    Point(1, 2),
    Point(3, 0),
    Point(10, -3),
    Point(-5, -7),
    Point(-1, 8),
    Point(3, 2)
]
# Sort by distance from origin (0, 0)
points.sort(key=operator.methodcaller('distance', 0, 0))
```

讨论¶

调用一个方法实际上是两步独立操作，第一步是查找属性，第二步是函数调用。 因此，为了调用某个方法，你可以首先通过 `getattr()` 来查找到这个属性，然后再去以函数方式调用它即可。

`operator.methodcaller()` 创建一个可调用对象，并同时提供所有必要参数， 然后调用的时候只需要将实例对象传递给它即可，比如：

```
>>> p = Point(3, 4)
>>> d = operator.methodcaller('distance', 0, 0)
>>> d(p)
5.0
>>>
```

通过方法名称字符串来调用方法通常出现在需要模拟 `case` 语句或实现访问者模式的时候。 参考下一小节获取更多高级例子。

8.21 实现访问者模式¶

问题¶

你要处理由大量不同类型的对象组成的复杂数据结构，每一个对象都需要进行不同的处理。 比如，遍历一个树形结构，然后根据每个节点的相应状态执行不同的操作。

解决方案¶

这里遇到的问题在编程领域中是很普遍的，有时候会构建一个由大量不同对象组成的数据结构。 假设你要写一个表示数学表达式的程序，那么你可能需要定义如下的类：

```
class Node:
    pass

class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value
```

然后利用这些类构建嵌套数据结构，如下所示：

```
# Representation of 1 + 2 * (3 - 4) / 5
t1 = Sub(Number(3), Number(4))
t2 = Mul(Number(2), t1)
t3 = Div(t2, Number(5))
t4 = Add(Number(1), t3)
```

这样做的问题是对于每个表达式，每次都要重新定义一遍，有没有一种更通用的方式让它支持所有的数字和操作符呢。 这里我们使用访问者模式可以达到这样的目的：

```
class NodeVisitor:
    def visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))
```

为了使用这个类，可以定义一个类继承它并且实现各种 `visit_Name()` 方法，其中Name是node类型。 例如，如果你想求表达式的值，可以这样写：

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -node.operand
```

使用示例：

```
>>> e = Evaluator()
>>> e.visit(t4)
0.6
>>>
```

作为一个不同的例子，下面定义一个类在一个栈上面将一个表达式转换成多个操作序列：

```
class StackCode(NodeVisitor):
    def generate_code(self, node):
        self.instructions = []
        self.visit(node)
        return self.instructions

    def visit_Number(self, node):
        self.instructions.append(('PUSH', node.value))

    def binop(self, node, instruction):
        self.visit(node.left)
        self.visit(node.right)
        self.instructions.append((instruction,))

    def visit_Add(self, node):
        self.binop(node, 'ADD')

    def visit_Sub(self, node):
        self.binop(node, 'SUB')

    def visit_Mul(self, node):
        self.binop(node, 'MUL')

    def visit_Div(self, node):
        self.binop(node, 'DIV')

    def unaryop(self, node, instruction):
        self.visit(node.operand)
        self.instructions.append((instruction,))

    def visit_Negate(self, node):
        self.unaryop(node, 'NEG')
```

使用示例：

```
>>> s = StackCode()
>>> s.generate_code(t4)
[('PUSH', 1), ('PUSH', 2), ('PUSH', 3), ('PUSH', 4), ('SUB',),
('MUL',), ('PUSH', 5), ('DIV',), ('ADD',)]
```

```
>>>
```

讨论¶

刚开始的时候你可能会写大量的if/else语句来实现， 这里访问者模式的好处就是通过 `getattr()` 来获取相应的方法，并利用递归来遍历所有的节点：

```
def binop(self, node, instruction):
    self.visit(node.left)
    self.visit(node.right)
    self.instructions.append((instruction,))
```

还有一点需要指出的是，这种技术也是实现其他语言中switch或case语句的方式。 比如，如果你正在写一个HTTP框架，你可能会写这样一个请求分发的控制器：

```
class HTTPHandler:
    def handle(self, request):
        methname = 'do_' + request.request_method
        getattr(self, methname)(request)
    def do_GET(self, request):
        pass
    def do_POST(self, request):
        pass
    def do_HEAD(self, request):
        pass
```

访问者模式一个缺点就是它严重依赖递归，如果数据结构嵌套层次太深可能会有问题， 有时候会超过Python的递归深度限制(参考 `sys.getrecursionlimit()`)。

可以参照8.22小节，利用生成器或迭代器来实现非递归遍历算法。

在跟解析和编译相关的编程中使用访问者模式是非常常见的。 Python本身的 `ast` 模块值得关注下，可以去看看源码。9.24小节演示了一个利用 `ast` 模块来处理Python源代码的例子。

8.22 不用递归实现访问者模式¶

问题¶

你使用访问者模式遍历一个很深的嵌套树形数据结构，并且因为超过嵌套层级限制而失败。 你想消除递归，并同时保持访问者编程模式。

解决方案¶

通过巧妙的使用生成器可以在树遍历或搜索算法中消除递归。 在8.21小节中，我们给出了一个访问者类。 下面我们利用一个栈和生成器重新实现这个类：

```
import types

class Node:
    pass

class NodeVisitor:
    def visit(self, node):
        stack = [node]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Node):
                    stack.append(self._visit(stack.pop()))
                else:
                    last_result = stack.pop()
            except StopIteration:
                stack.pop()

        return last_result

    def _visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))
```

如果你使用这个类，也能达到相同的效果。事实上你完全可以将它作为上一节中的访问者模式的替代实现。 考虑如下代码，遍历一个表达式的树：

```
class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass
```

```
class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value

# A sample visitor class that evaluates expressions
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -self.visit(node.operand)

if __name__ == '__main__':
    # 1 + 2*(3-4) / 5
    t1 = Sub(Number(3), Number(4))
    t2 = Mul(Number(2), t1)
    t3 = Div(t2, Number(5))
    t4 = Add(Number(1), t3)
    # Evaluate it
    e = Evaluator()
    print(e.visit(t4))  # Outputs 0.6
```

