到一个基于字典的注册中。由于它返回对象本身而不是一个包装器，所以它没有拦截随后的调用：

```
# Registering decorated objects to an API

registry = {}
def register(obj):                      # Both class and func decorator
    registry[obj.__name__] = obj        # Add to registry
    return obj                          # Return obj itself, not a wrapper

@register
def spam(x):
    return(x ** 2)                      # spam = register(spam)

@register
def ham(x):
    return(x ** 3)

@register
class Eggs:                             # Eggs = register(Eggs)
    def __init__(self, x):
        self.data = x ** 4
    def __str__(self):
        return str(self.data)

print('Registry:')
for name in registry:
    print(name, '=>', registry[name], type(registry[name]))

print('\nManual calls:')
print(spam(2))                          # Invoke objects manually
print(ham(2))                           # Later calls not intercepted
X = Eggs(2)
print(X)

print('\nRegistry calls:')
for name in registry:
    print(name, '=>', registry[name](3)) # Invoke from registry
```

当这段代码运行的时候，装饰的对象按照名称添加到注册中，但当随后调用它们的时候，它们仍然按照最初的编码工作，而没有指向一个包装器层。实际上，我们的对象可以手动运行，或从注册表内部运行：

```
Registry:
Eggs => <class '__main__.Eggs'> <class 'type'>
ham => <function ham at 0x02CFB738> <class 'function'>
spam => <function spam at 0x02CFB6F0> <class 'function'>

Manual calls:
4
8
16

Registry calls:
```

```
Eggs => 81
ham => 27
spam => 9
```

例如，一个用户界面可能使用这样的技术，为用户动作注册回调处理程序。处理程序可能通过函数或类名来注册，就像这里所做的一样，或者可以使用装饰器参数来指定主体事件；包含装饰器的一条额外的def语句可能会用来保持这样的参数以便在装饰时使用。

这个例子是仿造的，但是，其技术很通用。例如，函数装饰器也可能用来处理函数属性，并且类装饰器可能动态地插入新的类属性，或者甚至新的方法。考虑如下的函数装饰器——它们把函数属性分配给记录信息，以便随后供一个API使用，但是，它们没有插入一个包含器层来拦截随后的调用：

```
# Augmenting decorated objects directly

>>> def decorate(func):
...     func.marked = True              # Assign function attribute for later use
...     return func
...
>>> @decorate
... def spam(a, b):
...     return a + b
...
>>> spam.marked
True

>>> def annotate(text):                # Same, but value is decorator argument
...     def decorate(func):
...         func.label = text
...         return func
...     return decorate
...
>>> @annotate('spam data')
... def spam(a, b):                     # spam = annotate(...)(spam)
...     return a + b
...
>>> spam(1, 2), spam.label
(3, 'spam data')
```

这样的装饰器直接扩展了函数和类，没有捕捉对它们的随后调用。我们将在下一章见到更多的管理类的类装饰的例子，因为这证明了它已经转向了**元类**的领域；在本章剩余的部分，我们来看看使用装饰器的两个较大的案例。

示例：“私有”和“公有”属性

本章的最后两个小节介绍了使用装饰器的两个较大的例子。这两个例子都用尽量少的说明来展示，部分是由于本章的篇幅已经超出了限制，但主要是因为你应该已经很好地理

解了装饰器的基础知识，足够能够自行研究这些例子。作为通用用途的工具，这些例子使我们有机会来看看装饰器的概念如何融入到更为有用的代码中。

实现私有属性

如下的**类装饰器**实现了一个用于类实例属性的`Private`声明，也就是说，属性存储在一个实例上，或者从其一个类继承而来。不接受从装饰的类的**外部**对这样的属性的获取和修改访问，但是，仍然允许类自身在其方法中自由地访问那些名称。它不是具体的C++或Java，但它提供了类似的访问控制作为Python中的选项。

在第29章，我们见到了实例属性针对修改成为私有的不完整的、粗糙的实现。这里的版本扩展了这一概念以验证属性获取，并且它使用委托而不是继承来实现该模型。实际上，在某种意义上，这只是我们前面遇到的属性跟踪器类装饰器的一个扩展。

尽管这个例子利用了类装饰器的新语法糖来编写私有属性，但它的属性拦截最终仍然是基于我们在前面各章介绍的`__getattr__`和`__setattr__`运算符重载方法。当检测到访问一个私有属性的时候，这个版本使用`raise`语句引发一个异常，还有一条出错消息；异常可能在一个`try`中捕获，或者允许终止脚本。

代码如下所示，在文件的底部还有一个self测试。它在Python 2.6和Python 3.0下都能够工作，因为它使用了Python 3.0的`print`和`raise`语法，尽管它在Python 2.6下只是捕获运算符重载方法属性（稍后更多讨论这一点）：

```
"""
Privacy for attributes fetched from class instances.
See self-test code at end of file for a usage example.
Decorator same as: Doubler = Private('data', 'size')(Doubler).
Private returns onDecorator, onDecorator returns onInstance,
and each onInstance instance embeds a Doubler instance.
"""

traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def Private(*privates):                         # privates in enclosing scope
    def onDecorator(aClass):                    # aClass in enclosing scope
        class onInstance:                       # wrapped in instance attribute
            def __init__(self, *args, **kargs):
                self.wrapped = aClass(*args, **kargs)
            def __getattr__(self, attr):                # My attrs don't call getattr
                trace('get:', attr)                     # Others assumed in wrapped
                if attr in privates:
                    raise TypeError('private attribute fetch: ' + attr)
                else:
                    return getattr(self.wrapped, attr)
            def __setattr__(self, attr, value):         # Outside accesses
```

```
                trace('set:', attr, value)              # Others run normally
                if attr == 'wrapped':                   # Allow my attrs
                    self.__dict__[attr] = value         # Avoid looping
                elif attr in privates:
                    raise TypeError('private attribute change: ' + attr)
                else:
                    setattr(self.wrapped, attr, value)  # Wrapped obj attrs
            return onInstance                           # Or use __dict__
        return onDecorator

    if __name__ == '__main__':
        traceMe = True

        @Private('data', 'size')                        # Doubler = Private(...)(Doubler)
        class Doubler:
            def __init__(self, label, start):
                self.label = label                      # Accesses inside the subject class
                self.data = start                       # Not intercepted: run normally
            def size(self):
                return len(self.data)                   # Methods run with no checking
            def double(self):                           # Because privacy not inherited
                for i in range(self.size()):
                    self.data[i] = self.data[i] * 2
            def display(self):
                print('%s => %s' % (self.label, self.data))

        X = Doubler('X is', [1, 2, 3])
        Y = Doubler('Y is', [-10, -20, -30])

        # The followng all succeed
        print(X.label)                                  # Accesses outside subject class
        X.display(); X.double(); X.display()            # Intercepted: validated, delegated
        print(Y.label)
        Y.display(); Y.double()
        Y.label = 'Spam'
        Y.display()

        # The following all fail properly
        """
        print(X.size())                     # prints "TypeError: private attribute fetch: size"
        print(X.data)
        X.data = [1, 1, 1]
        X.size = lambda S: 0
        print(Y.data)
        print(Y.size())
        """
```

当traceMe为True的时候，模块文件的self测试代码产生如下的输出。注意，装饰器是如何捕获和验证在包装的类之外运行的属性获取和赋值的，但是，却没有捕获类自身内部的属性访问：

```
[set: wrapped <__main__.Doubler object at 0x02B2AAF0>]
[set: wrapped <__main__.Doubler object at 0x02B2AE70>]
[get: label]
```

```
X is
[get: display]
X is => [1, 2, 3]
[get: double]
[get: display]
X is => [2, 4, 6]
[get: label]
Y is
[get: display]
Y is => [-10, -20, -30]
[get: double]
[set: label Spam]
[get: display]
Spam => [-20, -40, -60]
```

实现细节之一

这段代码有点复杂，并且你最好自己跟踪运行它，看看它是如何工作的。然而，为了帮助你理解，这里给出一些值得注意的提示。

继承与委托的关系

第29章中给出的粗糙的私有示例使用**继承**来混入`__setattr__`捕获访问。然而，继承使得这很困难，因为从类的内部或外部的访问之间的区分不是很直接的（内部访问应该允许常规运行，并且外部的访问应该限制）。要解决这个问题，第29章的示例需要继承类，以使用`__dict__`赋值来设置属性，这最多是一个不完整的解决方案。

这里的版本使用的**委托**（在另一个对象中嵌入一个对象），而不是继承。这种模式更好地适合于我们的任务，因为它使得区分主体对象的内部访问和外部访问容易了很多。对主体对象的来自外部的属性访问，由包装器层的重载方法拦截，并且如果合法的话，委托给类。类自身内部的访问（例如，通过其方法代码内的`self`）没有拦截并且允许不经检查而常规运行，因为这里没有继承私有的属性。

装饰器参数

这里使用的类装饰器接受任意多个参数，以命名私有属性。然而，真正发生的情况是，参数传递给了`Private`函数，并且`Private`返回了应用于主体类的装饰器函数。也就是说，在装饰器发生之前使用这些参数；`Private`返回装饰器，装饰器反过来把私有的列表作为一个封闭作用域应用来“记住”。

状态保持和封闭作用域

说到封闭的作用域，在这段代码中，实际上用到了3个层级的状态保持：

- Private的参数在装饰发生前使用，并且作为一个封闭作用域引用保持，以用于onDecorator和onInstance中。
- onDecorator的类参数在装饰时使用，并且作为一个封闭作用域引用保持，以便在实例构建时使用。
- 包装的实例对象保存为onInstance中的一个实例属性，以便随后从类外部访问属性的时候使用。

由于Python的作用域和命名空间规则，这些都很自然地工作。

使用__dict__和__slots__

这段代码中__setattr__依赖于一个实例对象的__dict__属性命名空间字典，以设置onInstance自己的包装属性。正如我们在上一章所了解到的，不能直接赋值一个属性而避免循环。然而，它使用了setattr内置函数而不是__dict__来设置包装对象自身之中的属性。此外，getattr用来获取包装对象中的属性，因为它们可能存储在对象自身中或者由对象继承。

因此，这段代码将对大多数类有效。你可能还记得，在第31章中介绍过，带有__slots__的新式类不能把属性存储到一个__dict__中。然而，由于我们在这里只是在onInstance层级依赖于一个__dict__，而不是在包装的实例中，并且因为setattr和getattr应用于基于__dict__和__slots__的属性，所以我们的装饰器应用于使用任何一种存储方案的类。

公有声明的泛化

既然有了一个Private实现，泛化其代码以考虑Public声明就很简单了——它们基本上是Private声明的反过程，因此，我们只需要取消内部测试。本节列出的实例允许一个类使用装饰器来定义一组Private或Public的实例属性（存储在一个实例上的属性，或者从其类继承的属性），使用如下的语法：

- Private声明类实例的那些不能获取或赋值的属性，而从类的方法的代码内部获取或赋值除外。也就是说，任何声明为Private的名称都不能从类的外部访问，而任何没有声明为Private的名称都可以自由地从类的外部获取或赋值。
- Public声明了一个类的实例属性，它可以从类的外部以及在类的方法内部获取和访问。也就是说，声明为Public的任何名称，可以从任何地方自由地访问，而没有声明为Public的任何名称，不能从类的外部访问。

Private和Public声明规定为互斥的：当使用了Private，所有未声明的名称都被认为

是Public的；并且当使用了Public，所有未声明的名称都被认为是Private。它们基本上相反，尽管未声明的、不是由类方法创建的名称行为略有不同——它们可以赋值并且由此从类的外部在Private之下创建（所有未声明的名称都是可以访问的），但不是在Public下创建的（所有未声明的名称都是不可访问的）。

再一次，自己研究这些代码并体验它们是如何工作的。注意，这个方案在顶层添加了额外的第四层状态保持，超越了前面描述的3个层次：lambda所使用的测试函数保存在一个额外的封闭作用域中。这个示例编写为可以在Python 2.6或Python 3.0下运行，尽管它在Python 3.0下运行的时候带有一个缺陷（在文件的文档字符串之后简短地说明，并且在代码之后详细说明）：

```
"""
Class decorator with Private and Public attribute declarations.
Controls access to attributes stored on an instance, or inherited
by it from its classes. Private declares attribute names that
cannot be fetched or assigned outside the decorated class, and
Public declares all the names that can. Caveat: this works in
3.0 for normally named attributes only: __X__ operator overloading
methods implicitly run for built-in operations do not trigger
either __getattr__ or __getattribute__ in new-style classes.
Add __X__ methods here to intercept and delegate built-ins.
"""

traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)
            def __getattr__(self, attr):
                trace('get:', attr)
                if failIf(attr):
                    raise TypeError('private attribute fetch: ' + attr)
                else:
                    return getattr(self.__wrapped, attr)
            def __setattr__(self, attr, value):
                trace('set:', attr, value)
                if attr == '_onInstance__wrapped':
                    self.__dict__[attr] = value
                elif failIf(attr):
                    raise TypeError('private attribute change: ' + attr)
                else:
                    setattr(self.__wrapped, attr, value)
        return onInstance
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))
```

```
def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))
```

参见前面示例的self测试代码，它是一个用法示例。这里在交互提示模式下快速地看看这些类装饰器的使用（它们在Python 2.6和Python 3.0下一样地工作），正如所介绍的那样，非Private或Public名称可以从主体类之外访问和修改，但是Private或非Public的名称不可以：

```
>>> from access import Private, Public

>>> @Private('age')                              # Person = Private('age')(Person)
... class Person:                                # Person = onInstance with state
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age                       # Inside accesses run normally
...
>>> X = Person('Bob', 40)
>>> X.name                                       # Outside accesses validated
'Bob'
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch: age
>>> X.age = 'Tom'
TypeError: private attribute change: age

>>> @Public('name')
... class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> X = Person('bob', 40)                        # X is an onInstance
>>> X.name                                       # onInstance embeds Person
'bob'
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch: age
>>> X.age = 'Tom'
TypeError: private attribute change: age
```

实现细节之二

为了帮助你分析这段代码，这里有一些关于这一版本的最后提示。由于这只是前面小节的示例的泛化，所以那里的大多数提示也适用于这里。

使用__X伪私有名称

除了泛化，这个版本还使用了Python的__X伪私有名称压缩功能（我们在第30章遇到过），来把包装的属性局部化为控制类，通过自动将其作为类名的前缀就可以做到。这避免了前面的版本与一个真实的、包装类可能使用的包装属性冲突的风险，并且它也是一个有用的通用工具。然而，它不是很“私有”，因为压缩的名称可以在类之外自由地使用。注意，在__setattr__中，我们也必须使用完整扩展的名称字符串('_onInstance__wrapped')，因为这是Python对其的修改。

破坏私有

尽管这个例子确实实现了对一个实例及其类的属性的访问控制，它可能以各种方式破坏了这些控制——例如，通过检查包装属性的显式扩展版本（bob.pay可能无效，因为完全压缩的bob._onInstance__wrapped.pay可能会有效）。如果你必须显式地这么做，这些控制可能对于常规使用来说足够了。当然，私有控制通常在任何语言中都会遭到破坏，如果你足够努力地尝试的话（#define private public在某些C++实现中也可能有效）。尽管访问控制可以减少意外修改，但这样的情况大多取决于使用任何语言的程序员。不管何时，源代码可能会被修改，访问控制总是管道流中的一小部分。

装饰器权衡

不用装饰器，我们也可以实现同样的结果，通过使用管理函数或者手动编写装饰器的名称重绑定；然而，装饰器语法使得代码更加一致而明确。这一方法以及任何其他基于包装的方法的主要潜在缺点是，属性访问导致额外调用，并且装饰的类的实例并不真的是最初的装饰类的实例——例如，如果你用X.__class__或isinstance(X, C)测试它们的类型，将会发现它们是包装类的实例。除非你计划在对象类型上进行内省，否则类型问题可能是不相关的。

开放问题

还是老样子，这个示例设计为在Python 2.6和Python 3.0下都能工作（提供了运算符重载方法，以便在包装器中重定义委托）。然而，和大多数软件一样，总是有改进的地方。

缺陷：运算符重载方法无法在Python 3.0下委托

就像使用__getattr__的所有的基于委托的类，这个装饰器只对常规命名的属性能够跨版本工作。像__str__和__add__这样在新式类下不同工作的运算符方法，在Python 3.0下运行的时候，如果定义了嵌入的对象，将无法有效到达。

正如我们在上一章所了解到的，传统类通常在运行时在实例中查找运算符重载名

称，但新式类不这么做——它们完全略过实例，在类中查找这样的方法。因此，在Python 2.6的新式类和Python 3.0的所有类中，__X__运算符重载方法显式地针对内置操作运行，**不会**触发__getattr__和__getattribute__。这样的属性获取将会和我们的onInstance.__getattr__一起忽略，因此，它们无法验证或委托。

我们的装饰器类没有编写为新式类（通过派生自object），因此，如果在Python 2.6下运行，它将会捕获运算符重载方法。由于在Python 3.0下所有的类自动都是新式类，如果它们在嵌入的对象上编码，这样的方法将会**失效**。Python 3.0中最简单的解决方案是，在onInstance中重新冗余地定义所有那些可能在包装的对象中用到的运算符重载方法。例如，可以手动添加额外的方法，可以通过工具来自动完成部分任务（例如，使用类装饰器或者下一章将要介绍的元类），或者通过在超类中定义。

要亲自看到不同，可尝试在Python 2.6下对使用运算符重载方法的一个类使用该装饰器。验证与前面一样有效，但是打印所使用的__str__方法和为+而运行的__add__方法二者都会调用装饰器的__getattr__，并由此最终将验证并正确地委托给主体Person对象：

```
C:\misc> c:\python26\python
>>> from access import Private
>>> @Private('age')
... class Person:
...     def __init__(self):
...         self.age = 42
...     def __str__(self):
...         return 'Person: ' + str(self.age)
...     def __add__(self, yrs):
...         self.age += yrs
...
>>> X = Person()
>>> X.age                                   # Name validations fail correctly
TypeError: private attribute fetch: age
>>> print(X)                                # __getattr__ => runs Person.__str__
Person: 42
>>> X + 10                                  # __getattr__ => runs Person.__add__
>>> print(X)                                # __getattr__ => runs Person.__str__
Person: 52
```

同样的代码在Python 3.0下运行的时候，显式地调用__str__和__add__将会忽略装饰器的__getattr__，并且在装饰器类之中或其上查找定义；print最终查找到从类类型继承的默认显示（从技术上讲，是从Python 3.0中隐藏的object超类），并且+产生一个错误，因为没有默认继承：

```
C:\misc> c:\python30\python
>>> from access import Private
>>> @Private('age')
... class Person:
...     def __init__(self):
```

```
...         self.age = 42
...     def __str__(self):
...         return 'Person: ' + str(self.age)
...     def __add__(self, yrs):
...         self.age += yrs
...
>>> X = Person()                                # Name validations still work
>>> X.age                                       # But 3.0 fails to delegate built-ins!
TypeError: private attribute fetch: age
>>> print(X)
<access.onInstance object at 0x025E0790>
>>> X + 10
TypeError: unsupported operand type(s) for +: 'onInstance' and 'int'
>>> print(X)
<access.onInstance object at 0x025E0790>
```

使用替代的__getattribute__方法在这里帮不上忙——尽管它定义为捕获每次属性引用（而不只是未定义的名称），它也不会由内置操作运行。我们在本书第37章介绍的Python的特性功能，在这里也帮不上忙。回忆一下，特性自动运行与在编写类的时候定义的**特定**属性相关的代码，并且不会设计来处理包装对象中的任意属性。

正如前面所提到的，Python 3.0中最直接的解决方案是：在类似装饰器的基于委托的类中，冗余地重新定义可能在嵌入对象中出现的运算符重载名称。这种方法并不理想，因为它产生了一些代码冗余，特别是与Python 2.6的解决方案相比较尤其如此。然而，这不会有太大的编码工作，在某种程度上可以使用工具或超类来自动完成，足以使装饰器在Python 3.0下工作，并且也允许运算符重载名称声明为Private或Public（假设每个运算符重载方法内部都运行failIf测试）：

```
def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)

            # Intercept and delegate operator overloading methods
            def __str__(self):
                return str(self.__wrapped)
            def __add__(self, other):
                return self.__wrapped + other
            def __getitem__(self, index):
                return self.__wrapped[index]            # If needed
            def __call__(self, *args, **kargs):
                return self.__wrapped(*arg, *kargs)     # If needed
            ...plus any others needed...

            # Intercept and delegate named attributes
            def __getattr__(self, attr):
                ...
            def __setattr__(self, attr, value):
                ...
```

```
        return onInstance
    return onDecorator
```

添加了这样的运算符重载方法，前面带有__str__和__add__的示例在Python 2.6和Python 3.0下都能同样地工作，尽管在Python 3.0下可能需要增加大量的额外代码——从原则上讲，**每个**不会自动运行的运算符重载方法，都需要在这样的一个通用工具类中针对Python 3.0冗余地定义（这就是为什么我们的代码省略这一扩展）。由于在Python 3.0中每个类都是新式的，所以在这一版本中，基于委托的代码更加困难（尽管不是没有可能）。

另一方面，委托包装器可以直接从一个曾经重定义了运算符重载方法的公共超类继承，使用标准的委托代码。此外，像额外的类装饰器或元类这样的工具，可能会自动地向委托类添加这样的方法，从而使一部分工作自动化（参见第39章中类扩展的示例以了解更多信息）。尽管还是不像Python 2.6中的解决方案那样简单，这样的技术能够帮助Python 3.0的委托类更加通用。

实现替代：__getattribute__插入，调用堆栈检查

尽管在包装器中冗余地定义运算符重载方法可能是前面介绍的Python 3.0难题的最直接解决方案，但它不是唯一的方法。我们没有足够篇幅来更深入地介绍这一问题，因此，研究其他潜在的解决方案就放在了一个建议的练习中。由于一个棘手的替代方案非常强调类概念，因此这里简单提及一下其优点。

这个示例的一个缺点是，实例对象并不真的是最初的类的实例——它们是包装器的实例。在某些依赖于类型测试的程序中，这可能就有麻烦。为了支持这样的类，我们可能试图通过在最初类中**插入**一个__getattribute__方法来实现类似的效果，以捕获在其实例上的**每次**属性引用。插入的方法将会把有效的请求向上传递到其超类以避免循环，使用我们在前面一章学习过的技术。如下是对类装饰器代码的潜在修改：

```
# trace support as before

def accessControl(failIf):
    def onDecorator(aClass):
        def getattributes(self, attr):
            trace('get:', attr)
            if failIf(attr):
                raise TypeError('private attribute fetch: ' + attr)
            else:
                return object.__getattribute__(self, attr)
        aClass.__getattribute__ = getattributes
        return aClass
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))
```

```
def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))
```

这一替代方案解决了类型测试的问题，但是带来了其他的问题。例如，它只处理属性**获取**——也就是，这个版本允许自由地对私有名称**赋值**。拦截赋值仍然必须使用__setattr__，或者是一个实例包装器对象，或者插入另一个类方法。添加一个实例包装器来捕获赋值可能会再次改变类型，并且如果最初的类使用自己的__setattr__（或者一个__getattribute__，对前面的情况），插入方法将会失效。一个插入的__setattr__还必须考虑到客户类中的一个__slots__。

此外，这种方法解决了上一小节介绍的**内置**操作属性问题，因为在这些情况下__getattribute__并不运行。在我们的例子中，如果Person有一个__str__，打印操作将运行它，但只是因为它真正出现在了那个类中。和前面一样，__str__属性**不会**一般性地路由到插入的__getattribute__方法——打印将会绕过这个方法，并且直接调用类的__str__。

尽管这可能比在包装对象内根本不支持运算符重载方法要好（除非重定义），但这种方法仍然没有拦截和验证__X__方法，这使得它们不可能成为Private的。尽管大多数运算符重载方法意味着是公有的，但有一些可能不是。

更糟糕的是，由于这个非包装器方法通过向装饰类添加一个__getattribute__来工作，它也会拦截**类自身**做出的属性访问，并像对来自类外部的访问一样地验证它们——这也意味着类的方法不能够使用Private名称！

实际上，像这样插入方法功能上等同于**继承**它们，并且意味着与我们在第29章最初的私有代码同样的限制。要知道一个属性访问是源自于类的内部还是外部，我们的方法需要在Python**调用堆栈**上检查frame对象。这可能最终产生一个解决方案（例如，使用检查堆栈的特性或描述符来替代私有属性），但是它可能会进一步减慢访问，并且其内部对我们来说过于复杂，无法在此介绍。