如果嵌套层次太深那么上述的Evaluator就会失效：

```
>>> a = Number(0)
>>> for n in range(1, 100000):
... a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
Traceback (most recent call last):
...
    File "visitor.py", line 29, in _visit
return meth(node)
    File "visitor.py", line 67, in visit_Add
return self.visit(node.left) + self.visit(node.right)
RuntimeError: maximum recursion depth exceeded
>>>
```

现在我们稍微修改下上面的Evaluator：

```
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        yield (yield node.left) + (yield node.right)

    def visit_Sub(self, node):
        yield (yield node.left) - (yield node.right)
```

```
    def visit_Mul(self, node):
        yield (yield node.left) * (yield node.right)

    def visit_Div(self, node):
        yield (yield node.left) / (yield node.right)

    def visit_Negate(self, node):
        yield - (yield node.operand)
```

再次运行，就不会报错了：

```
>>> a = Number(0)
>>> for n in range(1,100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
4999950000
>>>
```

如果你还想添加其他自定义逻辑也没问题：

```
class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        print('Add:', node)
        lhs = yield node.left
        print('left=', lhs)
        rhs = yield node.right
        print('right=', rhs)
        yield lhs + rhs
    ...
```

下面是简单的测试：

```
>>> e = Evaluator()
>>> e.visit(t4)
Add: <__main__.Add object at 0x1006a8d90>
left= 1
right= -0.4
0.6
>>>
```

讨论¶

这一小节我们演示了生成器和协程在程序控制流方面的强大功能。 避免递归的一个通常方法是使用一个栈或队列的数据结构。 例如，深度优先的遍历算法，第一次碰到一个节点时将其压入栈中，处理完后弹出栈。`visit()` 方法的核心思路就是这样。

另外一个需要理解的就是生成器中yield语句。当碰到yield语句时，生成器会返回一个数据并暂时挂起。 上面的例子使用这个技术来代替了递归。例如，之前我们是这样写递归：

```
value = self.visit(node.left)
```

现在换成yield语句：

```
value = yield node.left
```

它会将 `node.left` 返回给 `visit()` 方法，然后 `visit()` 方法调用那个节点相应的 `visit_Name()` 方法。 yield暂时将程序控制器让出给调用者，当执行完后，结果会赋值给value，

看完这一小节，你也许想去寻找其它没有yield语句的方案。但是这么做没有必要，你必须处理很多棘手的问题。 例如，为了消除递归，你必须要维护一个栈结构，如果不使用生成器，代码会变得很臃肿，到处都是栈操作语句、回调函数等。 实际上，使用yield语句可以让你写出非常漂亮的代码，它消除了递归但是看上去又很像递归实现，代码很简洁。

8.23 循环引用数据结构的内存管理¶

问题¶

你的程序创建了很多循环引用数据结构(比如树、图、观察者模式等)，你碰到了内存管理难题。

解决方案¶

一个简单的循环引用数据结构例子就是一个树形结构，双亲节点有指针指向孩子节点，孩子节点又返回来指向双亲节点。 这种情况下，可以考虑使用 `weakref` 库中的弱引用。例如：

```
import weakref

class Node:
    def __init__(self, value):
        self.value = value
        self._parent = None
        self.children = []

    def __repr__(self):
        return 'Node({!r:})'.format(self.value)

    # property that manages the parent as a weak-reference
    @property
    def parent(self):
        return None if self._parent is None else self._parent()

    @parent.setter
    def parent(self, node):
        self._parent = weakref.ref(node)

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

这种是想方式允许parent静默终止。例如：

```
>>> root = Node('parent')
>>> c1 = Node('child')
>>> root.add_child(c1)
>>> print(c1.parent)
Node('parent')
>>> del root
>>> print(c1.parent)
None
>>>
```

讨论¶

循环引用的数据结构在Python中是一个很棘手的问题，因为正常的垃圾回收机制不能适用于这种情形。 例如考虑如下代码：

```
# Class just to illustrate when deletion occurs
class Data:
    def __del__(self):
        print('Data.__del__')

# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
```

```
        self.children.append(child)
        child.parent = self
```

下面我们使用这个代码来做一些垃圾回收试验：

```
>>> a = Data()
>>> del a # Immediately deleted
Data.__del__
>>> a = Node()
>>> del a # Immediately deleted
Data.__del__
>>> a = Node()
>>> a.add_child(Node())
>>> del a # Not deleted (no message)
>>>
```

可以看到，最后一个的删除时打印语句没有出现。原因是Python的垃圾回收机制是基于简单的引用计数。 当一个对象的引用数变成0的时候才会立即删除掉。而对于循环引用这个条件永远不会成立。 因此，在上面例子中最后部分，父节点和孩子节点互相拥有对方的引用，导致每个对象的引用计数都不可能变成0。

Python有另外的垃圾回收器来专门针对循环引用的，但是你永远不知道它什么时候会触发。 另外你还可以手动的触发它，但是代码看上去很挫：

```
>>> import gc
>>> gc.collect() # Force collection
Data.__del__
Data.__del__
>>>
```

如果循环引用的对象自己还定义了自己的 `__del__()` 方法，那么会让情况变得更糟糕。 假设你像下面这样给Node定义自己的 `__del__()` 方法：

```
# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
        self.children.append(child)
        child.parent = self

    # NEVER DEFINE LIKE THIS.
    # Only here to illustrate pathological behavior
    def __del__(self):
        del self.data
        del.parent
        del.children
```

这种情况下，垃圾回收永远都不会去回收这个对象的，还会导致内存泄露。 如果你试着去运行它会发现，`Data.__del__` 消息永远不会出现了,甚至在你强制内存回收时：

```
>>> a = Node()
>>> a.add_child(Node()
>>> del a # No message (not collected)
>>> import gc
>>> gc.collect() # No message (not collected)
>>>
```

弱引用消除了引用循环的这个问题，本质来讲，弱引用就是一个对象指针，它不会增加它的引用计数。 你可以通过 `weakref` 来创建弱引用。例如：

```
>>> import weakref
>>> a = Node()
>>> a_ref = weakref.ref(a)
>>> a_ref
<weakref at 0x100581f70; to 'Node' at 0x1005c5410>
```

```
>>>
```

为了访问弱引用所引用的对象，你可以像函数一样去调用它即可。如果那个对象还存在就会返回它，否则就返回一个None。 由于原始对象的引用计数没有增加，那么就可以去删除它了。例如;

```
>>> print(a_ref())
<__main__.Node object at 0x1005c5410>
>>> del a
Data.__del__
>>> print(a_ref())
None
>>>
```

通过这里演示的弱引用技术，你会发现不再有循环引用问题了，一旦某个节点不被使用了，垃圾回收器立即回收它。你还能参考8.25小节关于弱引用的另外一个例子。

8.24 让类支持比较操作¶

问题¶

你想让某个类的实例支持标准的比较运算(比如>=,!=,<=,<等)，但是又不想去实现那一大丢的特殊方法。

解决方案¶

Python类对每个比较操作都需要实现一个特殊方法来支持。 例如为了支持>=操作符，你需要定义一个 `__ge__()` 方法。尽管定义一个方法没什么问题，但如果要你实现所有可能的比较方法那就有点烦人了。

装饰器 `functools.total_ordering` 就是用来简化这个处理的。 使用它来装饰一个类，你只需定义一个 `__eq__()` 方法， 外加其他方法(__lt__, __le__, __gt__, or __ge__)中的一个即可。 然后装饰器会自动为你填充其它比较方法。

作为例子，我们构建一些房子，然后给它们增加一些房间，最后通过房子大小来比较它们：

```
from functools import total_ordering

class Room:
    def __init__(self, name, length, width):
        self.name = name
        self.length = length
        self.width = width
        self.square_feet = self.length * self.width

@total_ordering
class House:
    def __init__(self, name, style):
        self.name = name
        self.style = style
        self.rooms = list()

    @property
    def living_space_footage(self):
        return sum(r.square_feet for r in self.rooms)

    def add_room(self, room):
        self.rooms.append(room)

    def __str__(self):
        return '{}: {} square foot {}'.format(self.name,
                self.living_space_footage,
                self.style)

    def __eq__(self, other):
        return self.living_space_footage == other.living_space_footage

    def __lt__(self, other):
        return self.living_space_footage < other.living_space_footage
```

这里我们只是给House类定义了两个方法：`__eq__()` 和 `__lt__()` ，它就能支持所有的比较操作：

```
# Build a few houses, and add rooms to them
h1 = House('h1', 'Cape')
h1.add_room(Room('Master Bedroom', 14, 21))
h1.add_room(Room('Living Room', 18, 20))
h1.add_room(Room('Kitchen', 12, 16))
h1.add_room(Room('Office', 12, 12))
h2 = House('h2', 'Ranch')
h2.add_room(Room('Master Bedroom', 14, 21))
h2.add_room(Room('Living Room', 18, 20))
h2.add_room(Room('Kitchen', 12, 16))
h3 = House('h3', 'Split')
h3.add_room(Room('Master Bedroom', 14, 21))
h3.add_room(Room('Living Room', 18, 20))
```

```
h3.add_room(Room('Office', 12, 16))
h3.add_room(Room('Kitchen', 15, 17))
houses = [h1, h2, h3]
print('Is h1 bigger than h2?', h1 > h2) # prints True
print('Is h2 smaller than h3?', h2 < h3) # prints True
print('Is h2 greater than or equal to h1?', h2 >= h1) # Prints False
print('Which one is biggest?', max(houses)) # Prints 'h3: 1101-square-foot Split'
print('Which is smallest?', min(houses)) # Prints 'h2: 846-square-foot Ranch'
```

讨论¶

其实 `total_ordering` 装饰器也没那么神秘。 它就是定义了一个从每个比较支持方法到所有需要定义的其他方法的一个映射而已。 比如你定义了 `__le__()` 方法，那么它就被用来构建所有其他的需要定义的那些特殊方法。 实际上就是在类里面像下面这样定义了一些特殊方法：

```
class House:
    def __eq__(self, other):
        pass
    def __lt__(self, other):
        pass
    # Methods created by @total_ordering
    __le__ = lambda self, other: self < other or self == other
    __gt__ = lambda self, other: not (self < other or self == other)
    __ge__ = lambda self, other: not (self < other)
    __ne__ = lambda self, other: not self == other
```

当然，你自己去写也很容易，但是使用 `@total_ordering` 可以简化代码，何乐而不为呢。

8.25 创建缓存实例¶

问题¶

在创建一个类的对象时，如果之前使用同样参数创建过这个对象， 你想返回它的缓存引用。

解决方案¶

这种通常是因为你希望相同参数创建的对象时单例的。 在很多库中都有实际的例子，比如 `logging` 模块，使用相同的名称创建的 `logger` 实例永远只有一个。例如：

```
>>> import logging
>>> a = logging.getLogger('foo')
>>> b = logging.getLogger('bar')
>>> a is b
False
>>> c = logging.getLogger('foo')
>>> a is c
True
>>>
```

为了达到这样的效果，你需要使用一个和类本身分开的工厂函数，例如：

```
# The class in question
class Spam:
    def __init__(self, name):
        self.name = name

# Caching support
import weakref
_spam_cache = weakref.WeakValueDictionary()
def get_spam(name):
    if name not in _spam_cache:
        s = Spam(name)
        _spam_cache[name] = s
    else:
        s = _spam_cache[name]
    return s
```

然后做一个测试，你会发现跟之前那个日志对象的创建行为是一致的：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> a is b
False
>>> c = get_spam('foo')
>>> a is c
True
>>>
```