尽管有趣并且可能与一些其他的使用情况相关，但这种插入技术的方法并没有达到我们的目标。这里，我们不会进一步介绍这一选项的编码模式，因为我们将在下一章中学习类扩展技术，与元类联合使用。正如我们将在那里看到的，元类并不严格需要以这种方式修改类，因为类装饰器往往充当同样的角色。

Python不是关于控制

既然我已经用如此大的篇幅添加了对Python代码的Private和Public属性声明，必须再次提醒你，像这样为类添加访问控制不完全是Python的特性。实际上，大多数Python程

序员可能发现这一示例太大或者完全无关，除非充当装饰器的使用的一个展示。大多数大型的Python程序根本没有任何这样的控制而获得成功。如果你确实想要控制属性访问以杜绝编码错误，或者恰好近乎是专家级的C++或Java程序员，那么使用Python的运算符重载和内省工具，大多数事情是可以完成的。

示例：验证函数参数

作为装饰器工具的最后一个示例，本节开发了一个**函数装饰器**，它自动地测试传递给一个函数或方法的参数是否在有效的数值范围内。它设计用来在任何开发或产品阶段使用，并且它可以用作类似任务的一个模板（例如，参数类型测试，如果必须这么做的话）。由于本章的篇幅限制已经超出了，所以这个示例的代码主要靠自学，带有有限的说明。和往常一样，浏览代码来了解更多细节。

目标

在第27章面向对象教程中，我们编写了一个类，根据一个传入的百分比用来给表示人的对象涨工资：

```
class Person:
     ...
     def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

那里，我们注意到，如果想要编写健壮的代码，检查百分比以确保它不会太大或太小，这是一个好主意。我们可以在方法自身中使用`if`或`assert`语句来实现这样的检查，使用内嵌测试：

```
class Person:
    def giveRaise(self, percent):                # Validate with inline code
        if percent < 0.0 or percent > 1.0:
            raise TypeError, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))

class Person:                                    # Validate with asserts
    def giveRaise(self, percent):
        assert percent >= 0.0 and percent <= 1.0, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))
```

然而，这种方法使得带有内嵌测试的方法变得散乱，可能只有在开发阶段有用。对于更为复杂的情况，这可能很繁琐（假设试图内嵌代码来实现上一小节的装饰器所提供的属性私有）。可能更糟糕的是，如果需要修改验证逻辑，这里可能会有任意多个内嵌副本需要找到并更新。

一种更为有用和有趣的替代方法是，开发一个通用的工具来自动为我们执行范围测试，针对我们现在或将来要编写的任何函数或方法的参数。**装饰器**方法使得这明确而方便：

```
class Person:
    @rangetest(percent=(0.0, 1.0))              # Use decorator to validate
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

在装饰器中隔离验证逻辑，这简化了客户类和未来的维护。

注意，我们这里的目标和前面编写的属性验证不同。这里，我们想要验证传入的**函数参数**的值，而不是设置的**属性的值**。Python的装饰器和内省工具允许我们很容易地编写这一新的任务。

针对位置参数的一个基本范围测试装饰器

让我们从基本的范围测试实现开始。为了简化，我们将从编写一个只对位置参数有效的装饰器开始，并且假设它们在每次调用中总是出现在相同的位置。它们不能根据关键字名称传递，并且我们在调用中不支持额外的`**args`关键字，因为这可能会使装饰器中的位置声明无效。在名为*devtools.py*的文件中编写如下代码：

```
def rangetest(*argchecks):                      # Validate positional arg ranges
    def onDecorator(func):
        if not __debug__:                       # True if "python -O main.py args..."
            return func                         # No-op: call original directly
        else:                                   # Else wrapper while debugging
            def onCall(*args):
                for (ix, low, high) in argchecks:
                    if args[ix] < low or args[ix] > high:
                        errmsg = 'Argument %s not in %s..%s' % (ix, low, high)
                        raise TypeError(errmsg)
                return func(*args)
            return onCall
    return onDecorator
```

还是老样子，这段代码主要是修改了我们前面介绍的编码模式：我们使用装饰器参数，嵌套作用域以进行状态保持，等等。

我们还使用了嵌套的`def`语句以确保这对简单函数和方法都有效，就像在前面所学习到的。当用于类方法的时候，`onCall`在`*args`中的第一项接受主体类的实例，并且将其传递给最初的方法函数中的`self`；在这个例子中，范围测试中的参数数目从1开始，而不是从0开始。

还要注意到，这段代码使用了`__debug__`内置变量——Python将其设置为`True`，除非它

将以-O优化命令行标志运行（例如，python -O main.py）。当__debug__为False的时候，装饰器返回未修改的最初函数，以避免额外调用及其相关的性能损失。

这个第一次迭代解决方案使用如下：

```
# File devtools_test.py

from devtools import rangetest
print(__debug__)                        # False if "python —O main.py"

@rangetest((1, 0, 120))                 # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):                # age must be in 0..120
    print('%s is %s years old' % (name, age))

@rangetest([0, 1, 12], [1, 1, 31], [2, 0, 2009])
def birthday(M, D, Y):
    print('birthday = {0}/{1}/{2}'.format(M, D, Y))

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

    @rangetest([1, 0.0, 1.0])           # giveRaise = rangetest(...)(giveRaise)
    def giveRaise(self, percent):       # Arg 0 is the self instance here
        self.pay = int(self.pay * (1 + percent))

# Comment lines raise TypeError unless "python -O" used on shell command line

persinfo('Bob Smith', 45)               # Really runs onCall(...) with state
#persinfo('Bob Smith', 200)             # Or person if —O cmd line argument

birthday(5, 31, 1963)
#birthday(5, 32, 1963)

sue = Person('Sue Jones', 'dev', 100000)
sue.giveRaise(.10)                      # Really runs onCall(self, .10)
print(sue.pay)                          # Or giveRaise(self, .10) if —O
#sue.giveRaise(1.10)
#print(sue.pay)
```

运行的时候，这段代码中的有效调用产生如下的输出（本节中的所有代码在Python 2.6和Python 3.0下同样地工作，因为两个版本都支持函数装饰器，我们没有使用属性委托，并且我们使用Python 3.0式的print调用和意外构建语法）：

```
C:\misc> C:\python30\python devtools_test.py
True
Bob Smith is 45 years old
birthday = 5/31/1963
110000
```

取消掉对任何无效调用的注释，将会由装饰器引发一个TypeError。下面是允许最后两行运行的时候的结果（和往常一样，我省略了一些出错消息文本以节省篇幅）：

```
C:\misc> C:\python30\python devtools_test.py
True
Bob Smith is 45 years old
birthday = 5/31/1963
110000
TypeError: Argument 1 not in 0.0..1.0
```

在系统命令行，使用-O标志来运行Python，将会关闭范围测试，但是也会避免包装层的性能负担——我们最终直接调用最初未装饰的函数。假设这只是一个调试工具，可以使用这个标志来优化程序以供产品阶段使用：

```
C:\misc> C:\python30\python -O devtools_test.py
False
Bob Smith is 45 years old
birthday = 5/31/1963
110000
231000
```

针对关键字和默认泛化

前面的版本说明了我们需要使用的基础知识，但是它相当有局限——它只支持按照位置传递的参数的验证，并且它没有验证关键字参数（实际上，它假设不会有那种使得参数位置数不正确的关键字传递）。此外，对于在一个给定调用中可能忽略的默认参数，它什么也没有做。如果所有的参数都按照位置传递并且不是默认的，这没有问题，但在一个通用工具中，这是极少的理想状态。Python支持要灵活的多的参数传递模式，而这些我们还没有解决。

对示例的如下修改做得更好。通过把包装函数的期待参数与调用时实际传入的参数匹配，它支持对按照位置或关键字名称传入的参数的验证，并且对于调用忽略的默认参数，它会跳过测试。简而言之，要验证的参数通过关键字参数指定到装饰器，装饰器随后遍历*pargs positionals tuple和**kargs keywords字典以进行验证。

```
"""
File devtools.py: function decorator that performs range-test
validation for passed arguments. Arguments are specified by
keyword to the decorator. In the actual call, arguments may
be passed by position or keyword, and defaults may be omitted.
See devtools_test.py for example use cases.
"""

trace = True

def rangetest(**argchecks):             # Validate ranges for both+defaults
    def onDecorator(func):              # onCall remembers func and argchecks
        if not __debug__:               # True if "python -O main.py args..."
            return func                 # Wrap if debugging; else use original
        else:
            import sys
```

```
        code    = func.__code__
        allargs = code.co_varnames[:code.co_argcount]
        funcname = func.__name__

        def onCall(*pargs, **kargs):
            # All pargs match first N expected args by position
            # The rest must be in kargs or be omitted defaults
            positionals = list(allargs)
            positionals = positionals[:len(pargs)]

            for (argname, (low, high)) in argchecks.items():
                # For all args to be checked
                if argname in kargs:
                    # Was passed by name
                    if kargs[argname] < low or kargs[argname] > high:
                        errmsg = '{0} argument "{1}" not in {2}..{3}'
                        errmsg = errmsg.format(funcname, argname, low, high)
                        raise TypeError(errmsg)

                elif argname in positionals:
                    # Was passed by position
                    position = positionals.index(argname)
                    if pargs[position] < low or pargs[position] > high:
                        errmsg = '{0} argument "{1}" not in {2}..{3}'
                        errmsg = errmsg.format(funcname, argname, low, high)
                        raise TypeError(errmsg)
                else:
                    # Assume not passed: default
                    if trace:
                        print('Argument "{0}" defaulted'.format(argname))
                return func(*pargs, **kargs)        # OK: run original call
        return onCall
    return onDecorator
```

如下的测试脚本展示了如何使用装饰器——要验证的参数由关键字装饰器参数给定，并且在实际的调用中，我们可以按照名称或位置传递，或者如果它们是有效的话，可以用默认方式来忽略验证：

```
# File devtools_test.py
# Comment lines raise TypeError unless "python -O" used on shell command line
from devtools import rangetest

# Test functions, positional and keyword

@rangetest(age=(0, 120))                # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):
    print('%s is %s years old' % (name, age))

@rangetest(M=(1, 12), D=(1, 31), Y=(0, 2009))
def birthday(M, D, Y):
    print('birthday = {0}/{1}/{2}'.format(M, D, Y))

persinfo('Bob', 40)
persinfo(age=40, name='Bob')
birthday(5, D=1, Y=1963)
```

```
#persinfo('Bob', 150)
#persinfo(age=150, name='Bob')
#birthday(5, D=40, Y=1963)

# Test methods, positional and keyword

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay
                                        # giveRaise = rangetest(...)(giveRaise)
    @rangetest(percent=(0.0, 1.0))      # percent passed by name or position
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

bob = Person('Bob Smith', 'dev', 100000)
sue = Person('Sue Jones', 'dev', 100000)
bob.giveRaise(.10)
sue.giveRaise(percent=.20)
print(bob.pay, sue.pay)
#bob.giveRaise(1.10)
#bob.giveRaise(percent=1.20)

# Test omitted defaults: skipped

@rangetest(a=(1, 10), b=(1, 10), c=(1, 10), d=(1, 10))
def omitargs(a, b=7, c=8, d=9):

    print(a, b, c, d)

omitargs(1, 2, 3, 4)
omitargs(1, 2, 3)
omitargs(1, 2, 3, d=4)
omitargs(1, d=4)
omitargs(d=4, a=1)
omitargs(1, b=2, d=4)
omitargs(d=8, c=7, a=1)

#omitargs(1, 2, 3, 11)              # Bad d
#omitargs(1, 2, 11)                 # Bad c
#omitargs(1, 2, 3, d=11)            # Bad d
#omitargs(11, d=4)                  # Bad a
#omitargs(d=4, a=11)                # Bad a
#omitargs(1, b=11, d=4)             # Bad b
#omitargs(d=8, c=7, a=11)           # Bad a
```

这段脚本运行的时候，超出范围的参数会像前面一样引发异常，但参数可以按照名称或位置传递，并且忽略的默认参数不会验证。这段代码在Python 2.6和Python 3.0下都可以运行，但额外的元组圆括号在Python 2.6中会打印出来。跟踪输出并进一步测试它以自行体验；它像前面一样工作，但是，它的范围已经拓展了很多：

```
C:\misc> C:\python30\python devtools_test.py
Bob is 40 years old
Bob is 40 years old
birthday = 5/1/1963
```

```
110000 120000
1 2 3 4
Argument "d" defaulted
1 2 3 9
1 2 3 4
Argument "c" defaulted
Argument "b" defaulted
1 7 8 4
Argument "c" defaulted
Argument "b" defaulted
1 7 8 4
Argument "c" defaulted
1 2 8 4
Argument "b" defaulted
1 7 7 8
```

对于验证错误，当方法测试行之一注释掉的时候，我们像前面一样得到一个异常（除非-0命令行参数传递给Python）：

```
TypeError: giveRaise argument "percent" not in 0.0..1.0
```

实现细节

装饰器的代码依赖于内省API和对参数传递的细微限制。为了完整地泛化，以便我们可以从原则上整体模拟Python的参数匹配逻辑，来看到哪个名称以何种模式传入，但是，这对于我们的工具来说太复杂了。如果我们能够根据所有期待的参数的名称集合来按照名称匹配传入的参数，从而判断哪个位置参数真正地出现在给定的调用中，那将会更好。

函数内省

已经证实了内省API可以在函数对象以及其拥有我们所需的工具的相关代码对象上实现。第19章简单介绍过这个API，我们在这里实际地使用它。期待的参数名集合只是附加给一个函数的代码对象的前N个变量名：

```
# In Python 3.0 (and 2.6 for compatibility):
>>> def func(a, b, c, d):
...     x = 1
...     y = 2
...
>>> code = func.__code__                    # Code object of function object
>>> code.co_nlocals
6
>>> code.co_varnames                        # All local var names
('a', 'b', 'c', 'd', 'x', 'y')
>>> code.co_varnames[:code.co_argcount]     # First N locals are expected args
('a', 'b', 'c', 'd')

>>> import sys                              # For backward compatibility
```

```
>>> sys.version_info                          # [0] is major release number
(3, 0, 0, 'final', 0)
>>> code = func.__code__ if sys.version_info[0] == 3 else func.func_code
```

同样的API在较早的Python中也可以使用，但是，在Python 2.5及更早的版本中，func.__code__ attribute拼写为func.func_code in 2.5（为了可移植性，新的__code__属性在Python 2.6中也冗余地可用）。在函数上和代码对象上运行一个dir调用，以了解更多细节。

参数假设

给定期待的参数名的这个集合，该解决方案依赖于Python对于参数传递顺序所施加的两条限制（在Python 2.6和Python 3.0中都仍然成立）：

- 在调用时，所有的位置参数出现在所有关键字参数之前。
- 在def中，所有的非默认参数出现在所有的默认参数之前。

也就是说，在一个调用中，一个非关键字参数通常不会跟在一个关键字参数后面，并且在定义中，一个非默认参数不会跟在一个默认参数后面。在两种位置中，所有的“name=value”语法必须出现在任何简单的“name”之后。

为了简化，我们也可以假设一个调用一般是有效的——例如，所有的参数要么接收值（按照名称或位置），要么将有意忽略而选取默认值。不一定要进行这种假设，因为当包装逻辑测试有效性的时候，函数还没有真正调用——随后包装逻辑调用的时候，调用仍然可能失效，由于不正确的参数传递。只要这不会引发包装器在糟糕地失效，我们可以改进调用的验证。这是有帮助的，因为在调用发生之前验证它，将会要求我们完全模仿Python的参数匹配算法——再一次说明，这对我们的工具来说是一个过于复杂的过程。

匹配算法

现在，给定了这些限制和假设，我们可以用这一算法来考虑调用中的关键字以及忽略的默认参数。当拦截了一个调用，我们可以作如下假设：

- *pargs中的所有*N*个传递的位置参数，必须与从函数的代码对象获取的前*N*个期待的参数匹配。对于前面列出的每个Python的调用顺序规则都是如此，因为所有的位置参数在所有关键字参数之前。
- 要获取按照位置实际传递的参数的名称，我们可以把所有其他参数的列表分片为长度为*N*的*pargs位置参数元组。
- 前*N*个期待的参数之后的任何参数，要么是按照关键字传递，要么是调用时候忽略的默认参数。

- 对于要验证的每个参数名，如果它在**kargs中，它是按照名称传递的；如果它在前*N*个期待的参数中，它是按照位置传递的（在这种情况下，它在期待的列表中的相对位置给出了它在*pargs中的相对位置）；否则，我们可以假设它是在调用时候忽略的并且默认的参数，不需要检查。

换句话说，对于假设*pargs中前*N*个实际传递的位置参数，必须与期待的参数列表中的前*N*个参数匹配，并且任何其他的参数要么必须是按照关键字传送并位于**kargs中，要么必须是默认的参数，我们就可以略过对调用时忽略的参数的测试。在这种方法下，对于最右边的位置参数和最左边的关键字参数之间的、或者在关键字参数之间的、或者在所有的最右边的位置之后的那些忽略掉的参数，装饰器将省略对其检查。可以跟踪装饰器及其测试脚本，看看这是如何在代码中实现的。

开放问题

尽管我们的范围测试工具按照计划工作，但还是有两个缺陷。首先，正如前面提到的，对最初函数的无效调用在最终的装饰器中仍然会失效。例如，如下的两个调用都会触发异常：

```
omitargs()
omitargs(d=8, c=7, b=6)
```

然而，这只是失效，而我们想要调用最初的函数，在包装器的末尾。尽管我们可以尝试模仿Python的参数匹配来避免如此，但没有太多的理由来这么做——因为无论如何调用将会在此处失效，所以我们可能也想让Python自己的参数匹配逻辑来为我们检测问题。

最后，尽管最终的版本处理位置参数、关键字参数和省略的参数，但它仍然不会对将要在接受任意多个参数的装饰器函数中使用的*args和**args显式地做任何事情。然而，我们可能不需要关心自己的目标：

- 如果传递了一个额外的**关键字**参数，其名称将会出现在**kargs中，并且如果提交给装饰器，可以对其进行常规的测试。
- 如果**没有**传递一个额外的关键字，其名称不会出现在**kargs或者分片的期待位置列表中，由此，不会检查它——会当做默认参数来对待它，即便它实际上是一个可选的额外参数。
- 如果传递了一个额外的**位置**参数，没有办法在装饰器中引用它——其名称不会出现在**kargs或者分片的期待位置列表中，因此会直接忽略它。由于这样的参数没有在函数的定义中列出，所以没有办法把一个给装饰器的名称映射回到一个期待的相对位置。

换句话说，由于代码支持按照名称测试任意的关键字参数，但是不支持那些未命名的并且在函数的参数签名中没有预定位置的任意位置参数。

原则上，我们可以扩展装饰器的接口，以便在装饰的函数中支持*args，但这么做在极少情况下会有用（例如，一个特定参数名称带有一个测试，该测试应用于包装器的*pargs中超出期待参数列表的长度之外的所有参数）。既然我们已经在这个示例上耗费了很大的篇幅，所以如果你对这样的改进感兴趣，请在建议的练习中进一步研究这一主题。

装饰器参数 VS 函数注解

有趣的是，Python 3.0中提供的函数注解功能，可能为我们在指定范围测试的示例中所使用的装饰器参数给出了一种替代方法。正如我们在第19章中学到的，注解允许我们把表达式和参数及返回值关联起来，通过在自己的def头部行中编写它们。Python把注解收集到字典中并且将其附加给注解的函数。

我们可以在示例中的标题行编写范围限制，而不是在装饰器参数中编写。我们将仍然需要一个函数装饰器来包装函数以拦截随后的调用，但我们基本上换掉了装饰器参数语法：

```
@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                                  # func = rangetest(...)(func)
    print(a + b + c)
```

注解语法如下：

```
@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)
```

现在，范围限制移到了函数自身之中，而不是在外部编写。如下的脚本说明了两种方案下的最终装饰器的结构，以不完整的框架代码给出。装饰器参数编码模式就是我们前面给出的完整解决方案，注解替代方案需要一个较少层级的嵌套，因为它不需要保持装饰器参数：

```
# Using decorator arguments

def rangetest(**argchecks):
    def onDecorator(func):
        def onCall(*pargs, **kargs):
            print(argchecks)
            for check in argchecks: pass                # Add validation code here
            return func(*pargs, **kargs)
        return onCall
    return onDecorator
```

```
@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                              # func = rangetest(...)(func)
    print(a + b + c)

func(1, 2, c=3)                                 # Runs onCall, argchecks in scope

# Using function annotations

def rangetest(func):
    def onCall(*pargs, **kargs):
        argchecks = func.__annotations__
        print(argchecks)
        for check in argchecks: pass            # Add validation code here
        return func(*pargs, **kargs)
    return onCall

@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):            # func = rangetest(func)
    print(a + b + c)

func(1, 2, c=3)                                 # Runs onCall, annotations on func
```

运行的时候，两种方案都会访问同样的验证测试信息，但是，以不同的形式——装饰器参数版本的信息保持在封闭作用域的一个参数中；注解版本的信息保持在函数自身的一个属性中：

```
{'a': (1, 5), 'c': (0.0, 1.0)}
6
{'a': (1, 5), 'c': (0.0, 1.0)}
6
```

我将充实基于注解的版本留作一个建议的练习，其编码和我们前面给出的完整解决方案是相同的，因为范围测试信息直接在函数上而不是在一个封闭作用域中。实际上，所有这些给我们带来的是工具的一个不同的用户接口——仍然需要像前面一样，根据期待的参数名称来匹配参数名称，以获取相对位置。

实际上，在这个例子中，使用注解而不是装饰器参数确实**限制了其用途**。首先，注解只在Python 3.0下有效，因此，Python 2.6不再得到支持；另一方面，带有参数的函数装饰器，在两个版本下都有效。

更重要的是，通过把验证规范移动到`def`标题中，我们基本上给函数指定了一个**单独的角色**——因为注解允许我们为每个参数只编写一个表达式，它可以只有一个用途。例如，我们不能为其他用途而使用范围测试注解。

相反，由于装饰器参数在函数自身之外编写，所以它们更容易删除并且**更通用**——函数自身的代码并没有暗含任何单一的装饰目的。实际上，通过带有参数的**嵌套**装饰器，我

们可以对同一个函数应用多个扩展步骤；注解直接只支持一个步骤。使用装饰器参数，函数自身也保留一个简单的、常规的外观。

然而，如果有单一的目的，并且只使用Python 3.X，那么在注解和装饰器参数之间的选择很大程度上只是风格和个人主观爱好了。就像生活中常见的现象，一个人的注解恰好是另一个人的语法垃圾……

其他应用程序：类型测试

在处理装饰器参数时，我们所使用的编码模式可以应用于其他环境。例如，在开发时检查参数数据类型，这是一种直接的扩展：

```
def typetest(**argchecks):
    def onDecorator(func):
           ....
           def onCall(*pargs, **kargs):
               positionals = list(allargs)[:len(pargs)]
               for (argname, type) in argchecks.items():
                   if argname in kargs:
                       if not isinstance(kargs[argname], type):
                           ...
                           raise TypeError(errmsg)
                   elif argname in positionals:
                       position = positionals.index(argname)
                       if not isinstance(pargs[position], type):
                           ...
                           raise TypeError(errmsg)
                   else:
                       # Assume not passed: default
               return func(*pargs, **kargs)
           return onCall
    return onDecorator

@typetest(a=int, c=float)
def func(a, b, c, d):                   # func = typetest(...)(func)
    ...

func(1, 2, 3.0, 4)                      # Okay
func('spam', 2, 99, 4)                  # Triggers exception correctly
```

实际上，我们甚至可以通过传入一个测试函数来进一步泛化，就像我们在前面添加Public装饰时所做的那样；这种代码的单个副本足够用于范围和类型测试。正如前面的小节所述，对于这样的一个装饰器使用函数注解而不是装饰器参数，将会使其看起来更像是其他语言中的类型声明：

```
@typetest
def func(a: int, b, c: float, d):       # func = typetest(func)
    ...                                 # Gasp!...
```

正如我们已经在本书中学习到的，这一特定角色通常是工作代码中的一个糟糕思路，并且根本不是Python化的（实际上，它往往是有经验的C++程序员初次尝试使用Python的一种现象）。

类型测试限制了我们的函数只能在特定类型上工作，而不是允许它在任何具有可兼容性接口的类型上操作。实际上，它限制了代码并且破坏了其灵活性。另一方面，每个规则都有例外；类型检查可能在一种孤立的情况下很方便好用，即调试时以及与用更为限制性的语言（如C++等）编写的代码接口的时候。参数处理的这一通用模式，可能也适用于各种争议性较少的角色。