讨论¶

编写一个工厂函数来修改普通的实例创建行为通常是一个比较简单的方法。 但是我们还能否找到更优雅的解决方案呢？

例如，你可能会考虑重新定义类的 `__new__()` 方法，就像下面这样：

```
# Note: This code doesn't quite work
import weakref

class Spam:
    _spam_cache = weakref.WeakValueDictionary()
    def __new__(cls, name):
        if name in cls._spam_cache:
```

```
            return cls._spam_cache[name]
        else:
            self = super().__new__(cls)
            cls._spam_cache[name] = self
            return self
    def __init__(self, name):
        print('Initializing Spam')
        self.name = name
```

初看起来好像可以达到预期效果，但是问题是 `__init__()` 每次都会被调用，不管这个实例是否被缓存了。例如：

```
>>> s = Spam('Dave')
Initializing Spam
>>> t = Spam('Dave')
Initializing Spam
>>> s is t
True
>>>
```

这个或许不是你想要的效果，因此这种方法并不可取。

上面我们使用到了弱引用计数，对于垃圾回收来讲是很有帮助的，关于这个我们在8.23小节已经讲过了。 当我们保持实例缓存时，你可能只想在程序中使用到它们时才保存。 一个 `WeakValueDictionary` 实例只会保存那些在其它地方还在被使用的实例。 否则的话，只要实例不再被使用了，它就从字典中被移除了。观察下下面的测试结果：

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> c = get_spam('foo')
>>> list(_spam_cache)
['foo', 'bar']
>>> del a
>>> del c
>>> list(_spam_cache)
['bar']
>>> del b
>>> list(_spam_cache)
[]
>>>
```

对于大部分程序而已，这里代码已经够用了。不过还是有一些更高级的实现值得了解下。

首先是这里使用到了一个全局变量，并且工厂函数跟类放在一块。我们可以通过将缓存代码放到一个单独的缓存管理器中：

```
import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()

    def get_spam(self, name):
        if name not in self._cache:
            s = Spam(name)
            self._cache[name] = s
        else:
            s = self._cache[name]
        return s

    def clear(self):
            self._cache.clear()

class Spam:
    manager = CachedSpamManager()
    def __init__(self, name):
        self.name = name

def get_spam(name):
    return Spam.manager.get_spam(name)
```

这样的话代码更清晰，并且也更灵活，我们可以增加更多的缓存管理机制，只需要替代manager即可。

还有一点就是，我们暴露了类的实例化给用户，用户很容易去直接实例化这个类，而不是使用工厂方法，如：

```
>>> a = Spam('foo')
>>> b = Spam('foo')
>>> a is b
False
>>>
```

有几种方式可以防止用户这样做，第一个是将类的名字修改为以下划线(_)开头，提示用户别直接调用它。 第二种就是让这个类的 `__init__()` 方法抛出一个异常，让它不能被初始化：

```
class Spam:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
```

然后修改缓存管理器代码，使用 `Spam._new()` 来创建实例，而不是直接调用 `Spam()` 构造函数：

```
# ------------------------最后的修正方案------------------------
class CachedSpamManager2:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()

    def get_spam(self, name):
        if name not in self._cache:
            temp = Spam3._new(name)  # Modified creation
            self._cache[name] = temp
        else:
            temp = self._cache[name]
        return temp

    def clear(self):
            self._cache.clear()

class Spam3:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
        return self
```

最后这样的方案就已经足够好了。 缓存和其他构造模式还可以使用9.13小节中的元类实现的更优雅一点(使用了更高级的技术)。

8.3 让对象支持上下文管理协议¶

问题¶

你想让你的对象支持上下文管理协议(with语句)。

解决方案¶

为了让一个对象兼容 `with` 语句，你需要实现 `__enter__()` 和 `__exit__()` 方法。 例如，考虑如下的一个类，它能为我们创建一个网络连接：

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = family
        self.type = type
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

这个类的关键特点在于它表示了一个网络连接，但是初始化的时候并不会做任何事情(比如它并没有建立一个连接)。 连接的建立和关闭是使用 `with` 语句自动完成的，例如：

```
from functools import partial

conn = LazyConnection(('www.python.org', 80))
# Connection closed
with conn as s:
    # conn.__enter__() executes: connection open
    s.send(b'GET /index.html HTTP/1.0\r\n')
    s.send(b'Host: www.python.org\r\n')
    s.send(b'\r\n')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() executes: connection closed
```

讨论¶

编写上下文管理器的主要原理是你的代码会放到 `with` 语句块中执行。 当出现 `with` 语句的时候，对象的 `__enter__()` 方法被触发， 它返回的值(如果有的话)会被赋值给 `as` 声明的变量。然后， `with` 语句块里面的代码开始执行。 最后， `__exit__()` 方法被触发进行清理工作。

不管 `with` 代码块中发生什么，上面的控制流都会执行完，就算代码块中发生了异常也是一样的。 事实上， `__exit__()` 方法的三个参数包含了异常类型、异常值和追溯信息(如果有的话)。 `__exit__()` 方法能自己决定怎样利用这个异常信息，或者忽略它并返回一个None值。 如果 `__exit__()` 返回 `True` ，那么异常会被清空，就好像什么都没发生一样， `with` 语句后面的程序继续在正常执行。

还有一个细节问题就是 `LazyConnection` 类是否允许多个 `with` 语句来嵌套使用连接。 很显然，上面的定义中一次只能允许一个socket连接，如果正在使用一个socket的时候又重复使用 `with` 语句， 就会产生一个异常了。不过你可以像下面这样修改下上面的实现来解决这个问题：

```
from socket import socket, AF_INET, SOCK_STREAM
```

```
class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = family
        self.type = type
        self.connections = []

    def __enter__(self):
        sock = socket(self.family, self.type)
        sock.connect(self.address)
        self.connections.append(sock)
        return sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.connections.pop().close()