本章小结

在本章中，我们探讨了装饰器——适用于函数和类的各种情况。正如我们所学习的，装饰器是当一个函数或类定义的时候插入自动运行的代码的一种方式。当使用一个装饰器的时候，Python把函数或类名重新绑定到它返回的可调用对象。本书允许我们为函数调用和类实例创建调用添加一层包装器逻辑，以便管理函数和实例。正如我们已经看到的，管理器函数和手动名称重绑定都可以实现同样的效果，但装饰器提供了一种更加明确和统一的解决方案。

我们将在下一章看到，类装饰器也可以用来管理类本身，而不只是管理它们的实例。由于这一功能与下一章的主题元类有重合，所以需要阅读下一章。首先，做如下的练习题。由于本章主要关注其较大的示例，因此练习题将会要求你修改一些代码以便复习本章知识。

本章习题

1. 正如本章的提示中提到的，我们在本章“添加装饰器参数”小节所编写的带有装饰器参数的计时器函数装饰器，只能应用于简单**函数**，因为它使用了一个嵌套的类，该类带有一个`__call__`运算符重载方法来捕获调用。这种结构对于类**方法**无效，因为装饰器实例传递给`self`，而不是主体类实例。重新编写这个装饰器，以便它可以应用于简单函数和类方法，并且在函数和方法上都测试它（提示，参见本章“类错误之一：装饰类方法”小节）。注意，我们可以使用赋值函数对象属性来记录总的时间，因为你没有一个嵌套的类用于状态保持，并且不能从装饰器代码的外部访问非局部变量。
2. 我们在本章中编写的`Public/Private`类装饰器将会为一个装饰的类中的每次属性获取增加负担。尽管我们可以直接删除`@`装饰行以获取速度，但我们也可以扩展装饰

器自身类检查__debug__开关，并且在命令行传递-O Python标志的时候根本不执行包安装（正如我们对于参数范围测试装饰器所做的那样）。通过这种方法，我们可以加速程序而不用修改源代码，即通过命令行参数（python -O main.py...）。编写代码并测试这一扩展。

习题解答

1. 这里有一种方法来编写第一个问题的解决方案及其输出（虽然对类方法来说运行得太快而无法计时）。技巧在于用**嵌套的函数**替代嵌套的类，以便self参数不再是装饰器的实例，并且把整个事件赋值给装饰器函数自身，以便随后可以通过最初重绑定的名称来获取它（参见本章“状态信息保持选项”小节——函数支持任意的属性附件，并且函数名在这种环境中是一个封装的作用域引用）。

```
import time

def timer(label='', trace=True):                # On decorator args: retain args
    def onDecorator(func):                      # On @: retain decorated func
        def onCall(*args, **kargs):             # On calls: call original
            start = time.clock()                # State is scopes + func attr
            result = func(*args, **kargs)
            elapsed = time.clock() - start
            onCall.alltime += elapsed
            if trace:
                format = '%s%s: %.5f, %.5f'
                values = (label, func.__name__, elapsed, onCall.alltime)
                print(format % values)
            return result
        onCall.alltime = 0
        return onCall
    return onDecorator

# Test on functions

@timer(trace=True, label='[CCC]==>')
def listcomp(N):                                # Like listcomp = timer(...)(listcomp)
    return [x * 2 for x in range(N)]            # listcomp(...) triggers onCall

@timer(trace=True, label='[MMM]==>')
def mapcall(N):
    return list(map((lambda x: x * 2), range(N)))     # list() for 3.0 views

for func in (listcomp, mapcall):
    result = func(5)                            # Time for this call, all calls, return value
    func(5000000)
    print(result)
    print('allTime = %s\n' % func.alltime)      # Total time for all calls

# Test on methods

class Person:
    def __init__(self, name, pay):
```

```
            self.name = name
            self.pay = pay

        @timer()
        def giveRaise(self, percent):             # giveRaise = timer()(giveRaise)
            self.pay *= (1.0 + percent)           # tracer remembers giveRaise

        @timer(label='**')
        def lastName(self):                       # lastName = timer(...)(lastName)
            return self.name.split()[-1]          # alltime per class, not instance

    bob = Person('Bob Smith', 50000)
    sue = Person('Sue Jones', 100000)
    bob.giveRaise(.10)
    sue.giveRaise(.20)                            # runs onCall(sue, .10)
    print(bob.pay, sue.pay)
    print(bob.lastName(), sue.lastName())         # runs onCall(bob), remembers lastName
    print('%.5f %.5f' % (Person.giveRaise.alltime, Person.lastName.alltime))

    # Expected output

    [CCC]==>listcomp: 0.00002, 0.00002
    [CCC]==>listcomp: 1.19636, 1.19638
    [0, 2, 4, 6, 8]
    allTime = 1.19637775192
    [MMM]==>mapcall: 0.00002, 0.00002
    [MMM]==>mapcall: 2.29260, 2.29262
    [0, 2, 4, 6, 8]
    allTime = 2.2926232943

    giveRaise: 0.00001, 0.00001
    giveRaise: 0.00001, 0.00002
    55000.0 120000.0
    **lastName: 0.00001, 0.00001
    **lastName: 0.00001, 0.00002
    Smith Jones
    0.00002 0.00002
```

2. 下面的代码解决了第二个问题——它已经扩展来在优化的模式中返回最初的类(-O)，因此，属性访问不会引发速度问题。实际上，我所做的是添加调试模式的测试语句，并且进一步向右缩进类。如果想要在Python 3.0下也支持把这些委托给主体类，那么向包装类添加运算符重载方法重定义（Python 2.6通过__getattr__路由这些，但Python 3.0和Python 2.6中的新式类不这么做）。

```
traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
    def onDecorator(aClass):
        if not __debug__:
            return aClass
        else:
            class onInstance:
```

```
                def __init__(self, *args, **kargs):
                    self.__wrapped = aClass(*args, **kargs)
                def __getattr__(self, attr):
                    trace('get:', attr)
                    if failIf(attr):
                        raise TypeError('private attribute fetch: ' + attr)
                    else:
                        return getattr(self.__wrapped, attr)
                def __setattr__(self, attr, value):
                    trace('set:', attr, value)
                    if attr == '_onInstance__wrapped':
                        self.__dict__[attr] = value
                    elif failIf(attr):
                        raise TypeError('private attribute change: ' + attr)
                    else:
                        setattr(self.__wrapped, attr, value)
            return onInstance
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))

def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))

# Test code: split me off to another file to reuse decorator

@Private('age')                          # Person = Private('age')(Person)
class Person:                            # Person = onInstance with state
    def __init__(self, name, age):
        self.name = name
        self.age = age                   # Inside accesses run normally

X = Person('Bob', 40)
print(X.name)                            # Outside accesses validated
X.name = 'Sue'
print(X.name)
#print(X.age)           # FAILS unles "python -O"
#X.age = 999            # ditto
#print(X.age)           # ditto

@Public('name')
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

X = Person('bob', 40)                    # X is an onInstance
print(X.name)                            # onInstance embeds Person
X.name = 'Sue'
print(X.name)
#print(X.age)        # FAILS unless "python -O main.py"
#X.age = 999         # ditto
#print(X.age)        # ditto
```

第39章

元类

在上一章中，我们介绍了装饰器并研究了其应用的各种示例。在本书的最后一章中，我们将继续关注工具构建器，并讨论另一个高级话题：**元类**。

从某种意义上讲，元类只是扩展了装饰器的代码插入模式。正如我们在上一章所学到的，函数和类装饰器允许我们拦截并扩展函数调用以及类实例创建调用。以类似的思路，元类允许我们拦截并扩展**类创建**——它们提供了一个API以插入在一条`class`语句结束时运行的额外逻辑，尽管是以与装饰器不同的方式。同样，它们提供了一种通用的协议来管理程序中的类对象。

就像本书的这一部分其他各章所讨论的主题一样，这是一个**高级话题**，需要有一定的基础。实际上，元类允许我们获得更高层级的控制，来控制一组类如何工作。这是一个功能强大的概念，并且元类并不是打算供大多数应用程序员使用的（或者，坦白地说，不适合懦弱之人）。

另一方面，元类为各种没有它而难以实现或不可能实现的编码模式打开了大门，并且对于那些追求编写灵活的API或编程工具供其他人使用的程序员来说，它特别有用。即便你不属于此类程序员，元类也可以教你很多有关Python的类模型的一般性知识。

和上一章一样，我们这里的部分目标只是展示比本书前面的示例更为实际的代码实例。尽管元类是一个核心主题而且并非自成一个应用领域，本章的部分目标是激发你在阅读完本书后继续研究较大的应用程序编程示例的兴趣。

要么是元类，要么不是元类

元类可能是本书中最高级的主题，如果不把Python语言算作整体的话。借用经验丰富

的Python开发者Tim Peters在*comp.lang.python*新闻组中的话来说（Tim Peters是著名的Python座右铭“import this”的作者）：

> [元类]比99%的用户所担心的魔力要更深。如果你犹豫是否需要它们，那你不需要它们（真正需要元类的人，能够确定地知道需要它们，并且不需要说明为什么需要）。

换句话说，元类主要是针对那些构建API和工具供他人使用的程序员。在很多情况下（如果不是大多数的话），它们可能不是应用程序工作的最佳选择。在开发其他人将来会用的代码的时候，尤其如此。“因为某物很酷”而编写它，似乎不是一种合理的判断，除非你在做实验或者学习。

然而，元类有着各种各样广泛的潜在角色，并且知道它们何时有用是很重要的。例如，它们可能用来扩展具有跟踪、对象持久、异常日志等功能的类。它们也可以用来在运行时根据配置文件来构建类的一部分，对一个类的每个方法广泛地应用函数装饰器，验证其他的接口的一致性，等等。

在它们更宏观的化身中，元类甚至可以用来实现替代的编程模式，例如面向方面编程、数据库的对象/关系映射（ORM），等等。尽管常常有实现这些结果的替代方法（正如我们看到的，类装饰器和元类的角色常常有重合），元类提供了一种正式模型，可以裁减以完成那些任务。我们没有足够的篇幅在本章中介绍所有这些第一手的应用程序，但是，在此学完了基础知识后，你应该自行在Web上搜索以找到其他的用例。

在本书中学习元类的可能原因是，这个主题能够帮助更广泛地说明Python的类机制。尽管你可能在自己的工作中编写或重用它们，也可能不会这么做，大概理解元类也可以在很大程度上更深入地理解Python。

提高魔力层次

本书的大多数部分关注直接的应用程序编码技术，因为大多数程序员都花费时间来编写模块、函数和类来实现现实的目标。他们也使用类和创建实例，并且可能甚至做一些运算符重载，但是，他们可能不会太深入地了解类实际是如何工作的细节。

然而，在本书中我们已经看到了各种工具，它们允许我们以广泛的方式控制Python的行为，并且它们常常与Python的内部与工具构建有更多的关系，而与应用程序编程领域相关甚少：

内省属性

像`__class__`和`__dict__`这样的特殊属性允许我们查看Python对象的内部实现方面，以便更广泛地处理它们，列出对象的所有属性、显示一个类名，等等。

运算符重载方法

像__str__和__add__这样特殊命名的方法，在类中编写来拦截并提供应用于类实例的内置操作的行为，例如，打印、表达式运算符等等。它们自动运行作为对内置操作的响应，并且允许类符合期望的接口。

属性拦截方法

一类特殊的运算符重载方法提供了一种方法在实例上广泛地拦截属性访问：__getattr__、__setattr__和__getattribute__允许包装的类插入自动运行的代码，这些代码可以验证属性请求并且将它们委托给嵌入的对象。它们允许一个对象的任意数目的属性——要么是选取的属性，要么是所有的属性——在访问的时候计算。

类特性

内置函数property允许我们把代码和特殊的类属性关联起来，当获取、赋值或删除该属性的时候就自动运行代码。尽管不像前面一段所介绍的工具那样通用，特性考虑到了访问特定属性时候的自动代码调用。

类属性描述符

其实，特性只是定义根据访问自动运行函数的属性描述符的一种简洁方式。描述符允许我们在单独的类中编写__get__、__set__和__delete__处理程序方法，当分配给该类的一个实例的属性被访问的时候自动运行它们。它们提供了一种通用的方式，来插入当访问一个特定的属性时自动运行的代码，并且在一个属性的常规查找之后触发它们。

函数和类装饰器

正如我们在第38章看到的，装饰器的特殊的@可调用语法，允许我们添加当调用一个函数或创建一个类实例的时候自动运行的逻辑。这个包装器逻辑可以跟踪或计时调用，验证参数，管理类的所有实例，用诸如属性获取验证的额外行为来扩展实例，等等。装饰器语法插入名称重新绑定逻辑，在函数或类定义语句的末尾自动运行该逻辑——装饰的函数和类名重新绑定到拦截了随后调用的可调用对象。

正如本章的介绍中所提到的，**元类**是这些技术的延续——它们允许我们在一条class语句的末尾，插入当创建一个类对象的时候自动运行的逻辑。这个逻辑不会把类名重新绑定到一个装饰器可调用对象，而是把类自身的创建指向特定的逻辑。

换句话说，元类最终只是定义自动运行代码的另外一种方式。通过元类以及前面列出的其他工具，Python为我们提供了在各种环境中插入逻辑的方法——在运算符计算时、属性访问时、函数调用时、类实例创建时，现在是在类对象创建时。

和类装饰器不同，它通常是添加**实例**创建时运行的逻辑，元类在**类**创建时运行。同样的，它们都是通常用来管理或扩展类的钩子，而不是管理其实例。

例如，元类可以用来自动为类的所有方法添加装饰，把所有使用的类注册到一个API，自动为类添加用户接口逻辑，在文本文件中从简单声明来创建或扩展类，等等。由于我们可以控制如何创建类（并且通过它们的实例获取的行为），它们的实用性潜在地很广泛。

正如我们已经看到的，这些高级Python工具中的很多都有交叉的角色。例如，属性往往可以用特性、描述符或属性拦截方法来管理。正如我们在本章中见到的，类装饰器和元类往往可以交换使用。尽管类装饰器常常用来管理实例，它们也可以用来管理类；类似的，尽管元类设计用来扩展类构建，它们也常常插入代码来管理实例。尽管选择使用哪种技术有时候纯粹是主观的事情，但知道替代方案可以帮助我们为给定的任务挑选正确的工具。

“辅助”函数的缺点

也像上一章介绍的装饰器一样，元类常常是可选的，从理论的角度来看是这样。我们通常可以通过**管理器函数**（有时候叫做“辅助”函数）传递类对象来实现同样的效果，这和我们通过管理器代码传递函数和实例来实现装饰器的目的很相似。然而，就像装饰器一样，元类：

- 提供一种更为正式和明确的结构。
- 有助于确保应用程序员不会忘记根据一个API需求来扩展他们的类。
- 通过把类定制逻辑工厂化到一个单独的位置（元类）中，避免代码冗余及其相关的维护成本。

为了便于说明，假设我们想要在一组类中自动插入一个方法。当然，如果在编写类的时候知道主体方法，我们可以用简单的**继承**来做到这点。在那种情况下，我们可以直接在超类中编写该方法，并且让所有涉及的类继承它：

```
class Extras:
    def extra(self, args):              # Normal inheritance: too static
        ...

class Client1(Extras): ...              # Clients inherit extra methods
class Client2(Extras): ...
class Client3(Extras): ...

X = Client1()                           # Make an instance
X.extra()                               # Run the extra methods
```

然而，有时候，在编写类的时候不可能预计这样的扩展。考虑扩展类以响应在运行时用户界面中所做出的一个选择，或者在配置文件中指定类型的情况。尽管我们可以在我们想象的集合中编写每个类以**手动**检查这些，但有太多的问题要问客户类（这里的需求是抽象的，是需要填充的东西）：

```
def extra(self, arg): ...

class Client1: ...                          # Client augments: too distributed
if required():
    Client1.extra = extra

class Client2: ...
if required():
    Client2.extra = extra

class Client3: ...
if required():
    Client3.extra = extra

X = Client1()
X.extra()
```

我们可以像这样在`class`语句之后把方法添加到类，因为类方法只不过是和一个类相关的函数，并且拥有第一个参数来接收`self`实例。尽管这有效，但它把所有的扩展负担放到了客户类上（并且假设它们能记住做这些）。

从维护的角度，把选择逻辑隔离到一个单独的地方，这可能会更好。我们可以通过把类指向一个**管理器函数**，从而把一些这些额外工作的一部分**封装**起来——这样的一个管理器函数将根据需求扩展类，并且处理运行时测试和配置的所有工作：

```
def extra(self, arg): ...

def extras(Class):                          # Manager function: too manual
    if required():
        Class.extra = extra

class Client1: ...
extras(Client1)

class Client2: ...
extras(Client2)

class Client3: ...
extras(Client3)

X = Client1()
X.extra()
```

这段代码通过紧随类创建之后一个管理器函数来运行类。尽管像这样的一个管理器函数在这里可以实现我们的目标，但它们仍然给类的编写者增加了相当重的负担，所以编写

者必须理解需求并且将它们附加到代码中。如果有一种简单的方式在主体类中增强这种扩展，那将会更好，这样，编写者就不需要处理并且不会忘记使用扩展。换句话说，我们宁愿能够在一条class语句的末尾插入一些**自动**运行的代码，从而扩展该类。

这正是**元类**所做的事情——通过声明一个元类，我们告诉Python把类对象的创建路由到我们所提供的另一个类：

```
def extra(self, arg): ...

class Extras(type):
    def __init__(Class, classname, superclasses, attributedict):
        if required():
            Class.extra = extra

class Client1(metaclass=Extras): ...   # Metaclass declaration only
class Client2(metaclass=Extras): ...   # Client class is instance of meta
class Client3(metaclass=Extras): ...

X = Client1()                          # X is instance of Client1
X.extra()
```

由于创建新的类的时候，Python在class语句的末尾自动调用元类，因此它可以根据需要扩展、注册或管理类。此外，客户类唯一的需求是，它们声明元类。每个这么做的类都将自动获取元类所提供的扩展，现在可以，如果将来元类修改了也可以。然而在一个小的示例中看到这点可能有些困难，元类通常比其他方法能够更好地处理这样的任务。

元类与类装饰器的关系：第一回合

我们已经讲过，前面一章所介绍的**类装饰器**有时候在功能上与元类有重合，注意到这点也很有趣。尽管类装饰器通常用来管理或扩展实例，但它们也可以扩展类，而独立于任何创建的实例。

例如，假设我们编写自己的管理器函数以返回扩展的类，而不是直接在原处修改它。这就允许更大程度的灵活性，因为管理器将会自由地返回实现了类期待接口的任何类型的对象：

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

class Client1: ...
Client1 = extras(Client1)

class Client2: ...
Client2 = extras(Client2)
```

```
class Client3: ...
Client3 = extras(Client3)

X = Client1()
X.extra()
```

如果你认为这只是回顾类装饰器的开始，那么你是对的。在前一章中，我们介绍了类装饰器作为扩展**实例**创建调用的工具。由于它们通过自动把一个类名绑定到一个函数的结果而工作，那么，没有理由在任何实例创建之前不能用它来扩展类。也就是说，在创建的时候，类装饰器可以对**类**应用额外的逻辑，而不只是对**实例**应用：

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

@extras
class Client1: ...              # Client1 = extras(Client1)

@extras
class Client2: ...              # Rebinds class independent of instances

@extras
class Client3: ...

X = Client1()                   # Makes instance of augmented class
X.extra()                       # X is instance of original Client1
```

这里，装饰器基本上会把前面示例的手动名称重新绑定自动化。就像是使用元类，由于装饰器返回最初的类，实例由此创建，而不是由包装器对象创建。实际上，根本没有拦截实例创建。

在这个特定的例子中（在类创建的时候给类添加方法），元类和装饰器之间的选择有些随意。装饰器可以用来管理实例和类，并且它们与元类在第二种角色中交叉了。

然而，这真的只是说明了元类的一种操作模式。正如我们将看到的，在这种角色中，装饰器对应到元类的`__init__`方法，但是，元类还有其他的定制钩子。正如我们还将看到的，除了类初始化，元类可以执行任意的构建任务，而这些可能对装饰器来说更难。

此外，尽管装饰器可以管理实例和类，反之却不然——元类设计来管理类，并且用它们来管理实例却不是很容易。我们将在本章稍后的代码中介绍这一区别。

本节的大多数代码都已经简化过，但是，我们将在本章后面将其补充为真实有用的示例。要完全理解元类是如何工作的，首先需要对其底层模型有一个清晰的印象。

元类模型

要真正理解元类是如何工作的，需要更多地理解Python的类型模型以及在class语句的末尾发生了什么。

类是类型的实例

到目前为止，在本书中，我们已经通过创建内置类型（如列表和字符串）的实例，以及我们自己编写的类的实例，完成了大多数工作。正如我们已经看到的，类的实例有一些自己的状态信息属性，但它们也从所创建自的类继承了行为属性。对于内置类型也是如此，例如，列表实例拥有自己的值，但是它们从列表类型继承方法。

尽管我们可以用这样的实例对象做很多事情，但Python的类型模型实际上比我前面所介绍的要丰富一些。实际上，该模型中有一个我们目前为止可以看到的漏洞：如果实例自类创建，那么，是什么创建了类？这证明了类也是某物的实例：

- *在Python 3.0*中，用户定义的类对象是名为type的对象的实例，type本身是一个类。
- *在Python 2.6*中，新式类继承自object，它是type的一个子类；传统类是type的一个实例，并且并不创建自一个类。

我们在本书第9章中介绍了类型的概念，在第31章介绍了类和类型的关系，但是，让我们在这里回顾一下基础知识，看看它们是如何应用于元类的。

还记得吧，type内置函数返回任何对象的类型（它本身是一个对象）。对于列表这样的内置类型，实例的类型是一个内置的列表类型，但是，列表类型的类型是类型type自身——顶层的type对象创建了具体的类型，具体的类型创建了实例。你将会在交互提示模式中亲自看到这点。例如，在Python 3.0中：

```
C:\misc> c:\python30\python
>>> type([])                    # In 3.0 list is instance of list type
<class 'list'>
>>> type(type([]))              # Type of list is type class
<class 'type'>

>>> type(list)                  # Same, but with type names
<class 'type'>
>>> type(type)                  # Type of type is type: top of hierarchy
<class 'type'>
```

正如我们在第31章中学习新式类的时候所看到的，在Python 2.6中（及更早的版本中），通常也是一样的，但是，类型并不完全和类相同——type是一种独特的内置对象，它位于类型层级的顶层，并且用来构建类型：

```
C:\misc> c:\python26\python
>>> type([])                    # In 2.6, type is a bit different
<type 'list'>
>>> type(type([]))
<type 'type'>

>>> type(list)
<type 'type'>
>>> type(type)
<type 'type'>
```

这说明了类型/实例关系对于类来说也是成立的：实例创建自类，而类创建自type。在Python 3.0中，“类型”的概念与“类”的概念合并了。实际上，这两者基本上是同义词——**类是类型，类型也是类**。即：

- 类型由派生自type的类定义。
- 用户定义的类是类型类的实例。
- 用户定义的类是产生它们自己的实例的类型。

正如我们在前面看到的，这一对等效果的代码测试实例的类型：一个实例的类型是产生它的类。这也暗示着，创建类的方式证明是本章主题的关键所在。由于类通常默认地创建自一个根类型类，因此大多数程序员不需要考虑这种类型/类对等性。然而，它开创了定制类及其实例的新的可能性。

例如，Python 3.0中的类（以及Python 2.6中的新式类）是type类的实例，并且实例对象是它们的类的实例。实际上，类现在有了连接到type的一个__class__，就像是一个实例有了链接到创建它的类的__class__：

```
C:\misc> c:\python30\python
>>> class C: pass               # 3.0 class object (new-style)
...
>>> X = C()                     # Class instance object

>>> type(X)                     # Instance is instance of class
<class '__main__.C'>
>>> X.__class__                 # Instance's class
<class '__main__.C'>

>>> type(C)                     # Class is instance of type
<class 'type'>
>>> C.__class__                 # Class's class is type
<class 'type'>
```

特别注意这里的最后两行——类是type类的实例，就像常规的实例是一个类的实例一样。在Python 3.0中，这对于内置类和用于定义的类类型都是成立的。实际上，类根本不是一个独立的概念：它们就是用户定义的类型，并且type自身也是由一个类定义的。

在Python 2.6中，对于派生自object的新式类，情况也是如此，因此，这保证了Python 3.0的类行为：

```
C:\misc> c:\python26\python
>>> class C(object): pass                # In 2.6 new-style classes,
...                                      # classes have a class too
>>> X = C()

>>> type(X)
<class '__main__.C'>
>>> type(C)
<type 'type'>

>>> X.__class__
<class '__main__.C'>
>>> C.__class__
<type 'type'>
```

然而，Python 2.6中的传统类略有不同——因为它们反映了老Python版本中的类模型，它们没有一个__class__链接，并且像Python 2.6中的内置类型一样，它们是type的实例，而不是一个类型实例：

```
C:\misc> c:\python26\python
>>> class C: pass                        # In 2.6 classic classes,
...                                      # classes have no class themselves
>>> X = C()

>>> type(X)
<type 'instance'>
>>> type(C)
<type 'classobj'>

>>> X.__class__
<class __main__.C at 0x005F85A0>
>>> C.__class__
AttributeError: class C has no attribute '__class__'
```

元类是Type的子类

那么，为什么我们说类是Python 3.0中的一个type类的实例？事实证明，这是允许我们编写元类的钩子。由于**类型**的概念类似于今天的**类**，所以我们可以用常规的面向对象技术子类type，并且用类语法来定制它。由于类实际上是type类的实例，从type的定制的子类创建类允许我们实现各种定制的类。更详细地说，这是很自然的解决方案——在Python 3.0中以及在Python 2.6的新式类中：

- type是产生用户定义的类的一个类。
- 元类是type类的一个子类。

- 类对象是type类的一个实例，或一个子类。
- 实例对象产生字一个类。

换句话说，为了控制创建类以及扩展其行为的方式，我们所需要做的只是指定一个用户定义的类创建自一个用户定义的元类，而不是常规的type类。

注意，这个**类型实例**关系与**继承**并不完全相同：用户定义的类可能也拥有超类，它们及其实例从那里继承属性（继承超类在class语句的圆括号中列出，并且出现在一个类的__bases__元组中）。类创建自的类型，以及它是谁的实例，这是不同的关系。下一小节将描述Python所遵从的实现这种实例关系的过程。

Class语句协议

子类化type类以定制它，这其实只是元类背后的魔力的一半。我们仍然需要把一个类的创建指向元类，而不是默认type。为了完全理解这是如何安排的，我们还需要知道class语句如何完成其工作。

我们已经学习过，当Python遇到一条class语句，它会运行其嵌套的代码块以创建其属性——所有在嵌套代码块的顶层分配的名称都产生结果的类对象中的属性。这些名称通常是嵌套的def所创建的方法函数，但是，它们也可以是分配来创建由所有实例共享的类数据的任意属性。

从技术上讲，Python遵从一个标准的协议来使这发生：在**一条class语句的末尾**，并且在运行了一个命名控件词典中的所有嵌套代码之后，它调用type对象来创建class对象：

```
class = type(classname, superclasses, attributedict)
```

type对象反过来定义了一个__call__运算符重载方法，当调用type对象的时候，该方法运行两个其他的方法：

```
type.__new__(typeclass, classname, superclasses, attributedict)
type.__init__(class, classname, superclasses, attributedict)
```

__new__方法创建并返回了新的class对象，并且随后__init__方法初始化了新创建的对象。正如我们稍后将看到的，这是type的元类子类通常用来定制类的钩子。

例如，给定一个如下所示的类定义：

```
class Spam(Eggs):              # Inherits from Eggs
    data = 1                   # Class data attribute
    def meth(self, arg):       # Class method attribute
        pass
```

Python将会从内部运行嵌套的代码块来创建该类的两个属性（data和meth），然后在class语句的末尾调用type对象，产生class对象：

```
Spam = type('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

由于这个调用在class语句的末尾进行，它是用来扩展或处理一个类的、理想的钩子。技巧在于，用将要拦截这个调用的一个定制子类来替代类型，下一节将展示如何做到这一点。

声明元类

正如我们刚才看到的，类默认是type类创建的。要告诉Python用一个定制的元类来创建一个类，直接声明一个元类来拦截常规的类创建调用。怎么做到这点，依赖于你使用哪个Python版本。在Python 3.0中，在类标题中把想要的元类作为一个关键字参数列出来：

```
class Spam(metaclass=Meta):                    # 3.0 and later
```

继承超类也可以列在标题中，在元类之前。例如，在下面的代码中，新的类Spam继承自Eggs，但也是Meta的一个实例并且由Meta创建：

```
class Spam(Eggs, metaclass=Meta):              # Other supers okay
```

我们可以在Python 2.6中得到同样的效果，但是，我们必须不同地指定元类——使用一个类属性而不是一个关键字参数。为了使其成为一个新式类，需要object派生，并且这种形式在Python 3.0中不再有效，而是作为属性直接忽略：

```
class spam(object):                            # 2.6 version (only)
     __metaclass__ = Meta
```

在Python 2.6中，一个全局模块__metaclass__变量也可以用来把模块中的所有类链接到一个元类。这在Python 3.0中不再支持，因为它有意作为一个临时性措施，使得更容易预设为新式类而不用从object派生每个类。

当以这些方式声明的时候，创建类对象的调用在class语句的底部运行，修改为调用元类而不是默认的type：

```
class = Meta(classname, superclasses, attributedict)
```

由于元类是type的一个子类，所以type类的__call__把创建和初始化新的类对象的调用委托给元类，如果它定义了这些方法的定制版本：

```
Meta.__new__(Meta, classname, superclasses, attributedict)
Meta.__init__(class, classname, superclasses, attributedict)
```

为了展示，这里再次给出前一节的例子，用Python 3.0的元类声明扩展：

```
class Spam(Eggs, metaclass=Meta):       # Inherits from Eggs, instance of Meta
    data = 1                            # Class data attribute
    def meth(self, arg):                # Class method attribute
        pass
```

在这条class语句的末尾，Python内部运行如下的代码来创建class对象：

```
Spam = Meta('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

如果元类定义了__new__或__init__的自己版本，在此处的调用期间，它们将依次由继承的type类的__call__方法调用，以创建并初始化新类。下一节将介绍我们如何编写元类谜题的最后一块。

编写元类

到目前为止，我们已经看到了Python如何把类创建调用指向一个元类，如果提供了一个元类的话。然而，我们实际如何编写一个元类来定制type呢？

事实上，我们已经知道了大多数情况——用常规的Python class语句和语法来编写元类。唯一的实质区别是，Python在一条class语句的末尾自动调用它们，而且它们必须通过type超类附加到预期的接口。

基本元类

可能你能够编写的最简单元类只是带有一个__new__方法的type的子类，该方法通过运行type中的默认版本来创建类对象。像这样的一个元类__new__，通过继承自type的__new__方法而运行。它通常执行所需的任何定制并且调用type的超类的__new__方法来创建并运行新的类对象：

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
        # Run by inherited type.__call__
        return type.__new__(meta, classname, supers, classdict)
```

这个元类实际并没有做任何事情（我们可能也会让默认的type类创建类），但是它展示了将元类接入元类钩子中以定制——由于元类在一条class语句的末尾调用，并且因为type对象的__call__分派到了__new__和__init__方法，所以我们在这些方法中提供的代码可以管理从元类创建的所有类。下面是应用中的实例，将打印添加到元类和文件以便追踪：

```
class MetaOne(type):
```

```
    def __new__(meta, classname, supers, classdict):
        print('In MetaOne.new:', classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=MetaOne):    # Inherits from Eggs, instance of Meta
    data = 1                            # Class data attribute
    def meth(self, arg):                # Class method attribute
        pass

print('making instance')
X = Spam()
print('data:', X.data)
```

在这里，Spam继承自Eggs并且是MetaOne的一个实例，但是X是Spam的一个实例并且继承自它。当这段代码在Python 3.0下运行的时候，注意在class语句的**末尾**是如何调用元类的，在我们真正创建一个实例之前——元类用来处理类，并且类用来处理实例：

```
making class
In MetaOne.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02AEBA08>}
making instance
data: 1
```

定制构建和初始化

元类也可以接入__init__协议，由type对象的__call__调用：通常，__new__创建并返回了类对象，__init__初始化了已经创建的类。元类也可以用做在创建时管理类的钩子：

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('In MetaOne.new: ', classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

    def __init__(Class, classname, supers, classdict):
        print('In MetaOne init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=MetaOne):    # Inherits from Eggs, instance of Meta
    data = 1                            # Class data attribute
    def meth(self, arg):                # Class method attribute
        pass

print('making instance')
```

```
X = Spam()
print('data:', X.data)
```

在这个例子中，类初始化方法在类构建方法之后运行，但是，两者都在class语句最后运行，并且在创建任何实例之前运行：

```
making class
In MetaOne.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02AAB810>}
In MetaOne init:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02AAB810>}
...init class object: ['__module__', 'data', 'meth', '__doc__']
making instance
data: 1
```

其他元类编程技巧

尽管重新定义type超类的__new__和__init__方法是元类向类对象创建过程插入逻辑的最常见方法，其他的方案也是可能的。

使用简单的工厂函数

例如，元类根本不是真的需要类。正如我们所学习的，class语句发布了一条简单的调用，在其处理的最后创建了一个类。因此，实际上**任何可调用对象**都可以用作一个元类，只要它接收传递的参数并且返回与目标类兼容的一个对象。实际上，一个简单的对象工厂函数就像一个类一样工作：

```
# A simple function can serve as a metaclass too

def MetaFunc(classname, supers, classdict):
    print('In MetaFunc: ', classname, supers, classdict, sep='\n...')
    return type(classname, supers, classdict)

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=MetaFunc):           # Run simple function at end
    data = 1                                    # Function returns class
    def meth(self, args):
        pass

print('making instance')
X = Spam()
print('data:', X.data)
```

运行的时候，在声明class语句的末尾调用该函数，并且它返回期待的新的类对象。该函数直接捕获type对象的__call__通常会默认拦截的调用：

```
making class
In MetaFunc:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B8B6A8>}
making instance
data: 1
```

用元类重载类创建调用

由于它们涉及常规的OOP机制，所以对于元类来说，也可能直接在一条class语句的末尾捕获创建调用，通过定义type对象的__call__。然而，所需的协议有点多：

```
# __call__ can be redefined, metas can have metas

class SuperMeta(type):
    def __call__(meta, classname, supers, classdict):
        print('In SuperMeta.call: ', classname, supers, classdict, sep='\n...')
        return type.__call__(meta, classname, supers, classdict)

class SubMeta(type, metaclass=SuperMeta):
    def __new__(meta, classname, supers, classdict):
        print('In SubMeta.new: ', classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

    def __init__(Class, classname, supers, classdict):
        print('In SubMeta init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=SubMeta):
    data = 1
    def meth(self, arg):
        pass

print('making instance')
X = Spam()
print('data:', X.data)
```

当这段代码运行的时候，所有3个重新定义的方法都依次运行。这基本上就是type对象默认做的事情：

```
making class
In SuperMeta.call:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B7BA98>}
```

```
In SubMeta.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B7BA98>}
In SubMeta init:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B7BA98>}
...init class object: ['__module__', 'data', 'meth', '__doc__']
making instance
data: 1
```

用常规类重载类创建调用

前面的例子被复杂化了，事实是用元类来创建类对象，但并不产生它们自己的实例。因此，元类名查找规则与我们所习惯的方式有点不同。例如，`__call__`方法在一个对象的类中查找；对于元类，这意味着一个元类的元类。

要使用常规的基于继承的名称查找，我们可以用常规类和实例实现同样的效果。如下的输出和前面的版本相同，但是注意，`__new__`和`__init__`在这里必须有不同的名称，否则，当创建SubMeta实例的时候它们会运行，而不是随后作为一个元类调用：

```
class SuperMeta:
    def __call__(self, classname, supers, classdict):
        print('In SuperMeta.call: ', classname, supers, classdict, sep='\n...')
        Class = self.__New__(classname, supers, classdict)
        self.__Init__(Class, classname, supers, classdict)
        return Class

class SubMeta(SuperMeta):
    def __New__(self, classname, supers, classdict):
        print('In SubMeta.new: ', classname, supers, classdict, sep='\n...')
        return type(classname, supers, classdict)

    def __Init__(self, Class, classname, supers, classdict):
        print('In SubMeta init:', classname, supers, classdict, sep='\n...')
        print('...init class object:', list(Class.__dict__.keys()))

class Eggs:
    pass

print('making class')
class Spam(Eggs, metaclass=SubMeta()):          # Meta is normal class instance
    data = 1                                    # Called at end of statement
    def meth(self, arg):
        pass

print('making instance')
X = Spam()
print('data:', X.data)
```

尽管这种替代形式有效，但大多数元类通过重新定义type超类的`__new__`和`__init__`完

成它们的工作。实际上，这通常需要尽可能多的控制，并且它往往比其他方法简单。然而，我们随后将看到，一个简单的基于函数的元类往往更像是一个类装饰器一样工作，这允许元类管理实例以及类。

实例与继承的关系

由于元类以类似于继承超类的方式来制定，因此它们乍看上去有点容易令人混淆。一些关键点有助于概括和澄清这一模型：

- **元类继承自type类。**尽管它们有一种特殊的角色元类，但元类是用`class`语句编写的，并且遵从Python中有用的OOP模型。例如，就像`type`的子类一样，它们可以重新定义type对象的方法，需要的时候重载或定制它们。元类通常重新定义`type`类的`__new__`和`__init__`，以定制类创建和初始化，但是，如果它们希望直接捕获类末尾的创建调用的话，它们也可以重新定义`__call__`。尽管元类不常见，它们甚至是返回任意对象而不是`type`子类的简单函数。
- **元类声明由子类继承。**在用户定义的类中，`metaclass=M`声明由该类的子类**继承**，因此，对于在超类链中继承了这一声明的每个类的构建，该元类都将运行。
- **元类属性没有由类实例继承。**元类声明指定了一个**实例**关系，它和继承不同。由于类是元类的实例，所以元类中定义的行为应用于类，而不是类随后的实例。实例从它们的类和超类获取行为，但是，不是从任何元类获取行为。从技术上讲，实例属性查找通常只是搜索实例及其所有类的`__dict__`字典；元类不包含在实例查找中。

为了说明最后两点，考虑如下的例子：

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):          # Redefine type method
        print('In MetaOne.new:', classname)
        return type.__new__(meta, classname, supers, classdict)
    def toast(self):
        print('toast')

class Super(metaclass=MetaOne):          # Metaclass inherited by subs too
    def spam(self):                      # MetaOne run twice for two classes
        print('spam')

class C(Super):                          # Superclass: inheritance versus instance
    def eggs(self):                      # Classes inherit from superclasses
        print('eggs')                    # But not from metclasses

X = C()
X.eggs()              # Inherited from C
X.spam()              # Inherited from Super
X.toast()             # Not inherited from metaclass
```

当这段代码运行的时候，元类处理**两个**客户类的构建，并且实例继承类属性而**不是**元类属性：

```
In MetaOne.new: Super
In MetaOne.new: C
eggs
spam
AttributeError: 'C' object has no attribute 'toast'
```

尽管细节很有意义，但是，在处理元类的时候，头脑中保持大概念很重要。像我们已经在这里见到的元类将会对声明它们的每个子类自动运行。和我们在前面见到的辅助函数方法不同，这样的类将会自动获取元类所提供的任何扩展。此外，修改这样的扩展只需要在一个地方编码——元类中，这简化了所需要进行的修改。就像Python中如此众多的工具一样，元类通过除去冗余简化了维护工作。然而，要完全展示其功能，我们需要继续看一些更大的示例。

示例：向类添加方法

在本节以及下一节，我们将学习两个常见的元类示例：向一个类添加方法，以及自动装饰所有的方法。这只是元类众多用处中的两个，它们将占用本章剩余的篇幅。再一次说明，你应该在Web上查找以了解更多的高级应用。这些示例代表了元类的应用，因此它们足以说明基础知识。

此外，这两个示例都给了我们机会来对比类装饰器和元类——第一个示例比较了类扩展和实例包装的基于元类和基于装饰器的实现；第二个实例首先对元类应用一个装饰器，然后应用另一个装饰器。你将会看到，这两个工具往往是可以交换的，甚至是互补的。

手动扩展

在本章前面，我们看到了以不同方法向类添加方法来扩展类的骨干代码。正如我们所见到的，如果在编写类的时候，静态地知道额外的方法，那么简单的基于类的继承已经足够了。通过对象嵌入的组合，往往也能够实现同样的效果。然而，对于更加动态的场景，有时候需要其他的技术，辅助函数通常足够了，但元类提供了一种更加明确的结构并且减少了未来修改的成本。

让我们在这里通过实际代码把这些思想付诸实践。考虑下面示例中的手动类扩展——它向两个类添加了两个方法，在创建了它们之后：

```
# Extend manually - adding new methods to classes

class Client1:
    def __init__(self, value):
```

```
            self.value = value
        def spam(self):
            return self.value * 2

    class Client2:
        value = 'ni?'

    def eggsfunc(obj):
        return obj.value * 4

    def hamfunc(obj, value):
        return value + 'ham'

    Client1.eggs = eggsfunc
    Client1.ham = hamfunc

    Client2.eggs = eggsfunc
    Client2.ham = hamfunc

    X = Client1('Ni!')
    print(X.spam())
    print(X.eggs())
    print(X.ham('bacon'))

    Y = Client2()
    print(Y.eggs())
    print(Y.ham('bacon'))
```

这是有效的，因为方法总是在类创建之后分配给一个类，只要分配的方法是带有一个额外的第一个参数以接收主体`self`示例的函数，这个参数可以用来访问类实例中可用的状态信息，即便函数独立于类定义。

当这段代码运行的时候，我们接收到在第一个类的代码中编写的一个方法输出，以及在此后添加到类中的两个方法的输出：

```
Ni!Ni!
Ni!Ni!Ni!Ni!
baconham
ni?ni?ni?ni?
baconham
```

这种方法在独立的情况下工作得很好，并且可以在运行时任意地填充一个类。但它有一个潜在的主要缺点，对于需要这些方法的每个类，我们必须重复扩展代码。在我们的例子中，对两个类都添加两个方法还不是太繁琐，但是，在更为复杂的环境中，这种方法可能是耗时的而且容易出错。如果我们曾经忘记一致地这么做，或者我们需要修改扩展，就可能遇到问题。

基于元类的扩展

尽管手动扩展有效，但在较大的程序中，如果可以对整个一组类自动应用这样的修改，

可能会更好。通过这种方式，我们避免了对任何给定的类修改扩展的机会。此外，在单独位置编写扩展更好地支持了未来的修改——集合中的所有类都将自动接收修改。

满足这一目标的一种方式就是使用元类。如果我们在元类中编写扩展，那么声明了元类的每个类都将统一且正确地扩展，并自动地接收未来做出的任何修改。如下的代码展示了这一点：

```
# Extend with a metaclass - supports future changes better

def eggsfunc(obj):
    return obj.value * 4
def hamfunc(obj, value):
    return value + 'ham'

class Extender(type):
    def __new__(meta, classname, supers, classdict):
        classdict['eggs'] = eggsfunc
        classdict['ham'] = hamfunc
        return type.__new__(meta, classname, supers, classdict)

class Client1(metaclass=Extender):
    def __init__(self, value):
        self.value = value
    def spam(self):
        return self.value * 2

class Client2(metaclass=Extender):
    value = 'ni?'

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bacon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bacon'))
```

这一次，两个客户类都使用新的方法扩展了，因为它们是执行扩展的元类的实例。运行的时候，这个版本的输出和前面相同，我们没有做代码所做的修改，我们只是重构它们以便更整齐地封装修改：

```
Ni!Ni!
Ni!Ni!Ni!Ni!
baconham
ni?ni?ni?ni?
baconham
```

注意，这个示例中的元类仍然执行相当静态的工作：把两个已知的方法添加到声明了元类的每个类。实际上，如果我们所需要做的总是向一组类添加相同的两个方法，我们也

可以将它们编写为常规的超类并在子类中继承它们。然而，实际上，元类结构支持更多的动态行为。例如，主体类也可以基于运行时的任意逻辑配置：

```
# Can also configure class based on runtime tests

class MetaExtend(type):
    def __new__(meta, classname, supers, classdict):
        if sometest():
            classdict['eggs'] = eggsfunc1
        else:
            classdict['eggs'] = eggsfunc2
        if someothertest():
            classdict['ham'] = hamfunc
        else:
            classdict['ham'] = lambda *args: 'Not supported'
        return type.__new__(meta, classname, supers, classdict)
```

元类与类装饰器的关系：第二回合

如果本章中的示例还没有让你大脑爆炸，请记住，上一章的类装饰器常常和本章的元类在功能上有重合。这源自于如下的事实：

- 在class语句的末尾，类装饰器把类名重新绑定到一个函数的结果。
- 元类通过在一条class语句的末尾把类对象创建过程路由到一个对象来工作。

尽管这些是略有不同的模型，实际上，它们通常可实现同样的目标，虽然采用的方式不同。实际上，类装饰器可以用来管理一个类的实例以及类自身。尽管装饰器可以自然地管理类，然而，用元类管理实例有些不那么直接。元类可能最好用于类对象管理。

基于装饰器的扩展

例如，前面小节的元类示例，像创建的一个类添加方法，也可以用一个类装饰器来编写。在这种模式下，装饰器大致与元类的__init__方法对应，因为在调用装饰器的时候，类对象已经创建了。也与元类类似，最初的类类型是保留的，因为没有插入包装器对象层。如下的输出与前面的元类代码的输出相同：

```
# Extend with a decorator: same as providing __init__ in a metaclass

def eggsfunc(obj):
    return obj.value * 4

def hamfunc(obj, value):
    return value + 'ham'

def Extender(aClass):
    aClass.eggs = eggsfunc              # Manages class, not instance
    aClass.ham = hamfunc                # Equiv to metaclass __init__
    return aClass
```

```
@Extender
class Client1:                          # Client1 = Extender(Client1)
def __init__(self, value):              # Rebound at end of class stmt
    self.value = value
def spam(self):
    return self.value * 2

@Extender
class Client2:
    value = 'ni?'