# Example use
from functools import partial

conn = LazyConnection(('www.python.org', 80))
with conn as s1:
    pass
    with conn as s2:
        pass
        # s1 and s2 are independent sockets
```

在第二个版本中，`LazyConnection` 类可以被看做是某个连接工厂。在内部，一个列表被用来构造一个栈。 每次 `__enter__()` 方法执行的时候，它复制创建一个新的连接并将其加入到栈里面。 `__exit__()` 方法简单的从栈中弹出最后一个连接并关闭它。 这里稍微有点难理解，不过它能允许嵌套使用 `with` 语句创建多个连接，就如上面演示的那样。

在需要管理一些资源比如文件、网络连接和锁的编程环境中，使用上下文管理器是很普遍的。 这些资源的一个主要特征是它们必须被手动的关闭或释放来确保程序的正确运行。 例如，如果你请求了一个锁，那么你必须确保之后释放了它，否则就可能产生死锁。 通过实现 `__enter__()` 和 `__exit__()` 方法并使用 `with` 语句可以很容易的避免这些问题，因为 `__exit__()` 方法可以让你无需担心这些了。

在 `contextmanager` 模块中有一个标准的上下文管理方案模板，可参考9.22小节。 同时在12.6小节中还有一个对本节示例程序的线程安全的修改版。

8.4 创建大量对象时节省内存方法¶

问题¶

你的程序要创建大量(可能上百万)的对象，导致占用很大的内存。

解决方案¶

对于主要是用来当成简单的数据结构的类而言，你可以通过给类添加 `__slots__` 属性来极大的减少实例所占的内存。比如：

```
class Date:
    __slots__ = ['year', 'month', 'day']
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

当你定义 `__slots__` 后，Python就会为实例使用一种更加紧凑的内部表示。 实例通过一个很小的固定大小的数组来构建，而不是为每个实例定义一个字典，这跟元组或列表很类似。 在 `__slots__` 中列出的属性名在内部被映射到这个数组的指定小标上。 使用slots一个不好的地方就是我们不能再给实例添加新的属性了，只能使用在 `__slots__` 中定义的那些属性名。

讨论¶

使用slots后节省的内存会跟存储属性的数量和类型有关。 不过，一般来讲，使用到的内存总量和将数据存储在一个元组中差不多。 为了给你一个直观认识，假设你不使用slots直接存储一个Date实例， 在64位的Python上面要占用428字节，而如果使用了slots，内存占用下降到156字节。 如果程序中需要同时创建大量的日期实例，那么这个就能极大的减小内存使用量了。

尽管slots看上去是一个很有用的特性，很多时候你还是得减少对它的使用冲动。 Python的很多特性都依赖于普通的基于字典的实现。 另外，定义了slots后的类不再支持一些普通类特性了，比如多继承。 大多数情况下，你应该只在那些经常被使用到的用作数据结构的类上定义slots (比如在程序中需要创建某个类的几百万个实例对象)。

关于 `__slots__` 的一个常见误区是它可以作为一个封装工具来防止用户给实例增加新的属性。 尽管使用slots可以达到这样的目的，但是这个并不是它的初衷。 `__slots__` 更多的是用来作为一个内存优化工具。

8.5 在类中封装属性名¶

问题¶

你想封装类的实例上面的“私有”数据，但是Python语言并没有访问控制。

解决方案¶

Python程序员不去依赖语言特性去封装数据，而是通过遵循一定的属性和方法命名规约来达到这个效果。 第一个约定是任何以单下划线_开头的名字都应该是内部实现。比如：

```
class A:
    def __init__(self):
        self._internal = 0 # An internal attribute
        self.public = 1 # A public attribute

    def public_method(self):
        '''
        A public method
        '''
        pass

    def _internal_method(self):
        pass
```

Python并不会真的阻止别人访问内部名称。但是如果你这么做肯定是不好的，可能会导致脆弱的代码。 同时还要注意到，使用下划线开头的约定同样适用于模块名和模块级别函数。 例如，如果你看到某个模块名以单下划线开头(比如_socket)，那它就是内部实现。 类似的，模块级别函数比如 `sys._getframe()` 在使用的时候就得加倍小心了。

你还可能会遇到在类定义中使用两个下划线(__)开头的命名。比如：

```
class B:
    def __init__(self):
        self.__private = 0

    def __private_method(self):
        pass

    def public_method(self):
        pass
        self.__private_method()
```

使用双下划线开始会导致访问名称变成其他形式。 比如，在前面的类B中，私有属性会被分别重命名为 `_B__private` 和 `_B__private_method` 。 这时候你可能会问这样重命名的目的是什么，答案就是继承——这种属性通过继承是无法被覆盖的。比如：

```
class C(B):
    def __init__(self):
        super().__init__()
        self.__private = 1 # Does not override B.__private

    # Does not override B.__private_method()
    def __private_method(self):
        pass
```

这里，私有名称 `__private` 和 `__private_method` 被重命名为 `_C__private` 和 `_C__private_method` ，这个跟父类B中的名称是完全不同的。

讨论¶

上面提到有两种不同的编码约定(单下划线和双下划线)来命名私有属性，那么问题就来了：到底哪种方式好呢？ 大多数而言，你应该让你的非公共名称以单下划线开头。但是，如果你清楚你的代码会涉及到子类， 并且有些内部属性应

该在子类中隐藏起来，那么才考虑使用双下划线方案。

还有一点要注意的是，有时候你定义的一个变量和某个保留关键字冲突，这时候可以使用单下划线作为后缀，例如：

```
lambda_ = 2.0 # Trailing _ to avoid clash with lambda keyword
```

这里我们并不使用单下划线前缀的原因是它避免误解它的使用初衷 (如使用单下划线前缀的目的是为了防止命名冲突而不是指明这个属性是私有的)。 通过使用单下划线后缀可以解决这个问题。