X = Client1('Ni!')                      # X is a Client1 instance
print(X.spam())
print(X.eggs())
print(X.ham('bacon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bacon'))
```

换句话说，至少在某些情况下，装饰器可以像元类一样容易地管理类。反过来就不那么直接了，元类可以用来管理实例，但是只有有限的力量。下一小节将说明这点。

管理实例而不是类

正如我们已经见到的，类装饰器常常可以和元类一样充当**类管理**角色。元类往往和装饰器一样充当**实例管理**的角色，但是，这更复杂一点。即：

- *类装饰器*可以管理类和实例。
- *元类*可以管理类和实例，但是管理实例需要一些额外工作。

也就是说，某些应用可能用一种方法或另一种方法编写更好。例如，前一章中的类装饰器示例，无论何时，获取一个类实例的任意常规命名的属性的时候，它用来打印一条跟踪消息：

```
# Class decorator to trace external instance attribute fetches

def Tracer(aClass):                                     # On @ decorator
    class Wrapper:
        def __init__(self, *args, **kargs):             # On instance creation
            self.wrapped = aClass(*args, **kargs)       # Use enclosing scope name
        def __getattr__(self, attrname):
            print('Trace:', attrname)                   # Catches all but .wrapped
            return getattr(self.wrapped, attrname)      # Delegate to wrapped object
    return Wrapper

@Tracer
class Person:                                           # Person = Tracer(Person)
    def __init__(self, name, hours, rate):              # Wrapper remembers Person
        self.name = name
        self.hours = hours
```

```
        self.rate = rate                                # In-method fetch not traced
    def pay(self):
        return self.hours * self.rate

bob = Person('Bob', 40, 50)                             # bob is really a Wrapper
print(bob.name)                                         # Wrapper embeds a Person
print(bob.pay())                                        # Triggers __getattr__
```

这段代码运行的时候，装饰器使用类名重新绑定来把实例对象包装到一个对象中，该对象在如下的输出中给出跟踪行：

```
Trace: name
Bob
Trace: pay
2000
```

尽管用一个元类也可能实现同样的效果，但它似乎概念上不太直接明了。元类明确地设计来管理类对象创建，并且它们有一个为此目的而设计的接口。要使用元类来管理实例，我们必须依赖一些额外力量。如下的元类和前面的装饰器具有同样的效果和输出：

```
# Manage instances like the prior example, but with a metaclass

def Tracer(classname, supers, classdict):               # On class creation call
    aClass = type(classname, supers, classdict)         # Make client class
    class Wrapper:
        def __init__(self, *args, **kargs):             # On instance creation
            self.wrapped = aClass(*args, **kargs)
        def __getattr__(self, attrname):
            print('Trace:', attrname)                   # Catches all but .wrapped
            return getattr(self.wrapped, attrname)      # Delegate to wrapped object
    return Wrapper

class Person(metaclass=Tracer):                         # Make Person with Tracer
    def __init__(self, name, hours, rate):              # Wrapper remembers Person
        self.name = name
        self.hours = hours
        self.rate = rate                                # In-method fetch not traced
    def pay(self):
        return self.hours * self.rate

bob = Person('Bob', 40, 50)                             # bob is really a Wrapper
print(bob.name)                                         # Wrapper embeds a Person
print(bob.pay())                                        # Triggers __getattr__
```

这也有效，但是它依赖于两个技巧。首先，它必须使用一个简单的函数而不是一个类，因为`type`子类必须附加给对象创建协议。其次，必须通过手动调用`type`来手动创建主体类；它需要返回一个实例包装器，但是元类也负责创建和返回主体类。其实，在这个例子中，我们将使用元类协议来模仿装饰器，而不是按照相反的做法。由于它们都在一条`class`语句的末尾运行，所以在很多用途中，它们都是同一方法的变体。元类版本运行的时候，产生与装饰器版本同样的输出：

```
Trace: name
Bob
Trace: pay
2000
```

你应该自己研究这两个示例版本，以权衡其利弊。通常，元类可能更适合于类管理，因为它们就设计为如此。类装饰器可以管理实例或类，然而，它们不是更高级元类用途的最佳选择（我们没有足够篇幅在本书中介绍，如果你在阅读完本章后，还想学习关于装饰器和元类的更多内容，请在Web上搜索或参阅Python标准手册的内容）。下一小节用一个更为常见的例子来结束本章，自动对一个类的方法应用操作。

示例：对方法应用装饰器

正如我们在前一小节中见到的，由于它们都在一条`class`语句的末尾运行，所以元类和装饰器往往可以**互换地**使用，虽然语法不同。在这二者之间的选择，在很多情况下是随意的。也可能将二者**组合**起来使用，作为互补的工具。在本小节中，我们将展示一个示例，它就是这样的组合——对一个类的所有方法应用一个函数装饰器。

用装饰器手动跟踪

在前面的一章中，我们编写了两个函数装饰器，其中之一跟踪和统计对一个装饰函数的调用，另一个计时这样的调用。它们采用各种形式，其中的一些对于函数和方法都适用，另一些并不适用。下面把两个装饰器的最终形式收入一个模块文件中，以便重用或引用：

```
# File mytools.py: assorted decorator tools

def tracer(func):                           # Use function, not class with __call__
    calls = 0                               # Else self is decorator instance only
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return onCall

import time
def timer(label='', trace=True):                # On decorator args: retain args
    def onDecorator(func):                      # On @: retain decorated func
        def onCall(*args, **kargs):             # On calls: call original
            start = time.clock()                # State is scopes + func attr
            result = func(*args, **kargs)
            elapsed = time.clock() - start
            onCall.alltime += elapsed
            if trace:
               format = '%s%s: %.5f, %.5f'
```

```
            values = (label, func.__name__, elapsed, onCall.alltime)
            print(format % values)
            return result
        onCall.alltime = 0
        return onCall
    return onDecorator
```

正如我们在上一章了解到的，要手动使用这些装饰器，我们直接从模块导入它们，并且在想要跟踪或计时的每个方法前编写@装饰语法：

```
from mytools import tracer

class Person:
    @tracer
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @tracer
    def giveRaise(self, percent):                # giveRaise = tracer(giverRaise)
        self.pay *= (1.0 + percent)              # onCall remembers giveRaise

    @tracer
    def lastName(self):                          # lastName = tracer(lastName)
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)                               # Runs onCall(sue, .10)
print(sue.pay)
print(bob.lastName(), sue.lastName())            # Runs onCall(bob), remembers lastName
```

这段代码运行时，我们得到了如下输出——对装饰方法的调用指向了拦截逻辑，并且随后委托调用，因为最初的方法名已经绑定到了装饰器：

```
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```

用元类和装饰器跟踪

前一小节的手动装饰方法是有效的，但是它需要我们在想要跟踪的**每个**方法前面添加装饰语法，并且在不再想要跟踪的使用后删除该语法。如果想要跟踪一个类的每个方法，在较大的程序中，这会变得很繁琐。如果我们可以对一个类的所有方法自动地应用跟踪装饰器，那将会更好。

有了元类，我们确实可以做到——因为它们在构建一个类的时候运行，它们是把装饰包装器添加到一个类方法中的自然地方。通过扫描类的属性字典并测试函数对象，我们可以通过装饰器自动运行方法，并且把最初的名称重新绑定到结果。其效果与装饰器的自动方法名重新绑定是相同的，但是，我们可以更全面地应用它：

```
# Metaclass that adds tracing decorator to every method of a client class

from types import FunctionType
from mytools import tracer

class MetaTrace(type):
    def __new__(meta, classname, supers, classdict):
        for attr, attrval in classdict.items():
            if type(attrval) is FunctionType:                    # Method?
               classdict[attr] = tracer(attrval)                 # Decorate it
        return type.__new__(meta, classname, supers, classdict)  # Make class

class Person(metaclass=MetaTrace):
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())
```

当这段代码运行的时候，结果与前面是相同的——方法的调用将首先指向跟踪装饰器以跟踪，然后传递到最初的方法：

```
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```

我们这里看到的就是装饰器和元类组合工作的结果——在类创建的时候，元类自动把函数装饰器应用于每个方法，并且函数装饰器自动拦截方法调用，以便在此输出中打印出跟踪消息。这一组合能够有效，得益于两种工具的通用性。

把任何装饰器应用于方法

前面的元类示例只对一个特定的函数装饰器有效，即跟踪。然而，将这个通用化以把**任何**装饰器应用到一个类的所有方法，实际的意义不大。我们所要做的是，添加一个外围作用域层，以保持想要的装饰器，这很像是我们在上一章对装饰器所做的。如下的示例，编写了这样的一个泛化，然后使用它再次应用跟踪装饰器：

```
# Metaclass factory: apply any decorator to all methods of a class

from types import FunctionType
from mytools import tracer, timer

def decorateAll(decorator):
    class MetaDecorate(type):
        def __new__(meta, classname, supers, classdict):
            for attr, attrval in classdict.items():
                if type(attrval) is FunctionType:
                    classdict[attr] = decorator(attrval)
            return type.__new__(meta, classname, supers, classdict)
    return MetaDecorate

class Person(metaclass=decorateAll(tracer)):    # Apply a decorator to all
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())
```

当这段代码运行的时候，输出再次与前面的示例相同——最终我们仍然在一个客户类中用跟踪器函数装饰器装饰了每个方法，但是，我们以一种更为通用的方式做到了这点：

```
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```

现在，要对方法应用一种不同的装饰器，我们只要在类标题行替换装饰器名称。例如，要使用前面介绍的计时器函数装饰器，定义类的时候，我们可以使用如下示例标题行的最后两行中的任何一个——第一个接收了定时器的默认参数，第二个指定了标签文本：

```
class Person(metaclass=decorateAll(tracer)):                    # Apply tracer

class Person(metaclass=decorateAll(timer())):                   # Apply timer, defaults

class Person(metaclass=decorateAll(timer(label='**'))):         # Decorator arguments
```

注意，这种方法不支持对每个方法不同的非默认装饰器参数，但是，它可以传递到装饰器参数中以应用到所有方法，就像这里所做的一样。为了进行测试，使用这些元类声明的最后一个来应用定时器，并且在脚本的末尾添加如下的行：

```
# If using timer: total time per method

print('-'*40)
print('%.5f' % Person.__init__.alltime)
print('%.5f' % Person.giveRaise.alltime)
print('%.5f' % Person.lastName.alltime)
```

新的输出如下所示——现在，元类把方法包装到了定时器装饰器中，以便我们可以说出针对类的每个方法的每次调用花费多长时间：

```
**__init__: 0.00001, 0.00001
**__init__: 0.00001, 0.00002
Bob Smith Sue Jones
**giveRaise: 0.00001, 0.00001
110000.0
**lastName: 0.00001, 0.00001
**lastName: 0.00001, 0.00002
Smith Jones
----------------------------------------
0.00002
0.00001
0.00002
```

元类与类装饰器的关系：第三回合

类装饰器也与元类有交叉。如下的版本，用一个类装饰器替换了前面的示例中的元类。它定义并使用一个**类装饰器**，该装饰器把一个**函数装饰器应用于**一个类的所有方法。然而，前一句话听起来更像是禅语而不像是技术说明，这所有的工作相当自然——Python的装饰器支持任意的嵌套和组合：

```
# Class decorator factory: apply any decorator to all methods of a class

from types import FunctionType
from mytools import tracer, timer

def decorateAll(decorator):
    def DecoDecorate(aClass):
        for attr, attrval in aClass.__dict__.items():
            if type(attrval) is FunctionType:
                setattr(aClass, attr, decorator(attrval))       # Not __dict__
```

```
            return aClass
        return DecoDecorate

@decorateAll(tracer)                        # Use a class decorator
class Person:                               # Applies func decorator to methods
    def __init__(self, name, pay):          # Person = decorateAll(..)(Person)
        self.name = name                    # Person = DecoDecorate(Person)
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Bob Smith', 50000)
sue = Person('Sue Jones', 100000)
print(bob.name, sue.name)
sue.giveRaise(.10)
print(sue.pay)
print(bob.lastName(), sue.lastName())
```

当这段代码运行的时候，类装饰器把跟踪器函数装饰器应用于每个方法，并且在调用时产生一条跟踪消息（输出和本示例前面的元类版本相同）：

```
call 1 to __init__
call 2 to __init__
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```

注意，类装饰器返回最初的、扩展的类，而不是其包装器层（当返回包装示例对象的时候更为常见）。就像是元类版本一样，我们保留了最初的类的类型——Person的一个实例，而不是某个包装器类的实例。实际上，这个类装饰器只是处理了类创建，实例创建调用根本没有拦截。

这种区别对于需要对实例进行类型测试以产生最初的类而不是包装器的程序有影响。当扩展一个类而不是一个实例的时候，类装饰器可以保持最初的类类型，类的方法不是它们最初的函数，因为它们绑定到了装饰器，但是这在实际中并不重要，并在元类替代方案中也是如此。

还要注意到，和元类版本一样，这种结构不支持每个方法不同的函数装饰器参数，但是，如果它们适用于所有方法的话，可以处理这种参数。例如，要使用这种方法应用计时器装饰器，下面声明行的最后两个中的任何一个就够了，如果类定义的代码和前面一样的话——第一个使用装饰器参数默认，第二个显式地提供了一个参数：

```
@decorateAll(tracer)                    # Decorate all with tracer
@decorateAll(timer())                   # Decorate all with timer, defaults
@decorateAll(timer(label='@@'))         # Same but pass a decorator argument
```

和前面一样，让我们使用这些装饰器行的最后一个，并且在脚本的末尾添加如下代码，以用一种不同的装饰器来测试示例：

```
# If using timer: total time per method

print('-'*40)
print('%.5f' % Person.__init__.alltime)
print('%.5f' % Person.giveRaise.alltime)
print('%.5f' % Person.lastName.alltime)
```

同样的输出出现了，对于每个方法，我们针对每次调用和所有调用获取了计时数据，但是，我们已经给计数器装饰器传递了一个不同的标签参数：

```
@@__init__: 0.00001, 0.00001
@@__init__: 0.00001, 0.00002
Bob Smith Sue Jones
@@giveRaise: 0.00001, 0.00001
110000.0
@@lastName: 0.00001, 0.00001
@@lastName: 0.00001, 0.00002
Smith Jones
----------------------------------------
0.00002
0.00001
0.00002
```

正如你所看到的，元类和类装饰器不仅常常可以交换，而且通常是互补的。它们都对于定制和管理类和实例对象，提供了高级但强大的方法，因为这二者最终都允许我们在类创建过程中插入代码。尽管某些高级应用可能用一种方式或另一种方式编码更好，但在很多情况下，我们选择或组合这两种工具的方法，很大程度上取决于你。

“可选的”语言功能

我在本章开始处引用了一句话，提到元类不是99%的Python程序员都感兴趣的，用以强调它们相对难以理解。这句话不是很准确，只是用一个数字来说明。

说这句话的人是我最初使用Python时的一个朋友，并且，我并不是想不公平地嘲笑某人。另外，实际上，在这本书中，对于语言功能的灰色性，我也常常做出这样的表述。

然而问题在于，这样的语句真的只是适用于单独工作的人而且只是那些可能使用他们自己曾经编写的代码的人。只要一个组织中的**任何人**使用了一项“可选的”高级语言功能，它就不再是可选的——它有效地施加于组织中的**每个人**身上。对于你在系统中所使用的外部开发的软件来说，也是如此——如果软件的作者使用了一项高级功能，它对你来说完全不再是可选的，因为你必须理解该功能以便使用或修改代码。

这种观察适用于在本章开始处列出的所有高级工具——装饰器、特性、描述符、元类，等等。如果与你一起工作的任何人或任何程序用到了它们，它们自动地变成所需的知识基础的一部分。也就是说，**没有什么是真正“可选的”**。我们当中的大多数人不会去挑选或选择。

这就是为什么一些Python前辈（包括我自己）有时候悲叹，Python似乎随着时间的流逝变得更大并且更复杂了。由老手添加的新功能似乎已经增加了对初学者的智力障碍。尽管Python的核心思想，例如动态类型和内置类型，基本保持相同。它的高级附加功能，也变成了任何Python程序员所必须阅读的。正因为此，我选择在这里介绍这些主题，尽管在前面的版本中并没介绍它们。如果高级内容就在你必须理解的代码之中，那么省略它们是不可能的。

另外一方面，很多新的学习者可以挑选所需的高级话题。坦率地讲，应用程序员可能会把大多数的时间花在**处理库和扩展**上，而不是高级的并且有时候颇为不可思议的语言功能上。例如，本书的后续篇《Programming Python》，处理大多数把Python与应用库结合起来完成的任务，例如GUI、数据库以及Web，而不介绍深奥的语言工具。

这一增长的优点是，Python已经变得更为**强大**。当我们用好它的时候，像装饰器或元类这样的工具不仅毫无辩驳的“酷”，而且允许有创意的程序员来构建更为灵活和有用的API供其他程序员使用。正如我们已经看到的，它们也可以为封装和维护问题提供很好的解决方案。

是否使用所需的Python知识的潜在扩展，取决于你。遗憾的是，一个人的技能水平往往默认决定了这个问题——很多高级的程序员喜欢较为高级的工具，并且往往忘记它们对其他阵营的影响。幸运的是，这不是绝对的；好的程序员也理解**简单是最好的工程**，并且高级工具也应该只在需要的时候使用。对任何编程语言来说，这都是成立的，但是，特别是在像Python这样的语言中，它作为一种扩展工具广泛地展示给新的和初学的程序员。

如果你仍然不接受这一观点，别忘了，有很多Python用户不习惯基本的OOP和类。相信我的判断，我曾经遇到过数以千计这样的人。基于Python的系统需要它们的用户掌握元类、装饰器之间的细微差别，并且可能由此扩展它们的市场预期。

本章小结

在本章中，我们学习了元类并且介绍了它们的实际使用示例。元类允许我们接入Python的类创建协议，以便管理和扩展用户定义的类。由于它们使这一过程自动化了，因此它们可以为API编写者提供更好的解决方案，而且手动编码或使用辅助函数。由于它们封装了这样的代码，所以它们可以比其他的方法更好地减少维护成本。

在此过程中，我们还看到了类装饰器与元类的角色往往是如何交叉的：由于它们都在一条`class`语句的末尾运行，因而它们有时候可以互换地使用。类装饰器可以用来管理类和实例对象，元类也可以，尽管它们更直接地以类为目标。

由于本章介绍了一个高级话题，因此我们将通过一些练习题来回顾基础知识（如果你已经在关于元类的本章中读到了这里，你应该已经接受额外的奖励）。这是本书的最后一部分，我们将给出最后一部分的练习。确保查看后面的附录中关于安装的提示，以及前面各部分的练习的解答。

一旦完成了练习，你就真正完成了本书。既然你知道了Python里里外外的知识，那么下一步应该是选择接收它，即研究库、技巧，以及你的应用领域所能用到的工具。由于Python应用如此广泛，你可能会找到丰富的资源以在几乎可以考虑到的所有应用领域中使用它，从GUI、Web、数据库到数值计算、机器人以及系统管理。

Python由此变得真正有趣，但是这也只是本书的介绍到此结束，而另一段故事开始了。阅读完本书之后，要获取提示，可以参见前言中的推荐资料部分的列表。祝你好运。并且当然，“总是要看到生命的阳光灿烂的一面！”

本章习题

1. 什么是元类？
2. 如何声明一个类的元类？
3. 在管理类方面，类装饰器如何与元类重叠？
4. 在管理实例方面，类装饰器如何与元类重叠？

习题解答

1. 元类是用来创建一个类的类。常规的类默认的是`type`类的实例。元类通常是`type`类的子类，它重新定义了类创建协议方法，以便定制在一条`class`语句的末尾发布的类创建调用；它通常会重定义`__new__`和`__init__`方法以接入类创建协议。元类也可

以以其他的方式编码——例如，作为简单函数——但是它们负责为新类创建和返回一个对象。

2. 在Python 3.0及其以后的版本中，在类标题栏使用一个关键字参数：`class C(metaclass=M)`。在Python 2.X中，使用类属性：`__metaclass__= M`。在Python 3.0中，类标题栏也可以在`metaclass`关键字参数之前命名常规的超类（例如，基类）。

3. 由于二者都是在一条`class`语句的末尾自动触发，因此类装饰器和元类都可以用来管理类。装饰器把一个类名重新绑定到一个可调用对象的结果，而元类把类创建指向一个可调用对象，但它们都是可以用作相同目的的钩子。要管理类，装饰器直接扩展并返回最初的类对象。元类在创建一个类之后扩展它。

4. 由于二者都是在一条`class`语句的末尾自动触发，因此类装饰器和元类都可以用来管理类实例，通过插入一个包装器对象来捕获实例创建调用。装饰器把类名重新绑定到一个可调用对象，而该可调用对象在实例创建时运行以保持最初的类对象。元类可以做同样的事情，但是，它们必须也创建类对象，因此，它们用在这一角色中更复杂一些。

第九部分

附录

附录A

安装和配置

本附录提供其他安装和配置的细节，新接触这类话题的人可以参考这些资源。

安装Python解释器

因为需要用Python解释器运行Python脚本，所以使用Python的第一步通常就是安装Python。除非你的机器上已有一个Python，不然，你就得取得最新版Python，在计算机上安装和配置。每台机器只需安装和配置一次，如果你是运行冻结二进制文件（第2章介绍过）或自安装系统，那就完全不需要这样做了。

Python已经存在了吗

做任何事之前，应该检查机器上是否已有最新版本的Python。如果你用的是Linux、Mac OS X以及一些UNIX系统，Python可能已经安装在你的计算机上，尽管它可能和最新版本相比已经落后了一两个版本。可通过如下方式来检查：

- 在Windows上，查看“开始”→“所有程序”菜单中（位于屏幕左下方）是否有Python。
- 在Mac OS X下，打开一个Terminal窗口(Applications→Utilities→Terminal)，并且在提示符下输入**`Python`**。
- 在Linux和Unix上，在shell提示符下（有时被称作终端窗口）输入**`python`**，看看会发生什么事。此外，也可以在常见的位置搜索“python”：*/usr/bin*、*/usr/local/bin*等。

如果找到Python，要确保它是最近的一个版本。尽管任何较新的Python版本对本书中的大多数内容都适用，本书主要关注Python 3.0和Python 2.6，因此，你可能想要安装这两个版本之一来运行本书中的示例。

提到版本，如果你初次接触Python并且不需要处理已有的Python 2.X代码，我建议你从Python 3.0及其以后的版本开始；否则，应该使用Python 2.6。一些流行的基于Python的系统仍然使用旧版本（Python 2.5仍然很普遍），因此，如果你要用已有的系统工作，确保根据你的需要来选用版本；下一小节将介绍从哪里可以获取不同的Python版本。

从哪里获取Python

如果找不到Python，就需要自行安装。幸运的是，Python是开源系统，可在Web上免费获取，而且在大多数平台上安装都很简单。

你总是可以从Python的官方网站*http://www.python.org*获取最新、最好的标准Python版本。寻找网页上的Downloads链接，然后，选择所需平台的版本。你会发现预创建的Windows的自安装文件（点击文件图标就能安装）、针对Mac OS X的安装程序光盘镜像、完整的源代码包（通常在Linux、Unix或OS X机器上编译从而生成解释器），等等。

尽管如今Python是Linux上的标准，我们还是可以在Web上找到针对Linux的RPM（用*rpm*解压它们）。Python的Web站点也连接到站内或站外的各个页面，那里维护了针对其他平台的版本。Google Web搜索是找到Python包的另一种不错的方式。在这些平台中，我们可以找到针对iPod、Palm手机、Nokia手机、PlayStation和PSP、Solaris、AS/400和Windows Mobile的预编译的Python。

如果你发现自己在Windows机器上渴望一个UNIX环境，那么，可能对于安装*Cygwin*及其Python版本感兴趣（参见*http://www.cygwin.com*）。Cygwin是一个GPL许可的库和工具集，它在Windows机器上提供了完整的UNIX功能，并且它包含一个预编译的Python，它可以使用所提供的所有UNIX工具。

你也会在Linux CD发行版中找到Python。也许是随附在某些产品和计算机系统上，或者和其他Python书籍在一起。这些通常都会比当前版本落后，但通常不会落后太多。

此外，我们可以在其他的免费和商业开发包中找到Python。例如，ActiveState公司将Python作为其*ActivePython*包的一部分。这个包把标准Python与以下工具结合起来：支持Windows开发的PyWin32，一个名为PythonWin的IDE（第3章介绍过），以及其他常用的扩展包。Python如今还放入了*Enthought* Python包中，这个包瞄准了科学计算的需求，

而且还放入了*Portable Python*，它预配置来直接从一个便携设备启动。请在Web中搜索以了解更多细节。

最后，如果你对其他Python的实现感兴趣，可以搜索网络，看一看Jython（Python的Java实现）以及IronPython（Python的C#/.NET实现），而它们在第2章都介绍过。这些系统的安装说明不在本书的范围之内。

安装步骤

下载Python后，需要进行安装。安装步骤是与平台相关的，这里主要介绍安装Python平台的一些要点。

Windows

在Windows上，Python是自安装的MSI程序文件，只要双击文件图标，在每个提示文字下回答“Yes”或“Next”，就可执行默认安装。默认安装包括了Python的文档集以及tkinter（在Python 2.6中叫做Tkinter）、shelve数据库和IDLE GUI的支持。Python 3.0和Python 2.6一般是安装在目录*C:\Python30*和*C:\Python26*下的，这在安装时可进行修改。

为了方便，安装之后，Python会出现在“开始”→“所有程序”菜单中。Python的菜单有五个项目，可以快捷地打开常见的任务：打开IDLE用户界面、阅读模块文档、打开交互模式会话、在网页浏览器中阅读Python的标准手册以及卸载。大多数动作都涉及了本书各处所提到的概念细节。

在Windows上安装后，Python会自动注册，在单击Python文件图标时，打开Python文件程序（第3章谈到过这种程序启动技术）。也有可能在Windows上通过源代码编译创建Python，但通常并不这样做。

Windows Vista用户要注意：当前Vista版本的安全特性修改了使用MSI安装文件的一些规则。如果Python安装程序无法使用，或者没有把Python放在机器上的正确位置，可以参考本附录中边栏部分寻求帮助。

Linux

在Linux上，Python可能是一个或多个RPM文件，按通常的方式将其解压（更多细节参考RPM的manpage）。根据下载的RPM，Python本身也许是一个文件，而另一个是tkinter GUI和IDLE环境的支持文件。因为Linux是类UNIX系统，下一段也同样适用。

UNIX

在UNIX系统上，Python通常是以C源代码包编译而成。这通常只需解压解文件，运行简单的`config`和`make`命令。Python会根据其编译所在的系统，自动配置其创建流

程。尽管这样，要确定你看过了包中的*README*文件从而了解这个流程的细节。因为Python是开放源代码的，其源代码可以免费使用和分发。

在其他平台上，这些细节可能大不一样：例如，要替PalmOS安装Python的Pippy移植版本，你的PDA就得有hotsync操作才行，而Python对Sharp Zaurus Linux PDA来讲，会有一个或多个*.ipk*文件，你只需执行它们就能安装。不过，可执行文件形式和源代码形式的额外安装程序都有完整说明，我们就在这里跳过其更深入的细节。

Windows Vista的Python MSI安装程序

在我编写本书时，Python的Windows自安装程序是*.msi*安装文件。这个格式在Windows XP上工作正常（只需对该文件进行双击，它就会运行），但是在某些Windows Vista版本上可能有些问题。特别是，单击MSI安装程序会使Python安装到机器的C盘根目录上，而不是正确的*C:\PythonXX*。虽然Python在根目录也能工作，但这并不是正确的安装位置。

这是与Vista安全相关的话题。简而言之，MSI文件并不是真正的可执行文件，所以不会正确地继承管理员权限，即使是由administrator用户执行。事实上，MSI文件是通过Windows注册表运行的，其文件名会和MSI安装程序相关联。

这个问题似乎是特定于Python的或者特定于Vista版本的。例如，在一款较新的笔记本上，Python 2.6和Python 3.0的安装都没有问题。要在基于Vista的OQO掌上电脑上安装Python 2.5.2，得使用命令行，强制得到所需要的管理员权限。

如果Python没有安装在正确的位置，下面是解决办法：依次选择“开始”、“所有程序”、“附件”，在“命令提示符”右击鼠标，选择“以系统管理员身份运行”，然后在访问控制对话框中选择“继续”。现在，在“命令提示符”窗口，输入cd命令，改变到Python MSI安装文件所在目录（例如，**cd C:\user\downloads**），然后，输入**msiexec /i python-2.5.1.msi**命令，手动运行MSI安装程序。最后，按照一般的GUI交互窗口来完成安装。

当然，这个行为会随时间而发生改变。以后的Vista版本中，这个流程也许就不需要了，而且可能还有其他可行的方法（例如，如果有胆量的话，也可关闭Vista的安全机制）。此外，Python最终会提供不同格式的自安装程序也是有可能的，从而以后解决这个问题——例如，提供真正的可执行文件。尝试任何其他的安装方法前，一定要单击安装程序的图标来试一下，看是不是能够正确运作。

配置Python

安装好Python后，要配置一些系统设置，改变Python执行代码的方式（如果你刚开始使用这个语言，完全可以跳过这一节。对于基本的程序来说，通常没必要进行任何系统设置的修改）。

一般来说，Python解释器各部分的行为能够通过环境变量设置和命令行选项来配置。本节我们会简单看一看Python环境变量和Python命令行选项，但要获得更多细节参考其他文档资源。

Python环境变量

环境变量（有些人称为shell变量或DOS变量）存在于Python之外，可用于给定的计算机上定制解释器每次运行时的行为。Python识别一些环境变量的设置，但只有少数是常用的，值得在这里进行说明。表A-1是Python相关的主要环境变量的设置。

表A-1：重要环境变量

变量	角色
PATH（或path）	系统shell的搜索路径（查找“python”）
PYTHONPATH	Python模块的搜索路径（用来导入）
PYTHONSTARTUP	Python交互模式启动文件的路径
TCL_LIBRARY、TK_LIBRARY	GUI扩展包的变量（tkinter）

这些变量使用起来都很直接，这里有一些建议。

PATH

PATH设置列出一组目录，这些目录是操作系统用来搜索可执行程序的。一般来说，应该包含Python解释器所在的目录（UNIX上的*python*程序或Windows上的*python.exe*）。如果你打算在Python所在目录下工作，或者在命令行输入完整的Python路径，就不需要设置这个变量。例如，在Windows中，如果你在运行任何代码前，都要执行**cd C:\Python30**（来到Python所在目录），或者总是输入**C:\Python30\python**（给出完整路径）而不只是python。此外，PATH设置多半是和命令行启动程序有关的，通过图标点击和IDE启动时，通常就没有什么关系了。

PYTHONPATH

PYTHONPATH设置的角色类似于PATH：当你在程序中导入模块文件时，Python解释器会参考PYTHONPATH变量，找出模块文件的位置。使用时，这个变量会设置成一个平台特定的目录名的列表。在UNIX头是以冒号分隔，而Windows上则是以分号间

隔。在通常情况下，这份清单只包含了你自己的源代码目录。其内容合并到了sys.path模块导入搜索路径中，以及脚本的目录、任何路径文件设置以及标准库目录。

除非你要执行跨目录的导入，否则不用设置这个变量，因为Python会自动搜索程序顶层文件的主目录，只有当模块需要导入存在于不同目录的另一个模块时，才需要这个设置。参见本附录稍后对于*.pth*路径文件的介绍，它作为PYTHONPATH的一个替代方案。对于模块搜索路径的更多介绍，请参阅第21章。

PYTHONSTARTUP

如果PYTHONSTARTUP设为Python程序代码的路径名，每当启动交互模式解释器时，Python就会自动执行这个文件的代码，好像是在交互模式命令行中输入它一样。这很少使用，但是当通过交互模式工作时，要确保一定会加载某些工具，这样很方便，可以省去导入。

tkinter*设置*

如果想使用tkinter GUI工具集（在Python 2.6中叫Tkinter），可能要把表A-1的两个GUI变量，设成Tcl和Tk系统的源代码库的目录名（很像PYTHONPATH）。然而，这些设置在Windows系统上并不需要（tkinter会随Python一起安装），如果Tcl和Tk位于标准目录中，通常也是不需要的。

注意，因为这些环境设置（以及*.pth*文件）都位于Python外部，所以什么时候设置它们通常是无所谓的。你可以在Python安装之前或之后设置，只要在Python实际运行前按照你的需要设置过就可以了。

获得Linux上tkinter（和IDLE）GUI的支持

第2章所提到的IDLE接口是Python tkinter GUI程序。tkinter是GUI工具集，而且是Windows和其他平台上Python的标准组件。不过，在某些Linux系统上，底层GUI库可能不是标准的安装组件。要在Linux上让Python新增GUI功能，可以试着运行**yumt kinter**命令来自动安装tkinter底层链接库。这样应该适用于具有*yum*安装程序的Linux发行版上（以及一些其他的系统）。

如何设定配置选项

设置Python相关环境变量的方式以及该设置成什么，取决于你所使用计算机的类型。同样要记住，不用马上把它们全部都设置好。尤其是，如果你使用的是IDLE（第3章所述），并不需要事先配置。

但是，假设你在机器上的*utilities*和*package1*目录中有一些有用的模块文件，而你想从其

他目录中的文件导入这些模块。也就是说，要从*utilities*目录加载名为*spam.py*的文件，则要能够在计算机上其他位置的另一个文件中这么写：

```
import spam
```

为了让它能够工作，你得配置模块搜索路径，以引入包含*spam.py*的目录。下面是这个过程中的一些技巧。

UNIX/Linux shell变量

在UNIX系统上，设置环境变量的方式取决于你使用的shell。在*csh* shell下，你可以在*.cshrc*或*.login*文件中增加下面的行，来设置Python模块的搜索路径：

```
setenv PYTHONPATH /usr/home/pycode/utilities:/usr/lib/pycode/package1
```

这是告诉Python，在两个用户定义的目录中寻找要导入的模块。但是，如果你使用*ksh* shell，此设置会出现在*.kshrc*文件内，看起来就像这样：

```
export PYTHONPATH="/usr/home/pycode/utilities:/usr/lib/pycode/package1"
```

其他shell可能使用不同（但类似）的语法。

DOS变量（Windows）

如果你在使用MS-DOS，或旧版Windows，可能需要在*C:\autoexec.bat*文件中新增一个环境变量配置命令，重启电脑，让修改生效。这类机器上的配置命令有DOS独特的语法：

```
set PYTHONPATH=c:\pycode\utilities;d:\pycode\package1
```

你也可以在DOS终端窗口中输入类似的命令，这样的设置只能在那个终端窗口中有效。修改*.bat*文件则可以永久的修改，对于所有的程序都有效。

其他Windows选项

在新的Windows中，可以通过系统环境变量GUI设置PYTHONPATH和其他变量，而不用编译文件或重启。在XP上，选择“控制面板”→“系统”→“高级”标签，然后单击“环境变量”按钮来编辑或新增变量（PYTHONPATH通常是用户的变量）。使用前面的DOS set命令中给出的相同变量名和值语法。Vista上的过程是类似的，但是可能必须一路验证操作。

不需重新启动机器，不过如果Python开着，要记得重启它，从而让它也能使用你的修改（只在Python启动时才配置其路径）。如果在一个Windows命令提示符窗口中工作，可能需要重新启动并选择修改。

Windows注册表

如果你是有经验的Windows用户，也可以使用注册表编辑器来配置模块搜索路径。选择“开始”→“运行”，然后输入**regedit**。假设你的机器上有这个注册表工具，你就能浏览Python的项目，然后进行修改。不过，这是脆弱且易出错的方法，除非你非常熟悉注册表，不然建议使用其他方法（实际上，这类似于对你的计算机做脑手术，因此要慎重）。

路径文件

最后，如果你选择通过*.pth*文件扩展模块搜索路径，而不是使用PYTHONPATH变量，就可以改用编写文本文件，在Windows中，看起来就像这样（文件*C:\Python30\mypath.pth*）。

```
c:\pycode\utilities
d:\pycode\package1
```

其内容会随平台不同而各不相同，而它的容器目录也会随平台和Python版本而各不相同。Python在启动时会自动定位这个文件。

路径文件中的目录名，可以是绝对或相对于含有路径文件的目录。*.pth*文件可以有多个（所有目录都会加进来），而*.pth*文件可以出现在各种平台特定的以及版本特定的、自动检查的目录中。一般情况下，一个以Python N.M发布的Python版本，在Windows系统上在*C:\PythonNM*和*C:\PythonNM\Lib\site-packages*中查找路径文件，在UNIX和Linux上则在*/usr/local/lib/pythonN.M/site-packages*和*/usr/local/lib/site-python*中。关于使用路径文件配置sys.path导入搜索路径的更多介绍，参见第21章。

因为这些设置通常都是可选的，而且本书不是介绍操作系统shell的书，所以更多的细节请参考其他资源。参考系统shell的说明，或其他文档来了解更多的信息。此外，如果你不清楚你的设置应该是什么，可以询问系统管理员或本地的专家来获取帮助。

Python命令行选项

当我们从一个系统命令行启动Python的时候（即shell提示符），可以传入各种选项标志来控制Python如何运行。和系统范围的环境变量不同，每次运行脚本的时候，命令行选项可能不同。Python 3.0中的一个Python命令行调用的完整形式如下所示（Python 2.6中大致相同，只是一些选项不同）：

```
python [-bBdEhiOsSuvVWx?] [-c command | -m module-name | script | - ] [args]
```

大多数命令行只是使用这个形式的*script*和*args*部分，来运行程序的源文件，并带有供

程序自身使用的参数。为了说明这点，考虑脚本文件*main.py*，它打印出作为sys.argv可供脚本使用的命令行参数列表：

```
# File main.py
import sys
print(sys.argv)
```

在下面的命令行中，python和main.py都可以是完整的目录路径，并且3个参数(a b -c)用于出现在sys.argv列表中的脚本。sys.argv中的第一项总是脚本文件的名称：

```
c:\Python30> python main.py a b -c              # Most common: run a script file
['main.py', 'a', 'b', '-c']
```

其他代码格式化规范选项允许我们指定Python代码：在命令行自身上运行（-c），接受代码以从标准输入流运行（一个-意味着从一个管道或重定向输入文件读取），等等：

```
c:\Python30> python -c "print(2 ** 100)"        # Read code from command argument
1267650600228229401496703205376

c:\Python30> python -c "import main"            # Import a file to run its code
['-c']

c:\Python30> python - < main.py a b -c          # Read code from standard input
['-', 'a', 'b', '-c']

c:\Python30> python - a b -c < main.py          # Same effect as prior line
['-', 'a', 'b', '-c']
```

-m代码规范在Python的模块查找路径（sys.path）上定位一个模块，并且将其作为顶级脚本运行（作为模块__main__）。在这里省略了“.py”后缀，因为文件名是一个模块：

```
c:\Python30> python -m main a b -c              # Locate/run module as script
['c:\\Python30\\main.py', 'a', 'b', '-c']
```

-m选项还支持使用相对导入语法来运行包中的模块，以及位于*.zip*包中的模块。这个开关通常用来运行pdb调试器，并且针对一个脚本调用而不是交互来从一个命令行配置profiler模块，尽管这种用法在Python 3.0中有了一些变化（配置似乎受到了Python 3.0中移除execfile的影响，并且pdb在新的Python 3.0 io模块中划入了冗余输入/输出代码）：

```
c:\Python30> python -m pdb main.py a b -c                    # Debug a script
--Return--
> c:\python30\lib\io.py(762)closed()->False
-> return self.raw.closed
(Pdb) c

c:\Python30> C:\python26\python -m pdb main.py a b -c        # Better in 2.6?
> c:\python30\main.py(1)<module>()
-> import sys
```

```
(Pdb) c
c:\Python30> python -m profile main.py a b -c          # Profile a script
c:\Python30> python -m cProfile main.py a b -c         # Low-overhead profiler
```

紧跟在“python”之后和计划要运行的代码之前，Python接受了控制器自身行为的额外参数。这些参数由Python自身使用，并且对于将要运行的脚本没有意义。例如，-O以优化模式运行Python，-u强制标准流为unbuffered，而-i在运行一段脚本后进入交互模式：

```
c:\Python30> python -u main.py a b -c          # Unbuffered output streams
```

Python 2.6还支持额外的选项以提升对Python 3.0的兼容性（-3，-O），并且检测制表符缩进用法的不一致性，而这在Python 3.0中总是会检测并报告的（-t；参见第12章）。参见Python的手册或参考资料，以了解可用的命令行选项的具体细节。或者更好的做法是，问Python自己，即运行如下的命令行：

```
c:\Python30> python -?
```

以请求Python的帮助显示，它给出了可用的命令行选项。如果要处理复杂的命令行，应确保还查看标准库模块getopt和optparse，它们支持更加复杂的命令行处理。

寻求更多帮助

Python的标准手册集如今包含了针对各种平台上用法的有价值提示。在安装了Python之后，在Windows下通过“开始”按钮可以访问标准手册集，通过*http://www.python.org*也可以在线访问。找到手册集中标题为“Using Python”的顶级部分，以了解更加特定于平台的介绍和提示，以及最新的跨平台环境和命令行细节。

和往常一样，Web也是我们的朋友，尤其是在一个快速变化的领域，其变化速度比图书的更新快多了。由于Python广为采用，所以通过Web搜索可以找到关于Python使用问题的任何解答，这样的机会很大。

附录B

各部分练习题的解答

第一部分 使用入门

参考第3章“第一部分　练习题”中的习题。

1. *交互*。假设Python已正确配置，交互模式看起来应该就像这样（可以在IDLE或shell提示符下运行）。

```
% python
...copyright information lines...
>>> "Hello World!"
'Hello World!'
>>>                # Use Ctrl-D or Ctrl-Z to exit, or close window
```

2. *程序*。你的程序代码（即模块）文件*module1.py*和操作系统shell的交互看起来应该像这样：

```
print('Hello module world!')
```

```
% python module1.py
Hello module world!
```

同样，你也可以用其他方式运行：单击文件图标、使用IDLE的Run/Run Module菜单选项等。

3. *模块*。下面的交互说明了如何导入模块文件从而运行一个模块。

```
% python
>>> import module1
Hello module world!
>>>
```

要记住，不停止和重启解释器时，需要重载模块才能再次运行它。把文件移到不同目录并导入它，是很有技巧性的问题：如果Python在最初的目录中产生*module1.pyc*文件，即使源代码文件（*.py*）已被移到不在Python搜索路径中的目录，导入该模块时，Python依然会使用这个pyc文件。如果Python可读取源代码文件的目录，就会自动写*.pyc*文件，*.pyc*文件包含模块编译后的字节码的版本。参考第3章有关模块的内容。

4. *脚本*。假设你的平台支持#!技巧，你的解法看起来应该像这样（虽然你的#!行可能需要列出机器上的另一路径）：

```
#!/usr/local/bin/python          (or #!/usr/bin/env python)
print('Hello module world!')
% chmod +x module1.py

% module1.py
Hello module world!
```

5. *错误*。下面的交互模式（在Python 3.0下运行）示范了当你完成此练习题时会碰到的出错消息的种类。其实，你触发的是Python异常；默认异常处理行为会终止正在运行的Python程序，然后在屏幕上打印出错消息和堆栈的跟踪信息。堆栈的跟踪信息显示出当异常发生时，程序所处在的位置。在第七部分中，你会学到，可以使用`try`语句捕捉它，并进行任意的处理。你也会看到Python包含成熟的源代码调试器，从而可以满足特殊的错误检测的需求。就目前而言，程序错误发生时，Python会提供有意义的消息而不是默默地就崩溃了：

```
% python
>>> 2 ** 500
3273390607896141870013189696827599152216642046043064789483291368096133796404674554883270092325904157150886684127560071009217256545885393053328527589376
>>>
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
>>>
>>> spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

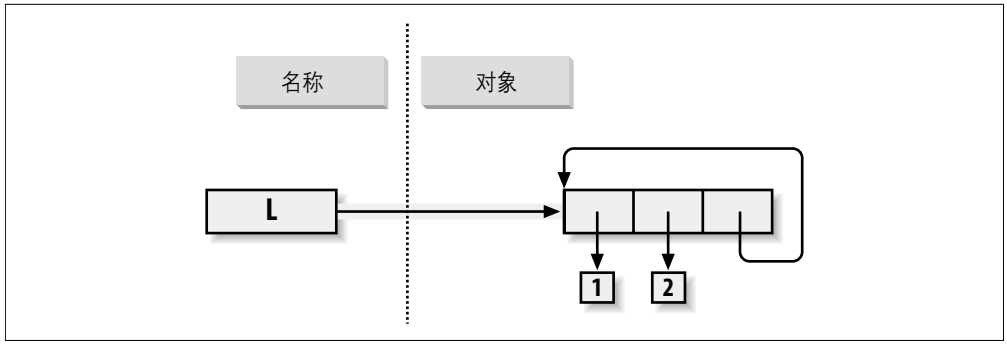
6. *中断和循环*。当你输入以下代码的时候：

```
L = [1, 2]
L.append(L)
```

会在Python中创建循环数据结构。在1.5.1版以前，Python打印不够智能，无法检测对象中的循环，而且会打印无止境的`[1, 2, [1, 2, [1, 2, [1, 2,`流，直到你按

下机器上的中断组合键（从技术上来讲，就是引发键盘中断异常，并打印默认消息）。从Python 1.5.1起，打印已经足够智能，可以检测循环，并打印[[...]]，而不是让你知道它已经在对象结构中检测到一个循环并避免永远打印。

循环的原因很微妙，而且需要第二部分的信息。但是，简而言之，Python中的赋值语句一定会产生对象的引用值，而不是它们的拷贝。你可以把对象看作是一块内存，把引用看作是隐式指向的指针。当你执行上面的第一个赋值语句时，名称L变成了对两个元素的列表对象的引用，也就是指向一段内存的指针。Python列表其实是对象引用值的数组，有一个append方法会通过在末尾添加另一个对象的引用，对数组进行原处修改。在这里，append调用会把L前面的引用加在L末尾，从而造成图B-1所示的循环：列表末尾的一个指针指回到列表的前面。



图B-1：循环对象，通过把列表附加在自身而生成。在默认情况下，Python是附加最初的列表的引用值，而不是列表的拷贝

除了特殊的打印，正如我们在第6章中学习的，循环对象还必须由Python的垃圾收集器特殊处理，否则，当它们不再使用的时候，其空间将保持未回收。尽管这种情况在实际中很少见，但在一些遍历任意对象或结构的程序中，你必须通过记录已经遍历到哪里，从而检测这样的循环，以避免陷入循环。不管你相信与否，循环数据结构偶尔也会很有用的（但不包括打印的时候）。

第二部分 类型和运算

参考第9章“第二部分　练习题”中的习题。

1. *基础*。以下是你应该得到的各种结果，还有其含义的注释。其中一些使用分号“;”把一个以上的语句挤在一行中（这里的“;”是语句分隔符），逗号构建了在圆括号中显示的元组。还要记住，靠近顶部的/除法结果在Python 2.6和Python 3.0中有所不同（参见第5章了解更多细节），并且，`list`包装字典方法调用以显示结果，这在Python 3.0中是必需的，但在Python 2.6中不是（参见第8章）。

```
# Numbers

>>> 2 ** 16                          # 2 raised to the power 16
65536
>>> 2 / 5, 2 / 5.0                   # Integer / truncates in 2.6, but not 3.0
(0.40000000000000002, 0.40000000000000002)

# Strings

>>> "spam" + "eggs"                  # Concatenation
'spameggs'
>>> S = "ham"
>>> "eggs " + S
'eggs ham'
>>> S * 5                            # Repetition
'hamhamhamhamham'
>>> S[:0]                            # An empty slice at the front -- [0:0]
''                                   # Empty of same type as object sliced

>>> "green %s and %s" % ("eggs", S)  # Formatting
'green eggs and ham'
>>> 'green {0} and {1}'.format('eggs', S)
'green eggs and ham'

# Tuples

>>> ('x',)[0]                        # Indexing a single-item tuple
'x'
>>> ('x', 'y')[1]                    # Indexing a 2-item tuple
'y'

# Lists

>>> L = [1,2,3] + [4,5,6]            # List operations
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> ([1,2,3]+[4,5,6])[2:4]
[3, 4]
>>> [L[2], L[3]]                     # Fetch from offsets; store in a list
[3, 4]
>>> L.reverse(); L                   # Method: reverse list in-place
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L                      # Method: sort list in-place
[1, 2, 3, 4, 5, 6]
>>> L.index(4)                       # Method: offset of first 4 (search)
3

# Dictionaries

>>> {'a':1, 'b':2}['b']              # Index a dictionary by key
2
>>> D = {'x':1, 'y':2, 'z':3}
>>> D['w'] = 0                       # Create a new entry
>>> D['x'] + D['w']
1
>>> D[(1,2,3)] = 4                   # A tuple used as a key (immutable)
```

```
>>> D
{'w': 0, 'z': 3, 'y': 2, (1, 2, 3): 4, 'x': 1}

>>> list(D.keys()), list(D.values()), (1,2,3) in D          # Methods, key test
(['w', 'z', 'y', (1, 2, 3), 'x'], [0, 3, 2, 4, 1], True)

# Empties

>>> [[]], ["",[],(),{},None]                  # Lots of nothings: empty objects
([[]], ['', [], (), {}, None])
```

2. *索引运算和分片运算*。超出边界的索引运算（例如，`L[4]`）会引发错误。Python一定会检查，以确保所有偏移值都在序列边界内。

   另外，分片运算超出边界（例如，`L[-1000:100]`）可工作，因为Python会缩放超出边界的分片以使其合用（必要时，限制值可设为零和序列长度）。

   以翻转的方式提取序列是行不通的（ 较低边界值比较高边界值更大， 例如，`L[3:1]`）。你会得到空分片（`[]`），因为Python会缩放分片限制值，以确定较低边界永远比较高边界小或相等（例如，`L[3:1]`会缩放成`L[3:3]`，空的插入点是在偏移值3处）。Python分片一定是从左至右抽取，即使你用负号索引值也是这样（会先加上序列长度转换成正值）。注意，Python 2.3的第三限制值分片会稍微修改此行为，例如`L[3:1:-1]`的确是从右至左抽取。

```
>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]
>>> L
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]
```

3. *索引运算、分片运算以及del*。你和解释器的交互看起来应该像下列程序代码。注意把空列表赋值给一个偏移值，会将空列表对象保存在这里，不过赋值空列表给一个分片，则会删除该分片。分片赋值运算期待得到的是另一个序列，否则你就会得到类型错误。这是把元素插入赋值之序列之内，而非序列本身：

```
>>> L = [1,2,3,4]
>>> L[2] = []
>>> L
[1, 2, [], 4]
>>> L[2:3] = []
>>> L
```

```
[1, 2, 4]
>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]
>>> L[1:2] = 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

4. *元组赋值运算*。交换X和Y的值。当元组出现在赋值符号（=）左右两边时，Python会根据左右两侧对象的位置，把右侧对象赋值给左边的目标。注意，左边的那些目标其实并非真正的元组（虽然看起来很像），可能最容易理解。那些只是一组独立的赋值目标。右侧的元素则是元组，也就是会在赋值运算进行时分解（元组提供所需要的临时赋值运算从而达到交换的效果）：

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
>>> X
'eggs'
>>> Y
'spam'
```