8.6 创建可管理的属性¶

问题¶

你想给某个实例attribute增加除访问与修改之外的其他处理逻辑，比如类型检查或合法性验证。

解决方案¶

自定义某个属性的一种简单方法是将它定义为一个property。 例如，下面的代码定义了一个property，增加对一个属性简单的类型检查：

```
class Person:
    def __init__(self, first_name):
        self._first_name = first_name

    # Getter function
    @property
    def first_name(self):
        return self._first_name

    # Setter function
    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Deleter function (optional)
    @first_name.deleter
    def first_name(self):
        raise AttributeError("Can't delete attribute")
```

上述代码中有三个相关联的方法，这三个方法的名字都必须一样。 第一个方法是一个 `getter` 函数，它使得 `first_name` 成为一个属性。 其他两个方法给 `first_name` 属性添加了 `setter` 和 `deleter` 函数。 需要强调的是只有在 `first_name` 属性被创建后， 后面的两个装饰器 `@first_name.setter` 和 `@first_name.deleter` 才能被定义。

property的一个关键特征是它看上去跟普通的attribute没什么两样， 但是访问它的时候会自动触发 `getter` 、 `setter` 和 `deleter` 方法。例如：

```
>>> a = Person('Guido')
>>> a.first_name # Calls the getter
'Guido'
>>> a.first_name = 42 # Calls the setter
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "prop.py", line 14, in first_name
        raise TypeError('Expected a string')
TypeError: Expected a string
>>> del a.first_name
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: can`t delete attribute
>>>
```

在实现一个property的时候，底层数据(如果有的话)仍然需要存储在某个地方。 因此，在get和set方法中，你会看到对 `_first_name` 属性的操作，这也是实际数据保存的地方。 另外，你可能还会问为什么 `__init__()` 方法中设置了 `self.first_name` 而不是 `self._first_name` 。 在这个例子中，我们创建一个property的目的就是在设置attribute的时候进行检查。 因此，你可能想在初始化的时候也进行这种类型检查。通过设置 `self.first_name` ，自动调用 `setter` 方法， 这个方法里面会进行参数的检查，否则就是直接访问 `self._first_name` 了。

还能在已存在的get和set方法基础上定义property。例如：

```
class Person:
    def __init__(self, first_name):
```

```
        self.set_first_name(first_name)

    # Getter function
    def get_first_name(self):
        return self._first_name

    # Setter function
    def set_first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Deleter function (optional)
    def del_first_name(self):
        raise AttributeError("Can't delete attribute")

    # Make a property from existing get/set methods
    name = property(get_first_name, set_first_name, del_first_name)
```

讨论¶

一个property属性其实就是一系列相关绑定方法的集合。如果你去查看拥有property的类， 就会发现property本身的fget、fset和fdel属性就是类里面的普通方法。比如：

```
>>> Person.first_name.fget
<function Person.first_name at 0x1006a60e0>
>>> Person.first_name.fset
<function Person.first_name at 0x1006a6170>
>>> Person.first_name.fdel
<function Person.first_name at 0x1006a62e0>
>>>
```

通常来讲，你不会直接取调用fget或者fset，它们会在访问property的时候自动被触发。

只有当你确实需要对attribute执行其他额外的操作的时候才应该使用到property。 有时候一些从其他编程语言(比如Java)过来的程序员总认为所有访问都应该通过getter和setter， 所以他们认为代码应该像下面这样写：

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        self._first_name = value
```

不要写这种没有做任何其他额外操作的property。 首先，它会让你的代码变得很臃肿，并且还会迷惑阅读者。 其次，它还会让你的程序运行起来变慢很多。 最后，这样的设计并没有带来任何的好处。 特别是当你以后想给普通attribute访问添加额外的处理逻辑的时候， 你可以将它变成一个property而无需改变原来的代码。 因为访问attribute的代码还是保持原样。

Properties还是一种定义动态计算attribute的方法。 这种类型的attributes并不会被实际的存储，而是在需要的时候计算出来。比如：

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return math.pi * self.radius ** 2

    @property
```

```
    def diameter(self):
        return self.radius * 2

    @property
    def perimeter(self):
        return 2 * math.pi * self.radius
```

在这里，我们通过使用properties，将所有的访问接口形式统一起来， 对半径、直径、周长和面积的访问都是通过属性访问，就跟访问简单的attribute是一样的。 如果不这样做的话，那么就要在代码中混合使用简单属性访问和方法调用。下面是使用的实例：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area  # Notice lack of ()
50.26548245743669
>>> c.perimeter  # Notice lack of ()
25.132741228718345
>>>
```

尽管properties可以实现优雅的编程接口，但有些时候你还是会想直接使用getter和setter函数。例如：

```
>>> p = Person('Guido')
>>> p.get_first_name()
'Guido'
>>> p.set_first_name('Larry')
>>>
```

这种情况的出现通常是因为Python代码被集成到一个大型基础平台架构或程序中。 例如，有可能是一个Python类准备加入到一个基于远程过程调用的大型分布式系统中。 这种情况下，直接使用get/set方法(普通方法调用)而不是property或许会更容易兼容。

最后一点，不要像下面这样写有大量重复代码的property定义：

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Repeated property code, but for a different name (bad!)
    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._last_name = value
```

重复代码会导致臃肿、易出错和丑陋的程序。好消息是，通过使用装饰器或闭包，有很多种更好的方法来完成同样的事情。 可以参考8.9和9.21小节的内容。

8.7 调用父类方法¶

问题¶

你想在子类中调用父类的某个已经被覆盖的方法。

解决方案¶

为了调用父类(超类)的一个方法，可以使用 `super()` 函数，比如：

```
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def spam(self):
        print('B.spam')
        super().spam()  # Call parent spam()
```

`super()` 函数的一个常见用法是在 `__init__()` 方法中确保父类被正确的初始化了：

```
class A:
    def __init__(self):
        self.x = 0

class B(A):
    def __init__(self):
        super().__init__()
        self.y = 1
```

`super()` 的另外一个常见用法出现在覆盖Python特殊方法的代码中，比如：

```
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value) # Call original __setattr__
        else:
            setattr(self._obj, name, value)
```