5. *字典键*。任何不可变对象都可作为字典的键，包括整数、元组和字符串等。这其实是字典，即使有些键看起来像整数偏移值。混合类型的键也能够正常工作：

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *字典索引运算*。对不存在的键进行索引运算（D['d']）会引发错误。对不存在的键做赋值运算（D['d']='spam'），则会创建新的字典元素。另一方面，列表超边界索引运算也会引发错误，超边界赋值运算也是。变量名称就像字典键那样，在引用时，必须已做了赋值。在首次赋值时，就会创建它。实际上，变量名能作为字典键来处理［在模块命名空间或堆栈框架字典（stack-frame dictionary）中都是可见的］：

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D['a']
1
>>> D['d']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
```

```
KeyError: d
>>> D['d'] = 4
>>> D
{'b': 2, 'd': 4, 'a': 1, 'c': 3}
>>>
>>> L = [0, 1]
>>> L[2]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[2] = 3
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

7. *通用运算*。问题解答：

   - +运算符无法用于不同/混合类型（例如，字符串+列表，列表+元组）。
   - +不适用于字典，因为那不是序列。
   - append方法只适用于列表，不适用于字符串，而键只适用于字典。append假设其目标是可变的，因为这是一个原地的扩展，字符串是不可变的。
   - 分片和合并运算一定会在对象处理后传回相同类型的新对象：

```
>>> "x" + 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
>>>
>>> {} + {}
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: bad operand type(s) for +
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: attribute-less object
>>>
>>> list({}.keys())                          # list needed in 3.0, not 2.6
[]
>>> [].keys()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: keys
>>>
>>> [][:]
[]
>>> ""[:]
''
```

8. *字符串索引运算*。因为字符串是单个字符的字符串的集合体，每次对字符串进行索引运算时，就会得到一个可再进行索引运算的字符串。S[0][0][0][0][0]就是一直对第一个字符做索引运算。这一般不适用于列表（列表可包含任意对象），除非列表包含了字符串：

```
>>> S = "spam"
>>> S[0][0][0][0][0]
's'
>>> L = ['s', 'p']
>>> L[0][0][0]
's'
```

9. *不可变类型*。下列任意解答都行。索引赋值运算不行，因为字符串是不可变的：

```
>>> S = "spam"
>>> S = S[0] + 'l' + S[2:]
>>> S
'slam'
>>> S = S[0] + 'l' + S[2] + S[3]
>>> S
'slam'
```

（参见第36章中关于Python 3.0的bytearray字符串类型的介绍——它是小整数的一个可变的序列，基本上可与字符串一样处理。）

10. *嵌套*。以下为例子：

```
>>> me = {'name':('John', 'Q', 'Doe'), 'age':'?', 'job':'engineer'}
>>> me['job']
'engineer'
>>> me['name'][2]
'Doe'
```

11. *文件*。下面是在Python中创建和读取文本文件的方法（ls是UNIX命令，在Windows中则使用dir）：

```
# File: maker.py
file = open('myfile.txt', 'w')
file.write('Hello file world!\n')          # Or: open().write()
file.close()                               # close not always needed

# File: reader.py
file = open('myfile.txt')                  # 'r' is default open mode
print(file.read())                         # Or print(open().read())

% python maker.py
% python reader.py
Hello file world!
% ls -l myfile.txt
-rwxrwxrwa  1 0       0 19 Apr 13 16:33 myfile.txt
```

第三部分 语句和语法

参考第15章“第三部分　练习题”中的习题。

1. *编写基本循环*。当你做这个练习题时，最后的代码会像这样：

```
>>> S = 'spam'
>>> for c in S:
...     print(ord(c))
...
115
112
97
109

>>> x = 0
>>> for c in S: x += ord(c)                # Or: x = x + ord(c)
...
>>> x
433

>>> x = []
>>> for c in S: x.append(ord(c))
...
>>> x
[115, 112, 97, 109]

>>> list(map(ord, S))                      # list() required in 3.0, not 2.6
[115, 112, 97, 109]
```

2. *反斜线字符*。这个例子会打印铃声字符（\a）50次。假设你的机器能处理，而且是在IDLE外运行，你就会听到一系列哔哔声（或者如果你的机器够快的话，就是一长声）。

3. *排序字典*。下面是做这个练习题的一种方式（如果看不懂的话，就参考第8章或第14章）。记住，你确实是应该把keys和sort调用像这样分开，因为sort会返回None。在Python 2.2和后续版本中，你可以直接迭代字典的键，而不需要调用keys（例如，for key in D:），但是，键列表无法像这段代码那样排序。在新近Python版本中，你也可以使用内置函数sorted来达到相同的效果：

```
>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = list(D.keys())                  # list() required in 3.0, not in 2.6
>>> keys.sort()
>>> for key in keys:
...     print(key, '=>', D[key])
...
a => 1
b => 2
```

```
c => 3
d => 4
e => 5
f => 6
g => 7

>>> for key in sorted(D):                    # Better, in more recent Pythons
...     print(key, '=>', D[key])
```

4. *程序逻辑替代方案*。这里是一些解答的样本代码。对于步骤e，把`2 ** X`的结果赋给步骤a和步骤b的循环外的一个变量，并且在循环内使用它。你的结果也许不同。这个练习题的设计目的，主要就是让你练习代码的替代方案，所以任何合理的结果都是满分：

```
# a

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

i = 0
while i < len(L):
    if 2 ** X == L[i]:
        print('at index', i)
        break
    i += 1
else:
    print(X, 'not found')

# b

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

for p in L:
    if (2 ** X) == p:
        print((2 ** X), 'was found at', L.index(p))
        break
else:
    print(X, 'not found')

# c

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# d

X = 5
L = []
for i in range(7): L.append(2 ** i)
```

```
print(L)

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')

# f

X = 5
L = list(map(lambda x: 2**x, range(7)))          # or [2**x for x in range(7)]
print(L)                                         # list() to print all in 3.0, not 2.6

if (2 ** X) in L:
    print((2 ** X), 'was found at', L.index(2 ** X))
else:
    print(X, 'not found')
```

第四部分 函数

参考第20章“第四部分　练习题”的习题。

1. *基础*。这题没什么，但是要注意，使用`print`（以及你的函数），从技术上来讲就是多态运算，也就是为每种类型的对象做正确的事：

```
% python
>>> def func(x): print(x)
...
>>> func("spam")
spam
>>> func(42)
42
>>> func([1, 2, 3])
[1, 2, 3]
>>> func({'food': 'spam'})
{'food': 'spam'}
```

2. *参数*。下面是示范的解答。记住，你得使用`print`才能查看测试调用的结果，因为文件和交互模式下输入的代码并不相同。一般而言，Python不会回显文件中表达式语句的结果：

```
def adder(x, y):
    return x + y

print(adder(2, 3))
print(adder('spam', 'eggs'))
print(adder(['a', 'b'], ['c', 'd']))

% python mod.py
5
spameggs
['a', 'b', 'c', 'd']
```

3. *可变参数*。在下面的*adders.py*文件中，有两个版本的adder函数。这里的难点在于，了解如何把累加器初始值设置为任何传入类型的空值。第一种解法是使用手动类型测试，从而找出整数，以及如果参数不是整数时，第一参数（假设为序列）的空分片。第二个解法是用第一个参数设定初始值，之后扫描第二元素和之后的元素，很像第18章中的各种min函数版本。

   第二个解法更好。这两种解法都假设所有参数为相同的类型，而且都无法用于字典（正如第二部分所看到的，+无法用在混合类型或字典上）。你也可以加上类型检测和特殊代码从而兼容字典，但那是额外的加分项了。

```
def adder1(*args):
    print('adder1', end=' ')
    if type(args[0]) == type(0):                  # Integer?
         sum = 0                                  # Init to zero
    else:                                         # else sequence:
         sum = args[0][:0]                        # Use empty slice of arg1
    for arg in args:
        sum = sum + arg
    return sum

def adder2(*args):
    print('adder2', end=' ')
    sum = args[0]                                 # Init to arg1
    for next in args[1:]:
        sum += next                               # Add items 2..N
    return sum

for func in (adder1, adder2):
    print(func(2, 3, 4))
    print(func('spam', 'eggs', 'toast'))
    print(func(['a', 'b'], ['c', 'd'], ['e', 'f']))
```

```
% python adders.py
adder1 9
adder1 spameggstoast
adder1 ['a', 'b', 'c', 'd', 'e', 'f']
adder2 9
adder2 spameggstoast
adder2 ['a', 'b', 'c', 'd', 'e', 'f']
```

4. *关键字参数*。下面是我对这个练习题第一部分的解答（文件*mod.py*）。要遍历关键词参数时，在函数开头列使用** args形式，并且使用循环［例如，for x in args.keys(): use args[x]］，或者使用args.values()，使其等同于计算*args位置参数的和：

```
def adder(good=1, bad=2, ugly=3):
    return good + bad + ugly

print(adder())
print(adder(5))
print(adder(5, 6))
```

```
print(adder(5, 6, 7))
print(adder(ugly=7, good=6, bad=5))

% python mod.py
6
10
14
18
18

# Second part solutions

def adder1(*args):                          # Sum any number of positional args
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder2(**args):                         # Sum any number of keyword args
    argskeys = list(args.keys())            # list needed in 3.0!
    tot = args[argskeys[0]]
    for key in argskeys[1:]:
        tot += args[key]
    return tot

def adder3(**args):                         # Same, but convert to list of values
    args = list(args.values())              # list needed to index in 3.0!
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder4(**args):                         # Same, but reuse positional version
    return adder1(*args.values())

print(adder1(1, 2, 3),       adder1('aa', 'bb', 'cc'))
print(adder2(a=1, b=2, c=3), adder2(a='aa', b='bb', c='cc'))
print(adder3(a=1, b=2, c=3), adder3(a='aa', b='bb', c='cc'))
print(adder4(a=1, b=2, c=3), adder4(a='aa', b='bb', c='cc'))
```

5. （和6.）下面是对练习题5和6的解答（文件*dicts.py*）。不过，这些只是编写代码的练习，因为Python 1.5新增了字典方法`D.copy()`和`D1.update(D2)`来处理字典的复制和更新（合并）等情况（参考Python的链接库手册或者O'Reilly的《Python Pocket Reference》以获得更多细节）。`X[:]`不适用于字典，因为字典不是序列（参考第8章的细节）。此外，记住，如果你是做赋值（`e = d`），而不是复制，将产生共享字典对象的引用值，修改`d`也会跟着修改`e`：

```
def copyDict(old):
    new = {}
    for key in old.keys():
        new[key] = old[key]
    return new

def addDict(d1, d2):
```

```
    new = {}
    for key in d1.keys():
        new[key] = d1[key]
    for key in d2.keys():
        new[key] = d2[key]
    return new
```

```
% python
>>> from dicts import *
>>> d = {1: 1, 2: 2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}

>>> x = {1: 1}
>>> y = {2: 2}
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}
```

6. 参见5。

7. *其他参数匹配的例子*。下面是你应该得到的交互模式下的结果，还有注释说明了其匹配情况：

```
def f1(a, b): print(a, b)                # Normal args

def f2(a, *b): print(a, b)               # Positional varargs

def f3(a, **b): print(a, b)              # Keyword varargs

def f4(a, *b, **c): print(a, b, c)       # Mixed modes

def f5(a, b=2, c=3): print(a, b, c)      # Defaults

def f6(a, b=2, *c): print(a, b, c)       # Defaults and positional varargs

% python
>>> f1(1, 2)                             # Matched by position (order matters)
1 2
>>> f1(b=2, a=1)                         # Matched by name (order doesn't matter)
1 2

>>> f2(1, 2, 3)                          # Extra positionals collected in a tuple
1 (2, 3)

>>> f3(1, x=2, y=3)                      # Extra keywords collected in a dictionary
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, x=2, y=3)                # Extra of both kinds
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                                # Both defaults kick in
1 2 3
```

```
>>> f5(1, 4)                    # Only one default used
1 4 3

>>> f6(1)                       # One argument: matches "a"
1 2 ()

>>> f6(1, 3, 4)                 # Extra positional collected
1 3 (4,)
```

8. *再谈质数*。下面是质数的实例，封装在函数和模块中（文件*primes.py*），可以多次运行。增加了一个`if`测试，从而考虑了负数、`0`以及`1`。把`/`改成`//`，从而使这个解答不会受到第5章提到的Python 3.0的`/`真除法改变的困扰，并且使其支持浮点数。（把`from`语句的注释去掉，把`//`改成`/`，看看在Python 2.6中的不同）：

```
#from __future__ import division

def prime(y):
    if y <= 1:                                  # For some y > 1
        print(y, 'not prime')
    else:
        x = y // 2                              # 3.0 / fails
        while x > 1:
            if y % x == 0:                      # No remainder?
                print(y, 'has factor', x)
                break                           # Skip else
            x -= 1
        else:
            print(y, 'is prime')

prime(13); prime(13.0)
prime(15); prime(15.0)
prime(3); prime(2)
prime(1); prime(-3)
```

下面是这个模块的运行。即使可能不该这样，但//运算符也适用于浮点数：

```
% python primes.py
13 is prime
13.0 is prime
15 has factor 5
15.0 has factor 5.0
3 is prime
2 is prime
1 not prime
-3 not prime
```

这个函数没有太好的可重用性，但可以改为返回值，而不是打印，不过作为实验已经足够。这也不是严格的数学质数（浮点数也行），而且依然没有效率。改进的事就留给数学考虑周密的读者作为练习。（提示：通过`for`循环来运行`range(y, 1,-1)`，可能会比`while`快一些，真正的瓶颈在于算法。）要测试替代方案的时间，

可以使用内置的time模块以及下面这个通用的函数调用timer中所用到的编写代码的模式（参考库手册以获得更多细节）：

```
def timer(reps, func, *args):
    import time
    start = time.clock()
    for i in range(reps):
        func(*args)
    return time.clock() - start
```

9. *列表解析*。下面是你应该写出来的代码的样子。其中有我自己的偏好，不要求都照着做：

```
>>> values = [2, 4, 9, 16, 25]
>>> import math

>>> res = []
>>> for x in values: res.append(math.sqrt(x))
...
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(map(math.sqrt, values))
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> [math.sqrt(x) for x in values]
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

10. *计时工具*。下面是我编写来对3个平方根选项计时的代码，带有在Python 2.6和Python 3.0中的结果。每个函数最后的结果打印出来，以验证所有3个方案都做同样的工作：

```
# File mytimer.py (2.6 and 3.0)

...same as listed in Chapter 20...

# File timesqrt.py

import sys, mytimer
reps = 10000
repslist = range(reps)                  # Pull out range list time for 2.6

from math import sqrt                   # Not math.sqrt: adds attr fetch time
def mathMod():
    for i in repslist:
        res = sqrt(i)
    return res

def powCall():
    for i in repslist:
        res = pow(i, .5)
    return res

def powExpr():
```

```
    for i in repslist:
        res = i ** .5
    return res

print(sys.version)
for tester in (mytimer.timer, mytimer.best):
    print('<%s>' % tester.__name__)
    for test in (mathMod, powCall, powExpr):
        elapsed, result = tester(test)
        print ('-'*35)
        print ('%s: %.5f => %s' %
               (test.__name__, elapsed, result))
```

如下是针对Python 3.0和Python 2.6的测试结果。对这两者而言，看上去math模块比**表达式更快，**表达式比pow调用更快。然而，应该在你自己的机器上以及Python版本中尝试一下。此外要注意，对于这一测试，Python 3.0几乎比Python 2.6慢两倍；Python 3.1或以后的版本可能表现更好些（将来进行测试自己看看结果）：

```
c:\misc> c:\python30\python timesqrt.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
<timer>
-----------------------------------
mathMod: 5.33906 => 99.994999875
-----------------------------------
powCall: 7.29689 => 99.994999875
-----------------------------------
powExpr: 5.95770 => 99.994999875
<best>
-----------------------------------
mathMod: 0.00497 => 99.994999875
-----------------------------------
powCall: 0.00671 => 99.994999875
-----------------------------------
powExpr: 0.00540 => 99.994999875


c:\misc> c:\python26\python timesqrt.py
2.6.1 (r261:67517, Dec 4 2008, 16:51:00) [MSC v.1500 32 bit (Intel)]
<timer>
-----------------------------------
mathMod: 2.61226 => 99.994999875
-----------------------------------
powCall: 4.33705 => 99.994999875
-----------------------------------
powExpr: 3.12502 => 99.994999875
<best>
-----------------------------------
mathMod: 0.00236 => 99.994999875
-----------------------------------
powCall: 0.00402 => 99.994999875
-----------------------------------
powExpr: 0.00287 => 99.994999875
```

要计时Python 3.0字典解析和对等的for循环交互的相对速度，应运行如下的一个会话。事实表明，这两者在Python 3.0下大致是相同的；然而，和列表解析不同，手动循环如今比字典解析略快（尽管差异并不大，当我们生成50个字典每个字典1 000 000项的时候，会节省半秒钟）。再次说明，你应该自己进一步调查，在自己的计算机和Python中测试，而不是把这些结果作为标准：

```
c:\misc> c:\python30\python
>>>
>>> def dictcomp(I):
...     return {i: i for i in range(I)}
...
>>> def dictloop(I):
...     new = {}
...     for i in range(I): new[i] = i
...     return new
...
>>> dictcomp(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>> dictloop(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>>
>>> from mytimer import best, timer
>>> best(dictcomp, 10000)[0]                # 10,000-item dict
0.0013519874732672577
>>> best(dictloop, 10000)[0]
0.001132965223233029
>>>
>>> best(dictcomp, 100000)[0]               # 100,000 items: 10 times slower
0.01816089754424155
>>> best(dictloop, 100000)[0]
0.01643484018219965
>>>
>>> best(dictcomp, 1000000)[0]              # 1,000,000 items: 10X time
0.18685105229855026
>>> best(dictloop, 1000000)[0]              # Time for making one dict
0.1769041177020938
>>>
>>> timer(dictcomp, 1000000, _reps=50)[0]       # 1,000,000-item dict
10.692516087938543
>>> timer(dictloop, 1000000, _reps=50)[0]       # Time for making 50
10.197276050447755
```

第五部分 模块

参考第24章“第五部分　练习题”的习题。

1. 导入*基础*。这一道题比你想象的更简单。做完后，文件（*mymod.py*）和交互的结果看起来如下所示。记住，Python可以把整个文件读成字符串列表，而len内置函数可返回字符串和列表的长度：

```
def countLines(name):
    file = open(name)
    return len(file.readlines())

def countChars(name):
    return len(open(name).read())

def test(name):                                     # Or pass file object
    return countLines(name), countChars(name)       # Or return a dictionary
```

```
% python
>>> import mymod
>>> mymod.test('mymod.py')
(10, 291)
```

这些函数一次把整个文件加载到了内存中，当文件过大以至于机器的内存无法容纳时，就不能用了。为了更健壮一些，你可以改用迭代器逐行读取，在此过程中进行计数：

```
def countLines(name):
    tot = 0
    for line in open(name): tot += 1
    return tot

def countChars(name):
    tot = 0
    for line in open(name): tot += len(line)
    return tot
```

在UNIX上，你可以使用`wc`命令确认输出。在Windows中，对文件单击鼠标右键，来查看其属性。但是请注意，你的脚本报告的字符数可能会比Windows的少：为了可移植，Python把Windows `\r\n`行尾标识符转换成了`\n`，每行会少一个字节（字符）。为了和Windows的字节计数相同，你得使用二进制模式打开文件（`'rb'`），或者根据行数，加上对应的字节数。

顺便提一下，要做这道练习题中的“志向远大”的部分（传入文件对象，只打开文件一次），你可能需要使用内置文件对象的`seek`方法。本书没有提到，但其工作起来就像C的`fseek`调用（也是调用`seek`）：`seek`会把文件当前位置重设为传入的偏移值。`seek`运行后，未来的输入/输出运算就是相对于新位置而开始的。想要回滚到文件开头位置而又不关闭文件并重新打开，就需要调用`file.seek(0)`。文件的`read`方法会从文件当前位置开始读起，你得回滚到开头重新读取文件。下面是调整后的程序：

```
def countLines(file):
    file.seek(0)                                    # Rewind to start of file
    return len(file.readlines())

def countChars(file):
    file.seek(0)                                    # Ditto (rewind if needed)
```

```
    return len(file.read())

def test(name):
    file = open(name)                               # Pass file object
    return countLines(file), countChars(file)       # Open file only once

>>> import mymod2
>>> mymod2.test("mymod2.py")
(11, 392)
```

2. from/from *。这里是from *部分；把*换成countChars就是其余的答案：

```
% python
>>> from mymod import *
>>> countChars("mymod.py")
291
```

3. __main__。如果你写得正确的话，哪种模式都能使用（运行程序或模块导入）：

```
def countLines(name):
    file = open(name)
    return len(file.readlines())

def countChars(name):
    return len(open(name).read())

def test(name):                                     # Or pass file object
    return countLines(name), countChars(name)       # Or return a dictionary

if __name__ == '__main__':
    print(test('mymod.py'))

% python mymod.py
(13, 346)
```

在这里可能应该开始考虑使用命令行参数或用户输入来提供要统计的文件名，而不是在脚本中硬编码它（参见第24章了解关于sys.argv的更多内容，参见第10章了解关于输入的更多内容）：

```
if __name__ == '__main__':
    print(test(input('Enter file name:')))

if __name__ == '__main__':
    import sys
    print(test(sys.argv[1]))
```

4. *嵌套导入*。下面是该题的解答（*myclient.py*文件）：

```
from mymod import countLines, countChars
print(countLines('mymod.py'), countChars('mymod.py'))

% python myclient.py
13 346
```

至于这个问题的其余部分，因为from只是在导入者中赋值变量名，所以mymod的

函数可以在myclient的顶层存取（可导入），就好像mymod的def是位于myclient中。例如，另一个文件可以写成：

```
import myclient
myclient.countLines(...)

from myclient import countChars
countChars(...)
```

如果myclient用的是import而不是from，就需要使用路径，通过myclient以获得mymod中的函数：

```
import myclient
myclient.mymod.countLines(...)

from myclient import mymod
mymod.countChars(...)
```

通常来说，你可以定义收集器模块，从其他模块导入所有的变量名，使得那些变量名能在单个方便的模块中使用。使用下面的代码，最后会有变量名somename的三个不同拷贝（mod1.somename、collector.somename以及__main__.somename）。这三个名称都共享相同的整数对象，而只有在交互模式提示符下存在着变量名somename：

```
# File mod1.py
somename = 42

# File collector.py
from mod1 import *                      # Collect lots of names here
from mod2 import *                      # from assigns to my names
from mod3 import *

>>> from collector import somename
```

5. 导入包。在这个练习题中，把练习题3的解答*mymod.py*文件放到一个目录包中。下面是我们所做的事：在Windows命令提示字符界面下，创建目录以及所需要的*__init__.py*文件。如果是其他的平台，你得进行修改（例如，使用mv和vi，而不是move和edit）。这些命令对于任意目录都适用（只是刚好在Python安装目录下运行命令），而你也可以从文件管理器GUI界面下完成其中的一些事。

   当这样做以后，就有个*mypkg*子目录含有文件*__init__.py*和*mymod.py*。*mypkg*目录内需要有*__init__.py*，但其上层目录则不需要。*mypkg*位于模块搜索路径的主目录上。目录的初始设置文件中所写的print语句只会在导入时执行一次，不会有第二次：

```
C:\python30> mkdir mypkg
C:\Python30> move mymod.py mypkg\mymod.py
C:\Python30> edit mypkg\__init__.py
```