在上面代码中， `__setattr__()` 的实现包含一个名字检查。 如果某个属性名以下划线(_)开头，就通过 `super()` 调用原始的 `__setattr__()` ， 否则的话就委派给内部的代理对象 `self._obj` 去处理。 这看上去有点意思，因为就算没有显式的指明某个类的父类， `super()` 仍然可以有效的工作。

讨论¶

实际上，大家对于在Python中如何正确使用 `super()` 函数普遍知之甚少。 你有时候会看到像下面这样直接调用父类的一个方法：

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')
```

尽管对于大部分代码而言这么做没什么问题，但是在更复杂的涉及到多继承的代码中就有可能导致很奇怪的问题发生。 比如，考虑如下的情况：

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')
```

如果你运行这段代码就会发现 `Base.__init__()` 被调用两次，如下所示：

```
>>> c = C()
Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__
>>>
```

可能两次调用 `Base.__init__()` 没什么坏处，但有时候却不是。 另一方面，假设你在代码中换成使用 `super()` ，结果就很完美了：

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A,B):
    def __init__(self):
        super().__init__()  # Only one call to super() here
        print('C.__init__')
```

运行这个新版本后，你会发现每个 `__init__()` 方法只会被调用一次了：

```
>>> c = C()
Base.__init__
B.__init__
A.__init__
C.__init__
>>>
```

为了弄清它的原理，我们需要花点时间解释下Python是如何实现继承的。 对于你定义的每一个类，Python会计算出一个所谓的方法解析顺序(MRO)列表。 这个MRO列表就是一个简单的所有基类的线性顺序表。例如：

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
```

```
<class '__main__.Base'>, <class 'object'>)
>>>
```

为了实现继承，Python会在MRO列表上从左到右开始查找基类，直到找到第一个匹配这个属性的类为止。

而这个MRO列表的构造是通过一个C3线性化算法来实现的。 我们不去深究这个算法的数学原理，它实际上就是合并所有父类的MRO列表并遵循如下三条准则：

- 子类会先于父类被检查
- 多个父类会根据它们在列表中的顺序被检查
- 如果对下一个类存在两个合法的选择，选择第一个父类

老实说，你所要知道的就是MRO列表中的类顺序会让你定义的任意类层级关系变得有意义。

当你使用 `super()` 函数时，Python会在MRO列表上继续搜索下一个类。 只要每个重定义的方法统一使用 `super()` 并只调用它一次， 那么控制流最终会遍历完整个MRO列表，每个方法也只会被调用一次。 这也是为什么在第二个例子中你不会调用两次 `Base.__init__()` 的原因。

`super()` 有个令人吃惊的地方是它并不一定去查找某个类在MRO中下一个直接父类， 你甚至可以在一个没有直接父类的类中使用它。例如，考虑如下这个类：

```
class A:
    def spam(self):
        print('A.spam')
        super().spam()
```

如果你试着直接使用这个类就会出错：

```
>>> a = A()
>>> a.spam()
A.spam
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in spam
AttributeError: 'super' object has no attribute 'spam'
>>>
```

但是，如果你使用多继承的话看看会发生什么：

```
>>> class B:
...     def spam(self):
...         print('B.spam')
...
>>> class C(A,B):
...     pass
...
>>> c = C()
>>> c.spam()
A.spam
B.spam
>>>
```

你可以看到在类A中使用 `super().spam()` 实际上调用的是跟类A毫无关系的类B中的 `spam()` 方法。 这个用类C的MRO列表就可以完全解释清楚了：

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class 'object'>)
>>>
```

在定义混入类的时候这样使用 `super()` 是很普遍的。可以参考8.13和8.18小节。

然而，由于 `super()` 可能会调用不是你想要的方法，你应该遵循一些通用原则。 首先，确保在继承体系中所有相同名字的方法拥有可兼容的参数签名(比如相同的参数个数和参数名称)。 这样可以确保 `super()` 调用一个非直接父类方法时不会出错。 其次，最好确保最顶层的类提供了这个方法的实现，这样的话在MRO上面的查找链肯定可以找到某个确定的方法。

在Python社区中对于 `super()` 的使用有时候会引来一些争议。 尽管如此，如果一切顺利的话，你应该在你最新代码中使用它。 Raymond Hettinger为此写了一篇非常好的文章 “Python’s super() Considered Super!”， 通过大量的例子向我们解释了为什么 `super()` 是极好的。

8.8 子类中扩展property¶

问题¶

在子类中，你想要扩展定义在父类中的property的功能。

解决方案¶

考虑如下的代码，它定义了一个property：

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

    # Getter function
    @property
    def name(self):
        return self._name

    # Setter function
    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._name = value

    # Deleter function
    @name.deleter
    def name(self):
        raise AttributeError("Can't delete attribute")
```

下面是一个示例类，它继承自Person并扩展了 `name` 属性的功能：

```
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

接下来使用这个新类：

```
>>> s = SubPerson('Guido')
Setting name to Guido
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
Setting name to Larry
>>> s.name = 42
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 16, in name
        raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

如果你仅仅只想扩展property的某一个方法，那么可以像下面这样写：

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```

或者，你只想修改setter方法，就这么写：

```
class SubPerson(Person):
    @Person.name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)
```

讨论¶

在子类中扩展一个property可能会引起很多不易察觉的问题， 因为一个property其实是 `getter`、`setter` 和 `deleter` 方法的集合，而不是单个方法。 因此，当你扩展一个property的时候，你需要先确定你是否要重新定义所有的方法还是说只修改其中某一个。

在第一个例子中，所有的property方法都被重新定义。 在每一个方法中，使用了 `super()` 来调用父类的实现。 在 `setter` 函数中使用 `super(SubPerson, SubPerson).name.__set__(self, value)` 的语句是没有错的。 为了委托给之前定义的setter方法，需要将控制权传递给之前定义的name属性的 `__set__()` 方法。 不过，获取这个方法的唯一途径是使用类变量而不是实例变量来访问它。 这也是为什么我们要使用 `super(SubPerson, SubPerson)` 的原因。