```
...coded a print statement...
C:\Python30> python
>>> import mypkg.mymod
initializing mypkg
>>> mypkg.mymod.countLines('mypkg\mymod.py')
13
>>> from mypkg.mymod import countChars
>>> countChars('mypkg\mymod.py')
346
```

6. *重载*。这道题只是要你实验一下修改本书的*changer.py*这个例子，所以这里没什么好写的。

7. *循环*导入。简单来说，结果就是先导入recur2，因为递归导入是发生在recur1中的import，而不是recur2的from。

   详细来讲是这样的：先导入recur2，这是因为从recur1到recur2的递归导入是整个取出recur2，而不是获取特定的变量名。从recur1导入时，recur2还不完整，因为其使用import而不是from，所以安全。Python会寻找并返回已创建的recur2模块对象，然后继续运行recur1剩余的部分，从而没有问题。当recur2的导入继续下去时，第二个from发现recur1（已完全执行）内的变量名Y，所以不会报告错误。把文件当成脚本执行与将其当成模块导入并不相同。这些情况与通过交互模式先运行脚本中的第一个import或from相同。例如，将recur1作为脚本执行，与通过交互模式导入recur2一样，因为recur2是recur1中导入的第一个模块。

第六部分 类和OOP

参考第31章“第六部分　练习题”的习题。

1. *继承*。下面是这个练习题的解答（*adder.py*文件），以及一些交互模式下的测试。__add__重载方法只出现一次，就是在超类中，因为它调用了子类中类型特定的add方法：

```
class Adder:
    def add(self, x, y):
        print('not implemented!')
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):                    # Or in subclasses?
        return self.add(self.data, other)        # Or return type?

class ListAdder(Adder):
    def add(self, x, y):
        return x + y

class DictAdder(Adder):
    def add(self, x, y):
```

```
        new = {}
        for k in x.keys(): new[k] = x[k]
        for k in y.keys(): new[k] = y[k]
        return new
```

```
% python
>>> from adder import *
>>> x = Adder()
>>> x.add(1, 2)
not implemented!
>>> x = ListAdder()
>>> x.add([1], [2])
[1, 2]
>>> x = DictAdder()
>>> x.add({1:1}, {2:2})
{1: 1, 2: 2}

>>> x = Adder([1])
>>> x + [2]
not implemented!
>>>
>>> x = ListAdder([1])
>>> x + [2]
[1, 2]
>>> [2] + x
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: __add__ nor __radd__ defined for these operands
```

在最后的测试中，得到表达式错误，因为+的右边出现类实例。如果你想修复它，可使用__radd__方法，就像第29章所描述的那样。

如果你在实例中保存一个值，可能也想重写add方法使其只带一个自变量（按照第六部分中其他例子的精神）：

```
class Adder:
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):            # Pass a single argument
        return self.add(other)           # The left side is in self
    def add(self, y):
        print('not implemented!')

class ListAdder(Adder):
    def add(self, y):
        return self.data + y

class DictAdder(Adder):
    def add(self, y):
        pass                             # Change to use self.data instead of x

x = ListAdder([1, 2, 3])
y = x + [4, 5, 6]
print(y)                                 # Prints [1, 2, 3, 4, 5, 6]
```

因为值是附加在对象上而不是到处传递，这个版本更加地面向对象。一旦你了解了，可能会发现，可以舍弃add，而在两个子类中定义类型特定的__add__方法。

2. *运算符重载*。答案中（文件*mylist.py*）使用的一些运算符重载方法，书中没有多谈，但它们应该是很容易理解的。复制构造函数中的初始值很重要，因为它是可变的。你不会想修改或者拥有可能被类外其他地方共享的对象参照值。__getattr__方法把调用转给包装列表。有关以Python 2.2以及后续版本编写这个代码的更为容易方式的提示，可以参考第31章：

```
class MyList:
    def __init__(self, start):
        #self.wrapped = start[:]                    # Copy start: no side effects
        self.wrapped = []                           # Make sure it's a list here
        for x in start: self.wrapped.append(x)
    def __add__(self, other):
        return MyList(self.wrapped + other)
    def __mul__(self, time):
        return MyList(self.wrapped * time)
    def __getitem__(self, offset):
        return self.wrapped[offset]
    def __len__(self):
        return len(self.wrapped)
    def __getslice__(self, low, high):
        return MyList(self.wrapped[low:high])
    def append(self, node):
        self.wrapped.append(node)
    def __getattr__(self, name):                    # Other methods: sort/reverse/etc
        return getattr(self.wrapped, name)
    def __repr__(self):
        return repr(self.wrapped)

if __name__ == '__main__':
    x = MyList('spam')
    print(x)
    print(x[2])
    print(x[1:])
    print(x + ['eggs'])
    print(x * 3)
    x.append('a')
    x.sort()
    for c in x: print(c, end=' ')
```

```
% python mylist.py
['s', 'p', 'a', 'm']
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 's', 'p', 'a', 'm', 's', 'p', 'a', 'm']
a a m p s
```

要注意，通过附加而不是分片复制初值是很重要的，否则结果就不是真正的列表，也就不会响应预期的列表方法，例如，append（例如，对字符串进行分片运算会传

回另一字符串，而不是列表）。你可以通过分片运算复制MyList的初值，因为其类重载了分片运算，而且提供预期的列表接口。然而，你需要避免对对象（例如字符串）做分片式的复制。此外，集合已经是Python的内置类型，这大体上只是编写代码的练习而已（参考第5章有关集合的细节）。

3. 子类。解答如下所示（*mysub.py*）。你的答案应该也类似：

```
from mylist import MyList

class MyListSub(MyList):
    calls = 0                                   # Shared by instances

    def __init__(self, start):
        self.adds = 0                           # Varies in each instance
        MyList.__init__(self, start)

    def __add__(self, other):
        MyListSub.calls += 1                    # Class-wide counter
        self.adds += 1                          # Per-instance counts
        return MyList.__add__(self, other)

    def stats(self):
        return self.calls, self.adds            # All adds, my adds

if __name__ == '__main__':
    x = MyListSub('spam')
    y = MyListSub('foo')
    print(x[2])
    print(x[1:])
    print(x + ['eggs'])
    print(x + ['toast'])
    print(y + ['bar'])
    print(x.stats())
```

```
% python mysub.py
a
['p', 'a', 'm']
['s', 'p', 'a', 'm', 'eggs']
['s', 'p', 'a', 'm', 'toast']
['f', 'o', 'o', 'bar']
(3, 2)
```

4. *元类方法*。注意，在Python 2.6中，运算符尝试通过__getattr__取得属性。你需要返回一个值使其能够工作。警告：正如第30章所提到的，__getattr__不会在Python 3.0中针对内置操作而调用，因此，如下的表达式不会像介绍的那样工作；在Python 3.0中，像这样的一个类必须显式地重新定义__X__运算符重载方法。关于这一点的更多介绍，参见第30章、第37章和第38章：

```
>>> class Meta:
...     def __getattr__(self, name):
...         print('get', name)
...     def __setattr__(self, name, value):
```

```
...          print('set', name, value)
...
>>> x = Meta()
>>> x.append
get append
>>> x.spam = "pork"
set spam pork
>>>
>>> x + 2
get __coerce__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
>>>
>>> x[1]
get __getitem__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function

>>> x[1:5]
get __len__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: call of non-function
```

5. *集合对象*。下面是应得到的交互模式下的结果。注释说明了调用的是哪个方法：

```
% python
>>> from setwrapper import Set
>>> x = Set([1, 2, 3, 4])                # Runs __init__
>>> y = Set([3, 4, 5])

>>> x & y                                # __and__, intersect, then __repr__
Set:[3, 4]
>>> x | y                                # __or__, union, then __repr__
Set:[1, 2, 3, 4, 5]

>>> z = Set("hello")                     # __init__ removes duplicates
>>> z[0], z[-1]                          # __getitem__
('h', 'o')

>>> for c in z: print(c, end=' ')        # __getitem__
...
h e l o
>>> len(z), z                            # __len__, __repr__
(4, Set:['h', 'e', 'l', 'o'])

>>> z & "mello", z | "mello"
(Set:['e', 'l', 'o'], Set:['h', 'e', 'l', 'o', 'm'])
```

对多个操作对象扩展的子类的解答，就像下面的类（*multiset.py*文件）一样。只需要取代最初集合中的两个方法。类的文档字符串说明了其工作原理：

```
from setwrapper import Set
```

```
class MultiSet(Set):
    """
    Inherits all Set names, but extends intersect
    and union to support multiple operands; note
    that "self" is still the first argument (stored
    in the *args argument now); also note that the
    inherited & and | operators call the new methods
    here with 2 arguments, but processing more than
    2 requires a method call, not an expression:
    """

    def intersect(self, *others):
        res = []
        for x in self:                          # Scan first sequence
            for other in others:                # For all other args
                if x not in other: break        # Item in each one?
            else:                               # No: break out of loop
                res.append(x)                   # Yes: add item to end
        return Set(res)

    def union(*args):                           # self is args[0]
        res = []
        for seq in args:                        # For all args
            for x in seq:                       # For all nodes
                if not x in res:
                    res.append(x)               # Add new items to result
        return Set(res)
```

与扩展的交互应该像下面所演示的。你可以使用&或调用`intersect`来做交集，但是对三个或以上的操作数，则必须调用`intersect`。&是二元（两边）运算符。此外，如果我们使用`setwrapper.Set`来引用`multiset`中的最初的类，那么也可以把`MultiSet`称为`Set`，让这样的改变变得更加透明。

```
>>> from multiset import *
>>> x = MultiSet([1,2,3,4])
>>> y = MultiSet([3,4,5])
>>> z = MultiSet([0,1,2])

>>> x & y, x | y                                # Two operands
(Set:[3, 4], Set:[1, 2, 3, 4, 5])

>>> x.intersect(y, z)                           # Three operands
Set:[]
>>> x.union(y, z)
Set:[1, 2, 3, 4, 5, 0]
>>> x.intersect([1,2,3], [2,3,4], [1,2,3])      # Four operands
Set:[2, 3]
>>> x.union(range(10))                          # Non-MultiSets work, too
Set:[1, 2, 3, 4, 0, 5, 6, 7, 8, 9]
```

6. *类树链接*。下面是修改`Lister`类的方法，并重新运行测试来显示其格式。对于基于`dir`的版本做同样的事情，并且当在树爬升变体中格式化类对象的时候也这么做：

```
class ListInstance:
    def __str__(self):
        return '<Instance of %s(%s), address %s:\n%s>' % (
                            self.__class__.__name__,        # My class's name
                            self.__supers(),                # My class's own supers
                            id(self),                       # My address
                            self.__attrnames()) )           # name=value list
    def __attrnames(self):
        ...unchanged...
    def __supers(self):
        names = []
        for super in self.__class__.__bases__:              # One level up from class
            names.append(super.__name__)                    # name, not str(super)
        return ', '.join(names)

C:\misc> python testmixin.py
<Instance of Sub(Super, ListInstance), address 7841200:
        name data1=spam
        name data2=eggs
        name data3=42
>
```

7. 组合。解答如下（*lunch.py*文件），注释混在代码中。这可能是用Python描述问题比英文更简单的情况之一：

```
class Lunch:
    def __init__(self):                             # Make/embed Customer, Employee
        self.cust = Customer()
        self.empl = Employee()
    def order(self, foodName):                      # Start Customer order simulation
        self.cust.placeOrder(foodName, self.empl)
    def result(self):                               # Ask the Customer about its Food
        self.cust.printFood()
class Customer:
    def __init__(self):                             # Initialize my food to None
        self.food = None
    def placeOrder(self, foodName, employee):       # Place order with Employee
        self.food = employee.takeOrder(foodName)
    def printFood(self):                            # Print the name of my food
        print(self.food.name)

class Employee:
    def takeOrder(self, foodName):                  # Return Food, with desired name
        return Food(foodName)

class Food:
    def __init__(self, name):                       # Store food name
        self.name = name

if __name__ == '__main__':
    x = Lunch()                                     # Self-test code
    x.order('burritos')                             # If run, not imported
    x.result()
    x.order('pizza')
    x.result()
```

```
% python lunch.py
burritos
pizza
```

8. *动物园动物继承层次*。下面是用Python编写的动物分类（*zoo.py*文件）。这是人工分法，这种通用化的编写代码模式适用于许多真实的结构，可以从GUI到员工数据库。Animal中引用的self.speak会触发独立的继承搜索，找到子类内的speak。通过交互模式测试这个练习题。试着用新类扩展这个层次，并在树中创建各种类的实例：

```
class Animal:
    def reply(self):    self.speak()              # Back to subclass
    def speak(self):    print('spam')             # Custom message

class Mammal(Animal):
    def speak(self):    print('huh?')

class Cat(Mammal):
    def speak(self):    print('meow')

class Dog(Mammal):
    def speak(self):    print('bark')

class Primate(Mammal):
    def speak(self):    print('Hello world!')

class Hacker(Primate):  pass                      # Inherit from Primate
```

9. *描绘死鹦鹉*。下面程序是我实现这道题的方法（*parrot.py*文件）。Actor超类的line方法的运作方式：读取self属性两次，让Python传回该实例两次。因此，会启动两次继承搜索（self.name和self.says()会在特定的子类内找到信息）：

```
class Actor:
    def line(self): print(self.name + ':', repr(self.says()))

class Customer(Actor):
    name = 'customer'
    def says(self): return "that's one ex-bird!"

class Clerk(Actor):
    name = 'clerk'
    def says(self): return "no it isn't..."

class Parrot(Actor):
    name = 'parrot'
    def says(self): return None

class Scene:
    def __init__(self):
        self.clerk = Clerk()                      # Embed some instances
        self.customer = Customer()                # Scene is a composite
        self.subject  = Parrot()
```

```
def action(self):
    self.customer.line()                    # Delegate to embedded
    self.clerk.line()
    self.subject.line()
```

第七部分 异常和工具

参考第35章“第七部分　练习题”的习题。

1. try/except。本书的oops函数如下所示（*oops.py*文件）。对于不是编程的问题，修改oops来引发KeyError而不是IndexError，意味着try处理器不会捕捉这个异常（而是“传播”到顶层，并触发Python的默认出错消息）。变量名KeyError和IndexError来自于最外层内置作用域。导入builtins（在Python 2.6中是__builtin__），将其作为一个参数传给dir函数，亲自看看结果：

```
def oops():
    raise IndexError()

def doomed():
    try:
        oops()
    except IndexError:
        print('caught an index error!')
    else:
        print('no error caught...')

if __name__ == '__main__': doomed()
```

```
% python oops.py
caught an index error!
```

2. *异常对象和列表*。下面是扩展这个模块来增加自己的异常（一开始，这里用字符串）：

```
class MyError(Exception): pass

def oops():
    raise MyError('Spam!')

def doomed():
    try:
        oops()
    except IndexError:
        print('caught an index error!')
    except MyError as data:
        print('caught error:', MyError, data)
    else:
        print('no error caught...')

if __name__ == '__main__':
    doomed()
```

```
% python oops.py
caught error: <class '__main__.MyError'> Spam!
```

就像所有类异常一样，实例变成了额外的数据。现在，出错信息会显示类(`<...>`)及其实例（`Spam!`）。该实例必须从Python的`Exception`类继承一个`__init__`和一个`__repr__`或`__str__`；否则，它将像类一样打印。参阅第34章，详细了解这在内置异常类中如何工作。

3. *错误处理*。下面是解这个练习题的方法（*safe2.py*文件）。在文件中做测试，而不是在交互模式下进行，结果差不多相同：

```
import sys, traceback

def safe(entry, *args):
    try:
        entry(*args)                        # Catch everything else
    except:
        traceback.print_exc()
        print('Got', sys.exc_info()[0], sys.exc_info()[1])

import oops
safe(oops.oops)
```

```
% python safe2.py
Traceback (innermost last):
  File "safe2.py", line 5, in safe
    entry(*args)                            # Catch everything else
  File "oops.py", line 4, in oops
    raise MyError, 'world'
hello: world
Got hello world
```

4. 这里是一些供你研究的例子。要找更多例子的话，可以参考后续的书籍和网络：

```
# Find the largest Python source file in a single directory

import os, glob
dirname = r'C:\Python30\Lib'

allsizes = []
allpy = glob.glob(dirname + os.sep + '*.py')
for filename in allpy:
    filesize = os.path.getsize(filename)
    allsizes.append((filesize, filename))

allsizes.sort()
print(allsizes[:2])
print(allsizes[-2:])

# Find the largest Python source file in an entire directory tree

import sys, os, pprint
if sys.platform[:3] == 'win':
    dirname = r'C:\Python30\Lib'
```

```
else:
    dirname = '/usr/lib/python'

allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    for filename in filesHere:
        if filename.endswith('.py'):
            fullname = os.path.join(thisDir, filename)
            fullsize = os.path.getsize(fullname)
            allsizes.append((fullsize, fullname))

allsizes.sort()
pprint.pprint(allsizes[:2])
pprint.pprint(allsizes[-2:])
```

 # Find the largest Python source file on the module import search path

```
import sys, os, pprint
visited = {}
allsizes = []
for srcdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(srcdir):
        thisDir = os.path.normpath(thisDir)
        if thisDir.upper() in visited:
            continue
        else:
            visited[thisDir.upper()] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                pypath = os.path.join(thisDir, filename)
                try:
                    pysize = os.path.getsize(pypath)
                except:
                    print('skipping', pypath)
                allsizes.append((pysize, pypath))

allsizes.sort()
pprint.pprint(allsizes[:3])
pprint.pprint(allsizes[-3:])
```

 # Sum columns in a text file separated by commas

```
filename = 'data.txt'
sums = {}

for line in open(filename):
    cols = line.split(',')
    nums = [int(col) for col in cols]
    for (ix, num) in enumerate(nums):
        sums[ix] = sums.get(ix, 0) + num

for key in sorted(sums):
    print(key, '=', sums[key])
```

 # Similar to prior, but using lists instead of dictionaries for sums

```
import sys
```

```
filename = sys.argv[1]
numcols = int(sys.argv[2])
totals = [0] * numcols

for line in open(filename):
    cols = line.split(',')
    nums = [int(x) for x in cols]
    totals = [(x + y) for (x, y) in zip(totals, nums)]

print(totals)
```

 # Test for regressions in the output of a set of scripts

```
import os
testscripts = [dict(script='test1.py', args=''),           # Or glob script/args dir
               dict(script='test2.py', args='spam')]

for testcase in testscripts:
    commandline = '%(script)s %(args)s' % testcase
    output = os.popen(commandline).read()
    result = testcase['script'] + '.result'
    if not os.path.exists(result):
        open(result, 'w').write(output)
        print('Created:', result)
    else:
        priorresult = open(result).read()
        if output != priorresult:
            print('FAILED:', testcase['script'])
            print(output)
        else:
            print('Passed:', testcase['script'])
```

 # Build GUI with tkinter (Tkinter in 2.6) with buttons that change color and grow

```
from tkinter import *                       # Use Tkinter in 2.6
import random
fontsize = 25
colors = ['red', 'green', 'blue', 'yellow', 'orange', 'white', 'cyan', 'purple']

def reply(text):
    print(text)
    popup = Toplevel()
    color = random.choice(colors)
    Label(popup, text='Popup', bg='black', fg=color).pack()
    L.config(fg=color)

def timer():
    L.config(fg=random.choice(colors))
    win.after(250, timer)

def grow():
    global fontsize
    fontsize += 5
    L.config(font=('arial', fontsize, 'italic'))
    win.after(100, grow)

win = Tk()
```

```
L = Label(win, text='Spam',
          font=('arial', fontsize, 'italic'), fg='yellow', bg='navy',
          relief=RAISED)
L.pack(side=TOP, expand=YES, fill=BOTH)
Button(win, text='press', command=(lambda: reply('red'))).pack(side=BOTTOM,fill=X)
Button(win, text='timer', command=timer).pack(side=BOTTOM, fill=X)
Button(win, text='grow', command=grow).pack(side=BOTTOM, fill=X)
win.mainloop()
```

 # Similar to prior, but use classes so each window has own state information

```
from tkinter import *
import random
class MyGui:
    """
    A GUI with buttons that change color and make the label grow
    """
    colors = ['blue', 'green', 'orange', 'red', 'brown', 'yellow']

    def __init__(self, parent, title='popup'):
        parent.title(title)
        self.growing = False
        self.fontsize = 10
        self.lab = Label(parent, text='Gui1', fg='white', bg='navy')
        self.lab.pack(expand=YES, fill=BOTH)
        Button(parent, text='Spam', command=self.reply).pack(side=LEFT)
        Button(parent, text='Grow', command=self.grow).pack(side=LEFT)
        Button(parent, text='Stop', command=self.stop).pack(side=LEFT)

    def reply(self):
        "change the button's color at random on Spam presses"
        self.fontsize += 5
        color = random.choice(self.colors)
        self.lab.config(bg=color,
                font=('courier', self.fontsize, 'bold italic'))

    def grow(self):
        "start making the label grow on Grow presses"
        self.growing = True
        self.grower()

    def grower(self):
        if self.growing:
            self.fontsize += 5
            self.lab.config(font=('courier', self.fontsize, 'bold'))
            self.lab.after(500, self.grower)

    def stop(self):
        "stop the button growing on Stop presses"
        self.growing = False

class MySubGui(MyGui):
    colors = ['black', 'purple']          # Customize to change color choices

MyGui(Tk(), 'main')
MyGui(Toplevel())
MySubGui(Toplevel())
```

```
mainloop()
```

 # Email inbox scanning and maintenance utility

```
"""
scan pop email box, fetching just headers, allowing
deletions without downloading the complete message
"""

import poplib, getpass, sys
mailserver = 'your pop email server name here'              # pop.rmi.net
mailuser = 'your pop email user name here'                  # brian
mailpasswd = getpass.getpass('Password for %s?' % mailserver)

print('Connecting...')
server = poplib.POP3(mailserver)
server.user(mailuser)
server.pass_(mailpasswd)

try:
    print(server.getwelcome())
    msgCount, mboxSize = server.stat()
    print('There are', msgCount, 'mail messages, size ', mboxSize)
    msginfo = server.list()
    print(msginfo)
   for i in range(msgCount):
       msgnum = i+1
       msgsize = msginfo[1][i].split()[1]
       resp, hdrlines, octets = server.top(msgnum, 0)       # Get hdrs only
       print('-'*80)
       print('[%d: octets=%d, size=%s]' % (msgnum, octets, msgsize))
       for line in hdrlines: print(line)

       if input('Print?') in ['y', 'Y']:
           for line in server.retr(msgnum)[1]: print(line) # Get whole msg
       if input('Delete?') in ['y', 'Y']:
           print('deleting')
           server.dele(msgnum)                              # Delete on srvr
       else:
           print('skipping')
finally:
   server.quit()                                    # Make sure we unlock mbox
input('Bye.')                                       # Keep window up on Windows
```

 # CGI server-side script to interact with a web browser

```
#!/usr/bin/python
import cgi
form = cgi.FieldStorage()                           # Parse form data
print("Content-type: text/html\n")                  # hdr plus blank line
print("<HTML>")
print("<title>Reply Page</title>")                  # HTML reply page
print("<BODY>")
if not 'user' in form:
    print("<h1>Who are you?</h1>")
else:
    print("<h1>Hello <i>%s</i>!</h1>" % cgi.escape(form['user'].value))
```

```
print("</BODY></HTML>")
```

 # Database script to populate and query a MySql database

```
from MySQLdb import Connect
conn = Connect(host='localhost', user='root', passwd='darling')
curs = conn.cursor()
try:
    curs.execute('drop database testpeopledb')
except:
    pass                                            # Did not exist

curs.execute('create database testpeopledb')
curs.execute('use testpeopledb')
curs.execute('create table people (name char(30), job char(10), pay int(4))')

curs.execute('insert people values (%s, %s, %s)', ('Bob', 'dev', 50000))
curs.execute('insert people values (%s, %s, %s)', ('Sue', 'dev', 60000))
curs.execute('insert people values (%s, %s, %s)', ('Ann', 'mgr', 40000))

curs.execute('select * from people')
for row in curs.fetchall():
    print(row)

curs.execute('select * from people where name = %s', ('Bob',))
print(curs.description)
colnames = [desc[0] for desc in curs.description]
while True:
    print('-' * 30)
    row = curs.fetchone()
    if not row: break
    for (name, value) in zip(colnames, row):
        print('%s => %s' % (name, value))

conn.commit()                                       # Save inserted records
```

 # Database script to populate a shelve with Python objects

```
# see also Chapter 27 shelve and Chapter 30 pickle examples

rec1 = {'name': {'first': 'Bob', 'last': 'Smith'},
        'job': ['dev', 'mgr'],
        'age': 40.5}

rec2 = {'name': {'first': 'Sue', 'last': 'Jones'},
        'job': ['mgr'],
        'age': 35.0}

import shelve
db = shelve.open('dbfile')
db['bob'] = rec1
db['sue'] = rec2
db.close()
```

 # Database script to print and update shelve created in prior script

```
import shelve
db = shelve.open('dbfile')
for key in db:
    print(key, '=>', db[key])

bob = db['bob']
bob['age'] += 1
db['bob'] = bob
db.close()
```