如果你只想重定义其中一个方法，那只使用 @property 本身是不够的。比如，下面的代码就无法工作：

```
class SubPerson(Person):
    @property  # Doesn't work
    def name(self):
        print('Getting name')
        return super().name
```

如果你试着运行会发现setter函数整个消失了：

```
>>> s = SubPerson('Guido')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 5, in __init__
        self.name = name
AttributeError: can't set attribute
>>>
```

你应该像之前说过的那样修改代码：

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```

这么写后，property之前已经定义过的方法会被复制过来，而getter函数被替换。然后它就能按照期望的工作了：

```
>>> s = SubPerson('Guido')
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
>>> s.name
Getting name
'Larry'
>>> s.name = 42
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
```

```
    File "example.py", line 16, in name
        raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

在这个特别的解决方案中，我们没办法使用更加通用的方式去替换硬编码的 `Person` 类名。 如果你不知道到底是哪个基类定义了property， 那你只能通过重新定义所有property并使用 `super()` 来将控制权传递给前面的实现。

值得注意的是上面演示的第一种技术还可以被用来扩展一个描述器(在8.9小节我们有专门的介绍)。比如：

```
# A descriptor
class String:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        instance.__dict__[self.name] = value

# A class with a descriptor
class Person:
    name = String('name')

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

# Extending a descriptor with a property
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)
```

最后值得注意的是，读到这里时，你应该会发现子类化 `setter` 和 `deleter` 方法其实是很简单的。 这里演示的解决方案同样适用，但是在 Python的issue页面 报告的一个bug，或许会使得将来的Python版本中出现一个更加简洁的方法。

8.9 创建新的类或实例属性¶

问题¶

你想创建一个新的拥有一些额外功能的实例属性类型，比如类型检查。

解决方案¶

如果你想创建一个全新的实例属性，可以通过一个描述器类的形式来定义它的功能。下面是一个例子：

```
# Descriptor attribute for an integer type-checked attribute
class Integer:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]
```

一个描述器就是一个实现了三个核心的属性访问操作(get, set, delete)的类， 分别为 `__get__()` 、 `__set__()` 和 `__delete__()` 这三个特殊的方法。 这些方法接受一个实例作为输入，之后相应的操作实例底层的字典。

为了使用一个描述器，需将这个描述器的实例作为类属性放到一个类的定义中。例如：

```
class Point:
    x = Integer('x')
    y = Integer('y')

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

当你这样做后，所有对描述器属性(比如x或y)的访问会被 `__get__()` 、 `__set__()` 和 `__delete__()` 方法捕获到。例如：

```
>>> p = Point(2, 3)
>>> p.x # Calls Point.x.__get__(p,Point)
2
>>> p.y = 5 # Calls Point.y.__set__(p, 5)
>>> p.x = 2.3 # Calls Point.x.__set__(p, 2.3)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "descrip.py", line 12, in __set__
        raise TypeError('Expected an int')
TypeError: Expected an int
>>>
```

作为输入，描述器的每一个方法会接受一个操作实例。 为了实现请求操作，会相应的操作实例底层的字典(__dict__属性)。 描述器的 `self.name` 属性存储了在实例字典中被实际使用到的key。

讨论¶

描述器可实现大部分Python类特性中的底层魔法， 包括 `@classmethod` 、 `@staticmethod` 、 `@property` ，甚至是

`__slots__` 特性。

通过定义一个描述器，你可以在底层捕获核心的实例操作(get, set, delete)，并且可完全自定义它们的行为。 这是一个强大的工具，有了它你可以实现很多高级功能，并且它也是很多高级库和框架中的重要工具之一。

描述器的一个比较困惑的地方是它只能在类级别被定义，而不能为每个实例单独定义。因此，下面的代码是无法工作的：

```
# Does NOT work
class Point:
    def __init__(self, x, y):
        self.x = Integer('x') # No! Must be a class variable
        self.y = Integer('y')
        self.x = x
        self.y = y
```

同时，`__get__()` 方法实现起来比看上去要复杂得多：

```
# Descriptor attribute for an integer type-checked attribute
class Integer:

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
```

`__get__()` 看上去有点复杂的原因归结于实例变量和类变量的不同。 如果一个描述器被当做一个类变量来访问，那么 `instance` 参数被设置成 `None` 。 这种情况下，标准做法就是简单的返回这个描述器本身即可(尽管你还可以添加其他的自定义操作)。例如：

```
>>> p = Point(2,3)
>>> p.x # Calls Point.x.__get__(p, Point)
2
>>> Point.x # Calls Point.x.__get__(None, Point)
<__main__.Integer object at 0x100671890>
>>>
```

描述器通常是那些使用到装饰器或元类的大型框架中的一个组件。同时它们的使用也被隐藏在后面。 举个例子，下面是一些更高级的基于描述器的代码，并涉及到一个类装饰器：

```
# Descriptor for a type-checked attribute
class Typed:
    def __init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('Expected ' + str(self.expected_type))
        instance.__dict__[self.name] = value
    def __delete__(self, instance):
        del instance.__dict__[self.name]

# Class decorator that applies it to selected attributes
def typeassert(**kwargs):
    def decorate(cls):
        for name, expected_type in kwargs.items():
            # Attach a Typed descriptor to the class
            setattr(cls, name, Typed(name, expected_type))
        return cls
    return decorate
```

```
# Example use
@typeassert(name=str, shares=int, price=float)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

最后要指出的一点是，如果你只是想简单的自定义某个类的单个属性访问的话就不用去写描述器了。 这种情况下使用8.6小节介绍的property技术会更加容易。 当程序中有很多重复代码的时候描述器就很有用了 (比如你想在你代码的很多地方使用描述器提供的功能或者将它作为一个函数库特性)。

第八章：类与对象¶

本章主要关注点的是和类定义有关的常见编程模型。包括让对象支持常见的Python特性、特殊方法的使用、 类封装技术、继承、内存管理以及有用的设计模式。

Contents: