第八部分

高级话题

第36章

Unicode和字节字符串

在本书的核心类型部分关于**字符串**的一章中（第7章），我有意地限制了大多数Python程序员需要了解的字符串话题的子集的范围。因为大多数程序员只是处理像ASCII这样的文本的简单形式，他们快乐地使用着Python的基本的`str`字符串类型及其相关的操作，并且不需要掌握更加高级的字符串概念。实际上，这样的程序员很大程度上可以忽略Python 3.0中的字符串的变化，并且继续使用他们过去所使用的字符串。

另一方面，一些程序员处理更加专业的数据类型：非ASCII的字符串集、图像文件内容，等等。对于这些程序员（以及其他可能某一天加入这一队伍的程序员），在本章中，我们将介绍Python字符串的其他内容，并且探讨Python字符串模型中一些较为高级的话题。

特别是，我们将介绍Python支持的Unicode**文本**的基础知识——在国际化应用程序中使用的宽字符字符串，以及**二进制数据**——表示绝对的字节值的字符串。我们将看到，高级的字符串表示法在Python当前版本中已经产生了分歧：

- *Python 3.0*为二进制数据提供了一种替代字符串类型，并且在其常规的字符串类型中支持Unicode文本（ASCII看作Unicode的一种简单类型）。
- *Python 2.6*为非ASCII Unicode文本提供了一种替代字符串类型，并且在其常规的字符串类型中支持简单文本和二进制数据。

此外，由于Python的字符串模式对于如何处理非ASCII**文件**有着直接的影响，我们还将在这里介绍相关话题的基础知识。最后，我们还将简单地看看一些高级字符串和二进制**工具**，例如模式匹配、对象pickle化、二进制数据包装和XML解析，以及Python 3.0的字符串变化对它们产生影响的方式。

正式来说，本章是关于高级话题的一章，因为并不是所有的程序员都需要深入Unicode编码或二进制数据的世界。如果你需要关注处理这两种形式，那么，你将会发现Python的字符串模式提供了所需的支持。

Python 3.0中的字符串修改

Python 3.0中最引人注目的修改之一，就是字符串对象类型的变化。简而言之，Python 2.X的`str`和`unicode`类型已经融入了Python 3.0的`str`和`bytes`类型，并且增加了一种新的可变的类型`bytearray`。`bytearray`类型在Python 2.6中也可以使用（但在更早的版本中不能用），但是，它在Python 3.0中得到完全支持，并且不像是Python 2.6中那样清楚地区分文本和二进制内容。

特别是，如果我们处理本质上是Unicode或二进制的数据，这些修改对于代码可能会有切实的影响。实际上，作为首要的一般性规则，我们需要如何关注这一话题，很大程度上取决于遇到如下的哪种情况：

- 如果处理非ASCII Unicode**文本**，例如，在国际化应用程序或某些XML解析器的结果这样的环境中，你将会发现Python 3.0中对文本编码的支持是不同的，但是可能比Python 2.6中的支持更加直接、易用和无缝。
- 如果处理**二进制数据**，例如，使用struct模块处理的图形或音频文件的形式或打包的数据，我们需要理解Python 3.0中新的bytes对象，以及Python 3.0对文本和二进制数据和文件的不同和严格区分。
- 如果不属于前面两种情况的任何一种，在Python 3.0中，通常可以像是在Python 2.6中一样使用字符串：使用通用的str字符串类型、文本文件，以及我们前面所介绍的所有熟悉的字符串操作。字符串将使用平台默认的编码来进行编码和解码（例如，美国的Windows上的ASCII或UTF-8，如果我们仔细检查的话，`sys.getdefaultencoding()`给出默认的编码方式），但是，你可能不会注意。

换句话说，如果你的文本总是ASCII，可以使用常规的字符串对象和文本文件，并且避免下面介绍的大多数情况。正如稍后我们将见到的，ASCII是一种简单的Unicode，并且是其他编码的一个子集，因此，如果你的程序处理ASCII文本，字符串操作和文件“刚好够用”。

即便你遇到了刚刚提及的3种情况的最后一种，然而对Python 3.0字符串模式的基本理解，既可以帮助你理解一些底层的行为，也可以帮助你更容易地掌握Unicode或二进制数据问题，以免它们将来会影响到你。

Python 3.0对Unicode和二进制数据的支持在Python 2.6中也可以使用，虽然形式不同。尽管本章中主要关注的是Python 3.0中的字符串类型，在此过程中，我们还将讨论一些Python 2.6中的不同之处。不管你使用的是哪个版本，我们在这里介绍的工具在很多类型程序中将变得重要起来。

字符串基础知识

在查看任何代码之前，让我们先开始概览一下Python的字符串模型。要理解为什么Python 3.0改变了字符串的工作方式，我们必须先简短地看看字符实际是如何在计算机中表示的。

字符编码方法

大多数程序员把字符串看作是用来表示文本数据的一系列字符。但是，根据必须记录何种字符集，计算机内存中存储字符的方式有所不同。

ASCII标准在美国创建，并且定义了大多数美国程序员使用的文本字符串表示法。ASCII定义了从0到127的字符代码，并且允许每个字符存储在一个8位的字节中（实际上，只有其中的7位真正用到）。例如，ASCII标准把字符`'a'`映射为整数值`97`（十六进制中的0x61），它存储在内存和文件的一个单个字节中。如果想要看到这是如何工作的，Python的内置函数`ord`给出了一个字符的二进制值，并且`chr`针对一个给定的整数代码值返回其字符：

```
>>> ord('a')                    # 'a' is a byte with binary value 97 in ASCII
97
>>> hex(97)
'0x61'
>>> chr(97)                     # Binary value 97 stands for character 'a'
'a'
```

然而，有时候每个字符一个字节并不够。例如，各种符号和重音字符并不在ASCII所定义的可能字符的范围中。为了容纳特殊字符，一些标准允许一个8位字节中的所有可能的值（即0到255）来表示字符，并且把（ASCII范围之外的）值128到255分配给特殊字符。这样的一个标准叫做*Latin-1*，广泛地用于西欧地区。在Latin-1中，127以上的字符代码分配给了重音和其他特殊字符。例如，分配给字节值196的字符，是一个特殊标记的非ASCII字符：

```
>>> 0xC4
196
>>> chr(196)
'Ä'
```

这个标准考虑到范围较广的额外特殊字符。然而，一些字母表定义了如此多的字符，以至于无法把其中的每一个都表示成一个字节。Unicode考虑到更多的灵活性。Unicode文本通常叫做“宽字符”字符串，因为每个字符可能表示为多个字节。Unicode通常用在**国际化的**程序中，以表示欧洲和亚洲的字符集，它们往往拥有比8位字节所能表示的更多的字符。

要在计算机内存中存储如此丰富的文本，我们要确保字符与原始字节之间可以使用一种**编码**相互转换，而编码就是把一个Unicode字符转换为字节序列以及从一个字节序列提取字符串的规则。更程序化地说，字节和字符串之间的来回转换由两个术语定义：

- *编码*是根据一个想要的编码名称，把一个字符串翻译为其原始字节形式。
- *解码*是根据其编码名称，把一个原始字节串翻译为字符串形式的过程。

也就是说，我们从字符串**编码**为原始字节，并且从原始字节解码为字符串。对于某些编码，翻译的过程很简单，例如ASCII和Latin-1，把每个字符映射为一个单个字节，因此，不需要翻译工作。对于其他的编码，映射可能更复杂些，并且每个字符产生多个字节。

例如，广为使用的*UTF-8*编码，通过采用可变的字节数的方案，允许表示众多的字符。小于128的字符代码表示为单个字节；128和0x7ff (2047)之间的代码转换为两个字节，而每个字节拥有一个128到255之间的值；0x7ff以上的代码转换为3个或4个字节序列，序列中的每个字节的值在128到255之间。这保持了ASCII字符串的紧凑，避免了字节排序问题，并且避免了可能对C库和网络连接引发问题的空（零）字节。

由于编码的字符映射把字符分配给同样的代码以保持兼容性，因此ASCII是Latin-1和UTF-8的**子集**。也就是说，一个有效的ASCII字符串也是一个有效的Latin-1和UTF-8编码字符串。当数据存储到文件中的时候，这也是成立的：每个ASCII文件也是有效的UTF-8文件，因为ASCII是UTF-8的一个7位的子集。

反过来，对于所有小于128的字符代码，UTF-8编码与ASCII是二进制兼容的。Latin-1和UTF-8只不过是考虑到了额外的字符：Latin-1是为了在一个字节内映射为128到255的值，UTF-8是考虑到可能用多个字节表示的字符串。其他的编码以类似的方式允许较宽的字符集合，但是，所有这些，ASCII、Latin-1、UTF-8以及很多其他的编码，都被认为是Unicode。

对于Python程序员来说，编码指定为包含了编码名的字符串。Python带有大约100种不同的编码，参见Python库参考可以找到一个完整的列表。导入`encodings`模块并运行`help(encodings)`也会显示很多的编码名称，一些是在Python中实现的，一些是在C中实现的。一些编码也有多个名称，例如，*latin-1*、*iso_8859_1*和*8859*都是相同编码的名

称，即Latin-1。我们将会在本章稍后学习在脚本中编写Unicode字符串技术的时候，再次介绍编码。

要了解关于Unicode的更多内容，参见Python标准手册集。它在“Python HOWTOs”部分包括了一个“Unicode HOWTO”，其中提供了额外的背景知识，考虑到篇幅的问题，我们暂时在此省略。

Python的字符串类型

具体来说，Python语言提供了字符串类型在脚本中表示字符文本。在脚本中所使用的字符串类型取决于所使用的Python的版本。*Python 2.X*有一种通用的字符串类型来表示二进制数据和像ASCII这样的8位文本，还有一种特定的类型用来表示多字节Unicode文本：

- `str`表示8位文本和二进制数据。
- `unicode`用来表示宽字符Unicode文本。

Python 2.X的两种字符串类型是不同的（`unicode`考虑到字符的额外大小并且支持编码和解码），但是，它们的操作集大多是重叠的。Python 2.X中的str字符串类型用于可以用8位字节表示的文本，以及绝对字节值所表示的二进制数据。

相反，*Python 3.X*带有3种字符串对象类型——一种用于文本数据，两种用于二进制数据：

- `str`表示Unicode文本（8位的和更宽的）。
- `bytes`表示二进制数据。
- `bytearray`，是一种可变的`bytes`类型。

正如前面所提到的，`bytearray`在Python 2.6中可以使用，但它是从Python 3.0才有的升级功能，此后较少有特定内容的行为，并且通常看做是一个Python 3.0类型。

Python 3.0中所有3种字符串类型都支持类似的操作集，但是，它们都有不同的角色。Python 3.X之后关于这一修改的主要目标是，把Python 2.X中常规的和Unicode字符串类型**合并**到一个单独的字符串类型中，以支持常规的和Unicode文本：开发者想要删除Python 2.X中的字符串区分，并且让Unicode的处理更加自然。假设ASCII和其他的8位文本真的是一种简单的Unicode，这种融合听起来很符合逻辑。

为了实现这一点，Python 3.0的`str`类型定义为一个**不可改变的字符序列**（不一定是字节），这可能是像ASCII这样的每个字符一个字节的常规文本，或者是像UTF-8 Unicode这样可能包含多字节字符的字符集文本。你的脚本所使用的字符串处理，带有这种每个

平台默认编码的类型，但是，在内存中以及在文件之间来回转换的时候，将会提供明确的编码名，以便在str对象和不同的方案之间来回转换。

尽管Python 3.0新的str类型确实实现了想要的字符串/Unicode结合，但很多程序仍然需要处理那些没有针对每个任意文本格式都编码的raw字节数据。图像和声音文件，以及用来与设备接口的打包数据，或者你想要用Python的struct模块处理的C程序，都属于这一类型。因此，为了支持真正的二进制数据的处理，还引入了一种新的类型，即bytes。

在Python 2.X中，通用的str类型填补了二进制数据的这一角色，因为字符串也只是字节的序列（单独的unicode类型处理宽字符串）。在Python 3.0中，bytes类型定义为一个**8位整数的不可变序列**，表示绝对的字节值。此外，Python 3.0的bytes类型支持几乎str类型所做的所有相同操作：这包括字符串方法、序列操作，甚至re模块模式匹配；但是不包括字符串格式化。

一个Python 3.0 bytes对象其实只是较小整数的一个序列，其中每个整数的范围都在0到255之间；索引一个bytes将返回一个int，分片一个bytes将返回另一个bytes，并且在一个bytes上运行内置函数list将返回整数，而不是字符的一个列表。当用那些假设字符的操作处理bytes的时候，bytes对象的内容被假设为ASCII编码的字节（例如，isalpha方法假设每个字节都是一个ASCII字符代码）。此外，为了方便起见，bytes对象打印为字符串而不是整数。

尽管如此，Python的开发者也在Python 3.0中添加了一个bytearray类型，bytearray是bytes类型的一个变体，它是**可变的**并且支持原处修改。它支持str和bytes所支持的常见的字符串操作，以及和列表相同的很多原处修改操作（例如，append和extend方法，以及向索引赋值）。假设字符串可以作为raw字节对待，bytearray最终为字符串数据添加了直接原处可修改的能力，这在Python 2.0中不通过转换为一个可变类型是不可能做到的，并且也是Python 3.0的str或bytes所不支持的。

尽管Python 2.6和Python 3.0提供了很多相同的功能，但它们还是以不同方式包装了功能。实际上，从Python 2.6到Python 3.0的字符串类型映射并不是直接的，Python 2.8的str等同于Python 3.0中的str和bytes，并且Python 3.0的str等同于Python 2.6中的str和Unicode。此外，Python 3.0的可变的bytearray是独特的。

然而，实际上，这种不对称并不像听上去那么令人生畏。它可以概括如下：在Python 2.6中，我们可以对简单的文本使用str并且对文本的更高级的形式使用二进制数据和unicode；在Python 3.0中，我们将针对任何类型的文本（简单的和Unicode）使用str，并且针对二进制数据使用bytes或bytearray。实际上，这种选择常常由你所使用的工具决定，尤其是在文件处理工具的例子中，这是下一小节的主题。

文本和二进制文件

文件I/O（输入和输出）在Python 3.0中也有所改进，以反映str/bytes的区分以及对编码Unicode文本的自动支持。Python现在在文本文件和二进制文件之间做了一个明显的独立于平台的区分：

文本文件

当一个文件以**文本模式**打开的时候，读取其数据会自动将其内容解码（每个平台一个默认的或一个提供的编码名称），并且将其返回为一个`str`，写入会接受一个`str`，并且在将其传输到文件之间自动编码它。文本模式的文件还支持统一的行尾转换和额外的编码特定参数。根据编码名称，文本文件也自动处理文件开始处的字节顺序标记序列（稍后详细介绍）。

二进制文件

通过在内置的open调用的模式字符串参数添加一个`b`（只能小写），以**二进制模式**打开一个文件的时候，读取其数据不会以任何方式解码它，而是直接返回其内容raw并且未经修改，作为一个`bytes`对象；写入类似地接受一个`bytes`对象，并且将其传送到文件中而未经修改。二进制模式文件也接受一个`bytearray`对象作为写入文件中的内容。

由于`str`和`bytes`之间的语言差距明显，所以必须确定数据本质上是文本或二进制，并且在脚本中相应地使用`str`或`bytes`对象来表示其内容。最终，以何种模式打开一个文件将决定脚本使用何种类型的对象来表示其内容：

- 如果正在处理图像文件，其他程序创建的、而且必须解压的打包数据，或者一些设备数据流，则使用`bytes`和**二进制模式**文件处理它更合适。如果想要更新数据而不在内存中产生其副本，也可以选择使用`bytearray`。
- 如果你要处理的内容实质是文本的内容，例如程序输出、HTML、国际化文本或CSV或XML文件，可能要使用`str`和**文本模式**文件。

注意，内置函数`open`的**模式字符串**参数（函数的第二个参数）在Python 3.0中变得至关重要，因为其内容不仅指定了一个**文件处理模式**，而且暗示了一个Python**对象类型**。通过给模式字符串添加一个`b`，我们可以指定二进制模式，并且当读取或写入的时候，将要接收或者必须提供一个`bytes`对象来表示文件的内容。没有`b`，我们的文件将以文本模式处理，并且将使用`str`对象在脚本中表示其内容。例如，模式`rb`、`wb`和`rb+`暗示`bytes`，而`r`、`w+`和`rt`暗示`str`。

文本模式文件也处理在某种编码方案下可能出现在文件开始处的**字节顺序标记**（byte order marker，BOM）序列。例如，在UTF-16和UTF-32编码中，BOM指定大尾还是小

尾格式（基本上，是确定一个位字符串的哪一端最重要）。一般来说，一个UTF-8文本文件也包含了一个BOM来声明它是UTF-8，但并不保证这样。当使用这些编码方法来读取和写入数据的时候，如果BOM有一个通用的编码暗示或者如果提供一个更为具体的编码名来强制这点的话，Python会自动省略或写出BOM。例如，BOM总是针对“utf-16”处理，更具体的编码名“utf-16-le”表示小尾UTF-16格式，更具体的编码名“utf-8-sig”迫使Python在输入和输出上分别都针对UTF-8文本省略并写入一个BOM（通用名称“utf-8”并不这么做）。

我们还将在本章后面的“在Python 3.0中处理BOM”一节中介绍更多有关BOM和文件的内容。首先，让我们探讨Python的新的Unicode字符串模型的含义。

Python 3.0中的字符串应用

让我们再看一些例子，这些例子展示了如何使用Python 3.0字符串类型。提前强调一点：本节中的代码都只在Python 3.0下运行和使用。然而，基本的字符串操作通常在Python各版本中是可移植的。用str类型表示的简单的ASCII字符串在Python 2.6和Python 3.0下都能工作（并且确实像我们在本书第7章中所见到的那样）。此外，尽管Python 2.6中没有bytes类型（它只有通用的str），它通常按照这样的方式来运行代码：在Python 2.6中，调用bytes(X)作为str(X)的同义词出现，并且新的常量形式b'...'看做与常量'...'相同。然而，我们仍然可能在一些个别的例子中遇到版本差异；例如，Python 2.6的bytes调用，不允许Python 3.0中bytes所要求的第二个参数（编码名称）。

常量和基本属性

当调用str或bytes这样的一个内置函数的时候，会引发Python 3.0字符串对象，来处理调用open（下一小节介绍）所创建的一个文件，或者在脚本中编写常量语法。对于后者，一种新的常量形式b'xxx'（以及对等的B'xxx'）用来创建Python 3.0中的bytes对象，bytearray对象可能通过调用bytearray函数来创建，这会带有各种可能的参数。

更正式地说，在Python 3.0中，所有当前字符串常量形式，'xxx'、"xxx"和三引号字符串块，都产生一个str；在它们任何一种前面添加一个b或B，则会创建一个bytes。这个新的b'...'字节常量类似于用来抑制反斜杠转义的r'...' raw字符串。考虑在Python 3.0中运行如下语句：

```
C:\misc> c:\python30\python

>>> B = b'spam'                 # Make a bytes object (8-bit bytes)
>>> S = 'eggs'                  # Make a str object (Unicode characters, 8-bit or wider)

>>> type(B), type(S)
```

```
(<class 'bytes'>, <class 'str'>)

>>> B                          # Prints as a character string, really sequence of ints
b'spam'
>>> S
'eggs'
```

bytes对象实际上是较小的整数的一个序列，尽管它尽可能地将自己的内容打印为字符：

```
>>> B[0], S[0]          # Indexing returns an int for bytes, str for str
(115, 'e')

>>> B[1:], S[1:]        # Slicing makes another bytes or str object
(b'pam', 'ggs')

>>> list(B), list(S)
([115, 112, 97, 109], ['e', 'g', 'g', 's'])      # bytes is really ints
```

bytes对象是不可修改的，就像str（尽管后面将要介绍的bytearray是可以修改的），我们可以把一个str、bytes或整数赋给一个bytes对象的偏移。bytes前缀对于任何字符串常量形式也有效：

```
>>> B[0] = 'x'                                          # Both are immutable
TypeError: 'bytes' object does not support item assignment

>>> S[0] = 'x'
TypeError: 'str' object does not support item assignment

>>> B = B"""            # bytes prefix works on single, double, triple quotes
... xxxx
... yyyy
... """
>>> B
b'\nxxxx\nyyyy\n'
```

正如前面提到的，在Python 2.6中，为了兼容性而使用b'xxx'，但是它与'xxx'是相同的，并且产生一个str，并且，bytes只是str的同义词；正如你已经看到的，在Python 3.0中，这二者都解决了bytes类型之间的差异。还要注意到，Python 2.6中的u'xxx'和U'xxx' Unicode字符串常量形式在Python 3.0中已经取消了，而是使用'xxx'替代，因为所有的字符串都是Unicode，即便它们包含所有的ASCII字符（在本章后面的“编码非ASCII文本”小节将更多地讨论）。

转换

尽管Python 2.X允许str和unicode类型对象自由地混合（如果该字符串只包含7位的ASCII文本的话），Python 3.0引入了一个更鲜明的区分——str和bytes类型对象不在表

达式中自动地混合，并且当传递给函数的时候不会自动地相互转换。期待一个str对象作为参数的函数，通常不能接受一个bytes；反之亦然。

因此，Python 3.0基本上要求遵守一种类型或另一种类型，或者手动执行显式转换：

- str.encode()和bytes(S, encoding)把一个字符串转换为其raw bytes形式，并且在此过程中根据一个str创建一个bytes。
- bytes.decode()和str(B, encoding)把raw bytes转换为其字符串形式，并且在此过程中根据一个bytes创建一个str。

encode和decode方法（以及文件对象，将在下一节介绍）针对你的平台使用一个默认编码，或者一个显式传入的编码名。例如，在Python 3.0中：

```
>>> S = 'eggs'
>>> S.encode()                          # str to bytes: encode text into raw bytes
b'eggs'

>>> bytes(S, encoding='ascii')          # str to bytes, alternative
b'eggs'

>>> B = b'spam'
>>> B.decode()                          # bytes to str: decode raw bytes into text
'spam'

>>> str(B, encoding='ascii')            # bytes to str, alternative
'spam'
```

这里有两点要注意。首先，平台的默认编码在sys模块中可用，但是，bytes的编码参数不是可选的，即便它在str.encode（和bytes.decode)中亦是如此。

其次，尽管调用str并不像bytes那样要求编码参数，但在str调用中省略它并不意味着它是默认的，相反，不带编码的一个str调用返回bytes对象的打印字符串，而不是其str转换后的形式（这通常不是我们想要的！）假设B和S仍然和前面相同:

```
>>> import sys
>>> sys.platform                        # Underlying platform
'win32'
>>> sys.getdefaultencoding()            # Default encoding for str here
'utf-8'

>>> bytes(S)
TypeError: string argument without an encoding

>>> str(B)                              # str without encoding
"b'spam'"                               # A print string, not conversion!
>>> len(str(B))
7
>>> len(str(B, encoding='ascii'))       # Use encoding to convert to str
4
```

编码Unicode字符串

当我们开始处理真正的非ASCII Unicode文本的时候，编码和解码变得更有意义了。要在字符串中编码任意的Unicode字符，有些字符可能甚至无法在键盘上输入，Python的字符串常量支持"\xNN"十六进制字节值转义以及"\uNNNN"和"\UNNNNNNNN" Unicode转义。在Unicode转义中，第一种形式给出了4个十六进制位以编码1个2字节（16位）字符码，而第二种形式给出8个十六进制位表示4字节（32位）代码。

编码ASCII文本

让我们来看一些例子以介绍文本编码的基础知识。正如我们已经介绍过的，ASCII文本是一种简单的Unicode，存储为表示字符的字节值的一个序列：

```
C:\misc> c:\python30\python

>>> ord('X')                    # 'X' has binary value 88 in the default encoding
88
>>> chr(88)                     # 88 stands for character 'X'
'X'

>>> S = 'XYZ'                   # A Unicode string of ASCII text
>>> S
'XYZ'
>>> len(S)                      # 3 characters long
3
>>> [ord(c) for c in S]         # 3 bytes with integer ordinal values
[88, 89, 90]
```

像这样的常规7位ASCII文本，在本章前面所介绍的每种Unicode编码方案中，都是以每字节一个字符的方式表现：

```
>>> S.encode('ascii')           # Values 0..127 in 1 byte (7 bits) each
b'XYZ'
>>> S.encode('latin-1')         # Values 0..255 in 1 byte (8 bits) each
b'XYZ'
>>> S.encode('utf-8')           # Values 0..127 in 1 byte, 128..2047 in 2, others 3 or 4
b'XYZ'
```

实际上，以这种方法编码ASCII文本而返回的bytes对象，其实是较短整数的一个序列，只不过这个序列尽可能地打印为ASCII字符：

```
>>> S.encode('latin-1')[0]
88
>>> list(S.encode('latin-1'))
[88, 89, 90]
```

编码非ASCII文本

要编码非ASCII字符，可能在字符串中使用十六进制或Unicode转义；十六进制转义限制于单个字节的值，但Unicode转义可以指定其值有两个和四个字节宽度的字符。例如，十六进制值0xCD 和0xE8，是ASCII的7位字符范围之外的两个特殊的重音字符，但是，我们可以将其嵌入Python 3.0的str对象中，因为如今的str支持Unicode：

```
>>> chr(0xc4)                          # 0xC4, 0xE8: characters outside ASCII's range
'Ä'
>>> chr(0xe8)
'è'

>>> S = '\xc4\xe8'                     # Single byte 8-bit hex escapes
>>> S
'Äè'

>>> S = '\u00c4\u00e8'                 # 16-bit Unicode escapes
>>> S
'Äè'
>>> len(S)                             # 2 characters long (not number of bytes!)
2
```

编码和解码非ASCII文本

现在，如果我们试图把一个非ASCII字符串**编码**为raw字节以像ASCII一样使用，我们会得到一个错误。像Latin-1这样的编码是有效的，并且为每个字符分配一个字节；像UTF-8这样的编码为每个字符分配2个字节。如果把这个字符串写入一个文件，这里显示的raw字节就是针对给定的编码类型而实际存储在文件中的内容：

```
>>> S = '\u00c4\u00e8'
>>> S
'Äè'
>>> len(S)
2

>>> S.encode('ascii')
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
ordinal not in range(128)

>>> S.encode('latin-1')                # One byte per character
b'\xc4\xe8'

>>> S.encode('utf-8')                  # Two bytes per character
b'\xc3\x84\xc3\xa8'

>>> len(S.encode('latin-1'))           # 2 bytes in latin-1, 4 in utf-8
2
>>> len(S.encode('utf-8'))
4
```

也可以用其他的办法，从一个文件读入raw字节并且将其**解码**回一个Unicode字符串。然而，正如我们随后将看到的，给open调用的编码模式会引发对输入自动使用这一解码（避免了在按照字节块读取的时候，由于读取部分字符序列可能引发的问题）：

```
>>> B = b'\xc4\xe8'
>>> B
b'\xc4\xe8'

>>> len(B)                                  # 2 raw bytes, 2 characters
2
>>> B.decode('latin-1')                     # Decode to latin-1 text
'Äè'

>>> B = b'\xc3\x84\xc3\xa8'
>>> len(B)                                  # 4 raw bytes
4
>>> B.decode('utf-8')
'Äè'
>>> len(B.decode('utf-8'))                  # 2 Unicode characters
2
```

其他Unicode编码技术

一些编码甚至使用较大的字节序列来表示字符。当需要的时候，我们可以为自己字符串中的字符指定16位或32位的Unicode值，例如，对于前者使用"\u..."表示4个十六进制位，对于后者使用"\U...."表示8个十六进制位。

```
>>> S = 'A\u00c4B\U000000e8C'
>>> S                                       # A, B, C, and 2 non-ASCII characters
'AÄBèC'
>>> len(S)                                  # 5 characters long
5

>>> S.encode('latin-1')
b'A\xc4B\xe8C'
>>> len(S.encode('latin-1'))                # 5 bytes in latin-1
5

>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'
>>> len(S.encode('utf-8'))                  # 7 bytes in utf-8
7
```

有趣的是，另一些编码可能使用有很大不同的字节格式。例如，cp500 EBCDIC编码，它编码ASCII的方式，甚至和我们已经见过的方法都不同（由于Python为我们编码和解码，所以我们通常只是在提供编码名的时候才需要关注这一点）：

```
>>> S
'AÄBèC'
>>> S.encode('cp500')                       # Two other Western European encodings
```

```
b'\xc1c\xc2T\xc3'
>>> S.encode('cp850')                    # 5 bytes each
b'A\x8eB\x8aC'

>>> S = 'spam'                           # ASCII text is the same in most
>>> S.encode('latin-1')
b'spam'
>>> S.encode('utf-8')
b'spam'
>>> S.encode('cp500')                    # But not in cp500: IBM EBCDIC!
b'\xa2\x97\x81\x94'
>>> S.encode('cp850')
b'spam'
```

从技术上讲，你也可以使用chr而不是Unicode转义或十六进制转义来构建Unicode字符串片段，但是，对于较大的字符串来说，这可能变得很繁琐：

```
>>> S = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'
>>> S
'AÄBèC'
```

这里有两点要注意。首先，Python 3.0允许特殊的字符以十六进制和Unicode转义的方式编码到str字符串中，但是，只能以十六进制转义的方式编码到bytes字符串中：Unicode转义会默默地逐字转换为字节常量，而不是转义。实际上，bytes必须编码为str字符串，以便将其打印为非ASCII字符：

```
>>> S = 'A\xC4B\xE8C'                   # str recognizes hex and Unicode escapes
>>> S
'AÄBèC'

>>> S = 'A\u00C4B\U000000E8C'
>>> S
'AÄBèC'

>>> B = b'A\xC4B\xE8C'                  # bytes recognizes hex but not Unicode
>>> B
b'A\xc4B\xe8C'

>>> B = b'A\u00C4B\U000000E8C'          # Escape sequences taken literally!
>>> B
b'A\\u00C4B\\U000000E8C'

>>> B = b'A\xC4B\xE8C'                  # Use hex escapes for bytes
>>> B                                   # Prints non-ASCII as hex
b'A\xc4B\xe8C'
>>> print(B)
b'A\xc4B\xe8C'
>>> B.decode('latin-1')                 # Decode as latin-1 to interpret as text
'AÄBèC'
```

其次，字节常量要求字符要么是ASCII字符，要么如果它们的值大于127就进行转义。另

一方面，str字符串允许常量包含源字符集中的任何字符（稍后讨论，除非在源文件中给定一个编码声明，否则默认为UTF-8）：

```
>>> S = 'AÄBèC'                          # Chars from UTF-8 if no encoding declaration
>>> S
'AÄBèC'

>>> B = b'AÄBèC'
SyntaxError: bytes can only contain ASCII literal characters.

>>> B = b'A\xC4B\xE8C'                   # Chars must be ASCII, or escapes
>>> B
b'A\xc4B\xe8C'
>>> B.decode('latin-1')
'AÄBèC'

>>> S.encode()                           # Source code encoded per UTF-8 by default
b'A\xc3\x84B\xc3\xa8C'                   # Uses system default to encode, unless passed
>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'

>>> B.decode()                           # Raw bytes do not correspond to utf-8
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: ...
```

转换编码

到目前为止，我们已经编码和解码字符串以查看其结构。更一般地讲，我们总是可以把一个字符串转换为不同于源字符集默认的一种编码，但是，我们必须显式地提供一个编码名称以进行编码和解码：

```
>>> S = 'AÄBèC'
>>> S
'AÄBèC'
>>> S.encode()                           # Default utf-8 encoding
b'A\xc3\x84B\xc3\xa8C'

>>> T = S.encode('cp500')                # Convert to EBCDIC
>>> T
b'\xc1c\xc2T\xc3'

>>> U = T.decode('cp500')                # Convert back to Unicode
>>> U
'AÄBèC'

>>> U.encode()                           # Default utf-8 encoding again
b'A\xc3\x84B\xc3\xa8C'
```

记住，只有当手动编写非ASCII Unicode的时候，才必须用到特殊的Unicode和十六进制字符转义。实际上，我们往往从文件载入这样的文本。正如我们将从本书稍后见到的，Python 3.0的文件对象（用open内置函数创建的）在读取文本字符串的时候自动地编码

它们，并且在写入文本字符串的时候自动解码它们。因此，脚本往往可以广泛地处理字符串，而不必直接编码特殊字符。

在本章稍后我们还将看到，从文件传出和向文件传入字符串的时候，也可以在编码之间进行转换，使用与上一个例子中非常类似的一种技术。尽管在打开一个文件的时候仍然需要显式地提供编码名称，但文件接口自动完成大多数转换工作。

在Python 2.6中编码Unicode字符串

既然已经介绍了Python 3.0中基本的Unicode字符串知识，我需要说明的是，在Python 2.6中可以做很多相同的事情，尽管所使用的工具是不同的。`unicode`在Python 2.6中是可用的，但它是与`str`截然不同的数据类型，并且当常规字符串和Unicode字符串兼容的时候，它允许自由组合。实际上，当遇到把raw字节编码为一个Unicode字符串的时候，我们基本上可以把Python 2.6的`str`当做Python 3.0的`bytes`，只要它保持正确的形式。如下是在Python 2.6中的实际应用（本章所有其他节都是在Python 3.0下运行的情况）：

```
C:\misc> c:\python26\python
>>> import sys
>>> sys.version
'2.6 (r26:66721, Oct 2 2008, 11:35:03) [MSC v.1500 32 bit (Intel)]'

>>> S = 'A\xC4B\xE8C'                    # String of 8-bit bytes
>>> print S                              # Some are non-ASCII
AÄBèC

>>> S.decode('latin-1')                  # Decode byte to latin-1 Unicode
u'A\xc4B\xe8C'

>>> S.decode('utf-8')                    # Not formatted as utf-8
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: invalid data

>>> S.decode('ascii')                    # Outside ASCII range
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1: ordinal
not in range(128)
```

为了存储任意的编码的Unicode文本，用`u'xxx'`常量形式创建一个unicode对象（这个常量在Python 3.0中不再可用，因为Python 3.0中所有字符串都支持Unicode）：

```
>>> U = u'A\xC4B\xE8C'                   # Make Unicode string, hex escapes
>>> U
u'A\xc4B\xe8C'
>>> print U
AÄBèC
```

一旦创建了它，可以把Unicode文本转换为不同的raw字节编码，这类似于在Python 3.0中把`str`对象编码为`bytes`对象：

```
>>> U.encode('latin-1')                    # Encode per latin-1: 8-bit bytes
'A\xc4B\xe8C'
>>> U.encode('utf-8')                      # Encode per utf-8: multibyte
'A\xc3\x84B\xc3\xa8C'
```

在Python 2.6中，非ASCII字符可以用十六进制或Unicode转义来编写到字符串常量中，就像在Python 3.0中一样。然而，和Python 3.0中的bytes一样，在Python 2.6中，"\u..."和"\U..."转义只是识别为unicode字符串，而不是8位str字符串：

```
C:\misc> c:\python26\python
>>> U = u'A\xC4B\xE8C'                     # Hex escapes for non-ASCII
>>> U
u'A\xc4B\xe8C'
>>> print U
AÄBèC

>>> U = u'A\u00C4B\U000000E8C'             # Unicode escapes for non-ASCII
>>> U                                      # u'' = 16 bits, U'' = 32 bits
u'A\xc4B\xe8C'
>>> print U
AÄBèC

>>> S = 'A\xC4B\xE8C'                      # Hex escapes work
>>> S
'A\xc4B\xe8C'
>>> print S                                # But some print oddly, unless decoded
A-BFC
>>> print S.decode('latin-1')
AÄBèC

>>> S = 'A\u00C4B\U000000E8C'              # Not Unicode escapes: taken literally!
>>> S
'A\\u00C4B\\U000000E8C'
>>> print S
A\u00C4B\U000000E8C
>>> len(S)
19
```

就像Python 3.0中的str和bytes一样，Python 2.6的unicode和str共享几乎相同的操作集，因此，除非你需要转换为其他的编码，通常可以把unicode当做是str一样对待。然而，Python 2.6和Python 3.0之间的一个主要区别在于，unicode和非Unicode的str对象可以在表达式中自由地**混合**，并且，只要str和unicode的编码兼容，Python将自动将其向上转换为unicode（在Python 3.0中，str和bytes不会自动混合，并且需要手动转换）：

```
>>> u'ab' + 'cd'                           # Can mix if compatible in 2.6
u'abcd'                                    # 'ab' + b'cd' not allowed in 3.0
```

实际上，类型的不同对于Python 2.6中的代码往往很小。像常规的字符串一样，Unicode字符串也可以合并、索引、分片，用re模块匹配，等等，并且它们不能原处修改。如果需要在两种类型之间显式地转换，可以使用内置的str和unicode函数：

```
>>> str(u'spam')                          # Unicode to normal
'spam'
>>> unicode('spam')                       # Normal to Unicode
u'spam'
```

然而，Python 2.6中这种自由混合字符串类型的方法，只有在字符串和unicode对象的编码类型兼容的情况下才有效：

```
>>> S = 'A\xC4B\xE8C'                    # Can't mix if incompatible
>>> U = u'A\xC4B\xE8C'
>>> S + U
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1: ordinal
not in range(128)

>>> S.decode('latin-1') + U              # Manual conversion still required
u'A\xc4B\xe8CA\xc4B\xe8C'

>>> print S.decode('latin-1') + U
AÄBèCAÄBèC
```

最后，正如我们将在本章稍后更具体介绍的，Python 2.6的open调用只支持8位字节的文件，将其内容返回为str字符串；将其内容解释为文本，还是解释为二进制数并根据需要解码，这取决于你。要读取和编写Unicode文件并自动编码或解码器内容，使用Python 2.6的codecs.open调用，Python 2.6的库手册中有所介绍。这个调用提供了与Python 3.0的open相同的功能，并且使用Python 2.6的unicode对象来表示文件内容——读取一个文件，把编码的字节翻译为解码的Unicode字符，并且在文件打开的时候把翻译字符串写入想要的指定编码。

源文件字符集编码声明

Unicode转义代码对于字符串常量中偶尔出现的Unicode字符很好用，但是，如果需要频繁地在字符串中嵌入非ASCII文本的话，这会变得很繁琐。对于在脚本文件中编码的字符串，Python默认地使用UTF-8编码，但是，它允许我们通过包含一个注释来指明想要的编码，从而将默认值修改为支持任意的字符集。这个注释必须拥有如下的形式，并且在Python 2.6或Python 3.0中必须作为脚本的第一行或第二行出现：

```
# -*- coding: latin-1 -*-
```

当出现这种形式的注释时，Python将自然按照给定的编码来识别表示的字符串。这意味着，我们可以在一个文本编辑中编辑脚本文件来正确地接受和显示重音及其他非ASCII字符，并且Python将在字符串常量中正确地解码它们。例如，注意如下文件*text.py*的顶部，注释是如何允许Latin-1字符嵌入字符串中的：

```
 # -*- coding: latin-1 -*-
```

```
# Any of the following string literal forms work in latin-1.
# Changing the encoding above to either ascii or utf-8 fails,
# because the 0xc4 and 0xe8 in myStr1 are not valid in either.

myStr1 = 'aÄBèC'

myStr2 = 'A\u00c4B\U000000e8C'

myStr3 = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'

import sys
print('Default encoding:', sys.getdefaultencoding())

for aStr in myStr1, myStr2, myStr3:
print('{0}, strlen={1}, '.format(aStr, len(aStr)), end='')

bytes1 = aStr.encode()                          # Per default utf-8: 2 bytes for non-ASCII
bytes2 = aStr.encode('latin-1')                 # One byte per char
#bytes3 = aStr.encode('ascii')                  # ASCII fails: outside 0..127 range

print('byteslen1={0}, byteslen2={1}'.format(len(bytes1), len(bytes2)))
```

运行这段脚本，将产生如下输出：

```
C:\misc> c:\python30\python text.py
Default encoding: utf-8
aÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
```

由于大多数程序员可能都遵从标准的UTF-8编码，所以关于这一选项以及其他高级Unicode支持的话题，例如字符串中的属性名和字符名转义，我们参考Python的标准手册以了解更多细节。

使用Python 3.0 Bytes对象

我们在第7章学习了各种针对Python 3.0通用str对象的操作，基本的字符串类型在Python 2.6和Python 3.0中都能同样地工作，因此，我们不会再回顾这一主题。相反，让我们深入一步介绍Python 3.0中新的bytes类型所提供的操作集。

正如前面提到的，Python 3.0 bytes对象是较小整数的一个序列，其中每个整数都在0到255之间，并且在显示的时候恰好打印为ASCII字符。它支持序列操作以及str对象（在Python 2.X中是str类型）上可用的大多数同样方法。然而，bytes不支持格式化方法或%格式化表达式，并且，不能不经过显式转换就将bytes和str类型对象混合和匹配——我们通常使用针对**文本数据**使用所有的str类型对象和文本文件，并且针对**二进制数据**使用所有的bytes对象和二进制文件。

方法调用

如果你真的想要看看str拥有哪些bytes所没有的属性，总是可以查看它们的dir内置函数调用结果。输出结果能够告诉你有关它们所支持的表达式操作符的一些事情（例如，__mod__和__rmod__实现了%操作符）：

```
C:\misc> c:\python30\python

# Attributes unique to str

>>> set(dir('abc')) - set(dir(b'abc'))
{'isprintable', 'format', '__mod__', 'encode', 'isidentifier',
'_formatter_field_name_split', 'isnumeric', '__rmod__', 'isdecimal',
'_formatter_parser', 'maketrans'}

# Attributes unique to bytes

>>> set(dir(b'abc')) - set(dir('abc'))
{'decode', 'fromhex'}
```

正如你所看到的，str和bytes拥有几乎相同的功能。它们唯一的属性是那些不能应用于对方的通用方法；例如，decode把一个raw bytes转换为其str表示，并且encode把一个字符串转换为其raw bytes表示。大多数这样的方法是相同的，尽管bytes方法需要bytes参数（再次，Python 3.0的字符串类型不能混合）。此外，还要记住，bytes对象是不可改变的，就像是Python 2.6和Python 3.0中的str对象一样（为了简单起见，这里简化了出错消息）：

```
>>> B = b'spam'                       # b'...' bytes literal
>>> B.find(b'pa')
1

>>> B.replace(b'pa', b'XY')           # bytes methods expect bytes arguments
b'sXYm'

>>> B.split(b'pa')
[b's', b'm']

>>> B
b'spam'

>>> B[0] = 'x'
TypeError: 'bytes' object does not support item assignment
```

一个显著的区别是，**字符串格式化**只在Python 3.0中对str有效，对bytes对象无效（参见本书第7章了解关于字符串格式化表达式和方法的更多内容）：

```
>>> b'%s' % 99
TypeError: unsupported operand type(s) for %: 'bytes' and 'int'

>>> '%s' % 99
```

```
'99'

>>> b'{0}'.format(99)
AttributeError: 'bytes' object has no attribute 'format'

>>> '{0}'.format(99)
'99'
```

序列操作

除了方法调用，我们知道的在Python 2.X中用于字符串和列表的所有常见通用序列操作，也都期待在Python 3.0的str和bytes上有效，这包括索引、分片、合并，等等。注意，如下的代码索引一个bytes对象并返回一个给出了该字节的二进制值的整数；bytes实际上是8位整数的一个序列，但是，当作为整体显示的时候，为了方便起见，它打印为ASCII编码的字符的一个字符串。要查看一个给定的字节的值，使用chr内置函数来将其转换回字符，如下所示：

```
>>> B = b'spam'                          # A sequence of small ints
>>> B                                    # Prints as ASCII characters
b'spam'

>>> B[0]                                 # Indexing yields an int
115
>>> B[-1]
109

>>> chr(B[0])                            # Show character for int
's'
>>> list(B)                              # Show all the byte's int values
[115, 112, 97, 109]

>>> B[1:], B[:-1]
(b'pam', b'spa')

>>> len(B)
4

>>> B + b'lmn'
b'spamlmn'
>>> B * 4
b'spamspamspamspam'
```

创建bytes对象的其他方式

到目前为止，我们已经见到了用b'...'常量语法创建bytes对象的主要方式；也可以用一个str和一个编码名来调用bytes构造函数，用一个可迭代的整数表示的字节值来调用bytes构造函数，或者按照每个默认（或传入的）编码来编码一个str对象，从而创建bytes对象。正如我们已经见到过的，编码会接受一个str并根据编码声明来返回该字

符串的raw二进制字节值；相反，解码会接受一个raw bytes序列并将其编码为字符串表示，即一系列可能宽度的字符。这两种操作都会创建新的字符串对象：

```
>>> B = b'abc'
>>> B
b'abc'

>>> B = bytes('abc', 'ascii')
>>> B
b'abc'

>>> ord('a')
97
>>> B = bytes([97, 98, 99])
>>> B
b'abc'

>>> B = 'spam'.encode()                    # Or bytes()
>>> B
b'spam'
>>>
>>> S = B.decode()                         # Or str()
>>> S
'spam'
```

从更大的角度来看，这些操作的最后两个是在str和bytes之间转换的真正工具，这是前面介绍过的话题，下一小节还将继续展开介绍。

混合字符串类型

在前面的“方法调用”小节的replace调用中，我们必须传入两个bytes对象，str对象在这里无效。尽管Python 2.X尽可能自动在str和unicode之间转换（例如，当str是一个7位ASCII文本的时候），Python 3.0需要在某些环境下要求特殊的字符串类型并且如果需要的话期待手动转换：

```
# Must pass expected types to function and method calls

>>> B = b'spam'

>>> B.replace('pa', 'XY')
TypeError: expected an object with the buffer interface

>>> B.replace(b'pa', b'XY')
b'sXYm'

>>> B = B'spam'
>>> B.replace(bytes('pa'), bytes('xy'))
TypeError: string argument without an encoding

>>> B.replace(bytes('pa', 'ascii'), bytes('xy', 'utf-8'))
b'sxym'
```

```
# Must convert manually in mixed-type expressions

>>> b'ab' + 'cd'
TypeError: can't concat bytes to str

>>> b'ab'.decode() + 'cd'                          # bytes to str
'abcd'

>>> b'ab' + 'cd'.encode()                          # str to bytes
b'abcd'

>>> b'ab' + bytes('cd', 'ascii')                   # str to bytes
b'abcd'
```

尽管你可以自行创建bytes对象，以表示打包的二进制数据，但它们也可以在二进制模式下读取打开的文件从而自动创建对象，正如我们将在本章稍后详细介绍的。首先，我们应该介绍bytes的亲密且可变的“近亲”。

使用Python 3.0（和Python 2.6）bytearray对象

到目前为止，我们已经关注了str和bytes，因为它们包含了Python 2.X的unicode和str。然而，Python 3.0还有第三个字符串类型bytearray，这是范围在0到255之间的整数的一个可变的序列，其本质是bytes的可变的变体。同样，它支持和bytes同样的字符串方法和序列操作，并且与列表支持同样多的可变的原处修改操作。bytearray类型在Python 2.6中也可用，作为来自Python 3.0的一个功能升级，但是，它不像Python 3.0中那样实施严格的文本/二进制区分。

让我们来快速看看。在Python 2.6中，bytearray对象可以通过调用bytearray内置函数来创建，并且可以用字符串来初始化：

```
# Creation in 2.6: a mutable sequence of small (0..255) ints

>>> S = 'spam'
>>> C = bytearray(S)                    # A back-port from 3.0 in 2.6
>>> C                                   # b'..' == '..' in 2.6 (str)
bytearray(b'spam')
```

在Python 3.0中，需要一个编码名称和字节字符串，因为文本和二进制字符串不能混合，尽管字节字符串能够反映编码的Unicode文本：

```
# Creation in 3.0: text/binary do not mix

>>> S = 'spam'
>>> C = bytearray(S)
TypeError: string argument without an encoding

>>> C = bytearray(S, 'latin1')          # A content-specific type in 3.0
>>> C
```

```
bytearray(b'spam')

>>> B = b'spam'                          # b'..' != '..' in 3.0 (bytes/str)
>>> C = bytearray(B)
>>> C
bytearray(b'spam')
```

创建之后，bytearray对象像bytes一样也是较小的整数序列，并且可以像列表一样修改，尽管它们需要一个整数而不是一个字符串进行索引赋值（如下的例子都是本节内容的延续，并且除非特别声明，都是在Python 3.0下运行，参见关于Python 2.6用法提示的注释）：

```
# Mutable, but must assign ints, not strings

>>> C[0]
115

>>> C[0] = 'x'                           # This and the next work in 2.6
TypeError: an integer is required

>>> C[0] = b'x'
TypeError: an integer is required

>>> C[0] = ord('x')
>>> C
bytearray(b'xpam')

>>> C[1] = b'Y'[0]
>>> C
bytearray(b'xYam')
```

处理bytearray对象借用了字符串和列表的方法，因为它们都是可修改的字节字符串。除了命名方法，bytearray中的__iadd__和__setitem__方法分别实现了+=原处连接和索引赋值：

```
# Methods overlap with both str and bytes, but also has list's mutable methods

>>> set(dir(b'abc')) - set(dir(bytearray(b'abc')))
{'__getnewargs__'}

>>> set(dir(bytearray(b'abc'))) - set(dir(b'abc'))
{'insert', '__alloc__', 'reverse', 'extend', '__delitem__', 'pop', '__setitem__'
, '__iadd__', 'remove', 'append', '__imul__'}
```

我们可以使用两种索引赋值来原处修改一个bytearray，正如已经看到的，并且类似列表的方法如下所示（要在Python 2.6中原处修改文本，可能需要使用list(str)和''.join(list)，将其转换为一个列表并转换回来）：

```
# Mutable method calls

>>> C
```

```
bytearray(b'xYam')

>>> C.append(b'LMN')                    # 2.6 requires string of size 1
TypeError: an integer is required

>>> C.append(ord('L'))
>>> C
bytearray(b'xYamL')

>>> C.extend(b'MNO')
>>> C
bytearray(b'xYamLMNO')
```

所有常见的序列操作和字符串方法都在bytearrays上有效，正如我们所期待的那样（注意，就像bytes对象一样，其表达式和方法期待bytes参数，而不是str参数）：

```
# Sequence operations and string methods

>>> C + b'!#'
bytearray(b'xYamLMNO!#')

>>> C[0]
120

>>> C[1:]
bytearray(b'YamLMNO')

>>> len(C)
8

>>> C
bytearray(b'xYamLMNO')

>>> C.replace('xY', 'sp')                    # This works in 2.6
TypeError: Type str doesn't support the buffer API

>>> C.replace(b'xY', b'sp')
bytearray(b'spamLMNO')

>>> C
bytearray(b'xYamLMNO')

>>> C * 4
bytearray(b'xYamLMNOxYamLMNOxYamLMNOxYamLMNO')
```

最后，概括起来，下面的例子展示了bytes和bytearray对象如何是int的序列，而str对象是字符的序列：

```
# Binary versus text

>>> B                                        # B is same as S in 2.6
b'spam'
>>> list(B)
[115, 112, 97, 109]
```

```
>>> C
bytearray(b'xYamLMNO')
>>> list(C)
[120, 89, 97, 109, 76, 77, 78, 79]

>>> S
'spam'
>>> list(S)
['s', 'p', 'a', 'm']
```

尽管Python 3.0中所有的三种字符串类型都可以包含字符值并且支持很多相同的操作，但我们总是应该：

- 对文本数据使用str；
- 对二进制数据使用bytes；
- 对想要原处修改的二进制数据使用bytearray。

像文件这样的相关工具，常常也是我们的选择，这是下一节将要介绍的主题。

使用文本文件和二进制文件

本节进一步介绍Python 3.0的字符串模型对本书前面所介绍的文件处理基础知识的影响。正如前面提到的，关键是用哪种模式打开一个文件，它决定了在脚本中将要使用哪种对象类型表示文件的内容。文本模式意味着str对象，二进制模式意味着bytes对象：

- *文本模式文件*根据Unicode**编码**来解释文件内容，要么是平台的默认编码，要么是我们传递进的编码名。通过传递一个编码名来打开文件，我们可以强行进行Unicode文件的各种类型的转换。文本模型的文件也执行通用的**行末转换**：默认地，所有的行末形式映射为脚本中的一个单个的'\n'字符，而不管在什么平台上运行。正如前面所描述的，文本文件也负责阅读和写入在某些Unicode编码方案中存储文件开始处的**字节顺序标记**（Byte Order Mark，BOM）。
- *二进制模式文件*不会返回**原始的**文件内容，而是作为表示字节值的整数的一个序列，没有编码或解码，也没有行末转换。

open的第二个参数是想要处理文本文件还是二进制文件，就像在Python 2.X中所做的一样，给这个字符串添加一个“b”表示二进制模式（例如，"rb"表示读取二进制数据文件）。默认的模式是"rt"，这等同于"r"，意味着文本输入（就像在Python 2.X中一样）。

然而，在Python 3.0中，open的这种模式参数也意味着文件内容表示的一个对象类型，而不管底层的平台是什么——文本文件返回一个str供读取，并且期待一个str以写入；但二进制文件返回一个bytes供读取，并且期待一个bytes（或bytearray）供写入。

文本文件基础

为了便于展示，让我们从基本的文件I/O开始。只要你打算处理基本的文本文件（例如，ASCII）并且不担心绕过平台默认的字符串编码，Python 3.0中文件的显示和处理和它们在Python 2.X中有很大相似之处（由此可见，字符串是通用的）。例如，如下示例在Python 3.0中把一行文本写入一个文件并将其读取回来，和在Python 2.6中的处理是一样的（注意，Python 3.0中`file`不再是内置的名称，因此，这里使用它作为变量完全没有问题）：

```
C:\misc> c:\python30\python

# Basic text files (and strings) work the same as in 2.X

>>> file = open('temp', 'w')
>>> size = file.write('abc\n')          # Returns number of bytes written
>>> file.close()                        # Manual close to flush output buffer

>>> file = open('temp')                 # Default mode is "r" (== "rt"): text input
>>> text = file.read()
>>> text
'abc\n'
>>> print(text)
abc
```

Python 3.0中的文本和二进制模式

在Python 2.6中，文本文件和二进制文件之间没有主要区别——都是接受并返回作为`str`字符串的内容。唯一的主要区别是，在Windows下文本文件自动把`\n`行末字符和`\r\n`相互映射，而二进制文件不这么做（这里，为了简单起见，我们把操作连接到了一行之中）：

```
C:\misc> c:\python26\python
>>> open('temp', 'w').write('abd\n')          # Write in text mode: adds \r
>>> open('temp', 'r').read()                  # Read in text mode: drops \r
'abd\n'
>>> open('temp', 'rb').read()                 # Read in binary mode: verbatim
'abd\r\n'

>>> open('temp', 'wb').write('abc\n')         # Write in binary mode
>>> open('temp', 'r').read()                  # \n not expanded to \r\n
'abc\n'
>>> open('temp', 'rb').read()
'abc\n'
```

在Python 3.0中，情况稍微复杂一些，因为用于文本数据的`str`和用于二进制数据的`bytes`之间存在区别。为了说明这点，让我们写入一个**文本文件**并在Python中以两种模式来读取它。注意，我们需要为写入提供一个`str`，但是，根据打开模式，读取给我们一个`str`或`bytes`：

```
C:\misc> c:\python30\python

# Write and read a text file

>>> open('temp', 'w').write('abc\n')              # Text mode output, provide a str
4

>>> open('temp', 'r').read()                      # Text mode input, returns a str
'abc\n'

>>> open('temp', 'rb').read()                     # Binary mode input, returns a bytes
b'abc\r\n'
```

注意，在Windows上，文本模式的文件是在输出中如何把\n**行末符号**转换为\r\n的；在输入上，文本模式把\r\n转换回\n，但二进制模式不会这么做。这在Python 2.6中是一样的，并且，这也是我们希望对二进制数据所做的，尽管如果我们愿意的话，可以在Python 3.0中用额外的open参数控制这一行为。

现在，让我们再次做同样的事情，但是是对**二进制文件**来做。我们提供一个bytes以便在这个例子中写入，并且根据输入模式，我们仍然得到一个str或一个bytes：

```
# Write and read a binary file

>>> open('temp', 'wb').write(b'abc\n')            # Binary mode output, provide a bytes
4

>>> open('temp', 'r').read()                      # Text mode input, returns a str
'abc\n'

>>> open('temp', 'rb').read()                     # Binary mode input, returns a bytes
b'abc\n'
```

注意，在二进制模式输出中，\n行末字符没有扩展为\r\n——再一次说明，这是二进制数据想要的结果。即便我们要写入二进制文件中的数据本身真的是二进制的，类型需求和文件行为还是相同的。例如，在下面的例子中，"\x00"是二进制0字节并且不是一个可打印的字符：

```
# Write and read truly binary data

>>> open('temp', 'wb').write(b'a\x00c')           # Provide a bytes
3

>>> open('temp', 'r').read()                      # Receive a str
'a\x00c'

>>> open('temp', 'rb').read()                     # Receive a bytes
b'a\x00c'
```

二进制模式文件总是作为一个bytes对象返回内容，但是接受一个bytes或bytearray对象以供写入。既然bytearray基本上是bytes的一个可变的变体，自然就遵从这一方式。实际上，Python 3.0中的大多数API接受一个bytes，也允许一个bytearray：

```
# bytearrays work too

>>> BA = bytearray(b'\x01\x02\x03')

>>> open('temp', 'wb').write(BA)
3

>>> open('temp', 'r').read()
'\x01\x02\x03'

>>> open('temp', 'rb').read()
b'\x01\x02\x03'
```

类型和内容错误匹配

注意，当遇到文件的时候，我们不能违反Python的str/bytes类型差异并侥幸成功。正如如下的例子所示，如果试图向一个文本文件写入一个bytes或者向二进制文件写入一个str，将会得到错误（这里缩写了出错信息）：

```
# Types are not flexible for file content

>>> open('temp', 'w').write('abc\n')             # Text mode makes and requires str
4
>>> open('temp', 'w').write(b'abc\n')
TypeError: can't write bytes to text stream

>>> open('temp', 'wb').write(b'abc\n')           # Binary mode makes and requires bytes
4
>>> open('temp', 'wb').write('abc\n')
TypeError: can't write str to binary stream
```

这是有意义的：对于二进制模式，在文本编码之前，它是没有意义的。尽管往往可能通过编码str和解码bytes在类型之间转换，但正如本章前面所介绍的那样，我们通常会坚持对文本数据使用str或对二进制数据使用bytes。由于str和bytes操作集有很大程度的重合，所以对于大多数程序来说，做出选择并不是那么难（参见本章最后一个部分针对这种情况的基础示例所介绍的字符串工具）。

除了类型限制，在Python 3.0中，**文件内容**也有关系。文本模式的输入文件需要一个str而不是一个bytes用于内容，因此，在Python 3.0中，没有方法把真正的二进制数据写入一个文本模式文件中。根据编码规则，默认字符集以外的bytes有时候可以嵌入一个常规的字符串中，并且它们总是可以在二进制模式中写入。然而，由于Python 3.0中的文本模式输入文件必须能够针对每个Unicode编码来解码内容，因此，没有办法在文本模式中读取真正的二进制数据：

```
# Can't read truly binary data in text mode

>>> chr(0xFF)                                    # FF is a valid char, FE is not
'ÿ'
```

```
>>> chr(0xFE)
UnicodeEncodeError: 'charmap' codec can't encode character '\xfe' in position 1...

>>> open('temp', 'w').write(b'\xFF\xFE\xFD')      # Can't use arbitrary bytes!
TypeError: can't write bytes to text stream

>>> open('temp', 'w').write('\xFF\xFE\xFD')       # Can write if embeddable in str
3
>>> open('temp', 'wb').write(b'\xFF\xFE\xFD')     # Can also write in binary mode
3

>>> open('temp', 'rb').read()                     # Can always read as binary bytes
b'\xff\xfe\xfd'

>>> open('temp', 'r').read()                      # Can't read text unless decodable!
UnicodeEncodeError: 'charmap' codec can't encode characters in position 2-3: ...
```

最后一个错误源自于一个事实——Python 3.0中的所有文本文件实际都是Unicode文本文件，正如下一节所介绍的那样。

使用Unicode文件

到目前为止，我们已经读取和写入了基本的文本文件和二进制文件，但是，如何处理Unicode文件呢？事实证明，读取和写入存储在文件中的Unicode文本很容易，因为Python 3.0的`open`调用针对文本文件接受一个编码，在数据传输的时候，它自动为我们编码和解码。这允许我们处理用不同编码创建的Unicode文本，而不仅是平台默认编码的Unicode文本，并且以不同的编码存储以供转换。

在Python 3.0中读取和写入Unicode

实际上，我们有两种办法可以把字符串**转换**为不同的编码：用方法调用手动地转换和在文件输入输出上自动地转换。在本节中，我们将使用如下的Unicode字符串来说明这一点：

```
C:\misc> c:\python30\python
>>> S = 'A\xc4B\xe8C'                             # 5-character string, non-ASCII
>>> S
'AÄBèC'
>>> len(S)
5
```

手动编码

我们已经介绍过，总是可以根据目标编码名称把一个字符串转换为raw bytes：

```
# Encode manually with methods

>>> L = S.encode('latin-1')                       # 5 bytes when encoded as latin-1
>>> L
```

```
b'A\xc4B\xe8C'
>>> len(L)
5

>>> U = S.encode('utf-8')              # 7 bytes when encoded as utf-8
>>> U
b'A\xc3\x84B\xc3\xa8C'
>>> len(U)
7
```

文件输出编码

现在，要把我们的字符串以特定编码写入一个文本文件，我们可以直接把想要的编码名称传递给open，尽管我们可以先手动地编码并以二进制格式写入，但没有必要这么做：

```
# Encoding automatically when written

>>> open('latindata', 'w', encoding='latin-1').write(S)          # Write as latin-1
5
>>> open('utf8data', 'w', encoding='utf-8').write(S)             # Write as utf-8
5

>>> open('latindata', 'rb').read()                               # Read raw bytes
b'A\xc4B\xe8C'

>>> open('utf8data', 'rb').read()                                # Different in files
b'A\xc3\x84B\xc3\xa8C'
```

文件输入编码

类似地，要读取任意的Unicode数据，我们直接把文件的编码类型名称传入open，并且，它自动根据raw bytes解码出字符串；我们也可以手动地读取raw byte并解码，但是，当读取数据块的时候（我们可能读取不完整的字符），这可能有些繁琐，并且也没有必要这么做：

```
# Decoding automatically when read

>>> open('latindata', 'r', encoding='latin-1').read()            # Decoded on input
'AÄBèC'
>>> open('utf8data', 'r', encoding='utf-8').read()               # Per encoding type
'AÄBèC'

>>> X = open('latindata', 'rb').read()                           # Manual decoding
>>> X.decode('latin-1')                                         # Not necessary
'AÄBèC'
>>> X = open('utf8data', 'rb').read()
>>> X.decode()                                                  # UTF-8 is default
'AÄBèC'
```

解码错误匹配

最后，别忘了，Python 3.0中的这些文件行为仅限于可以作为文本载入的内容。正如前

面的部分所介绍的，Python 3.0真的必须能够把文本文件中的数据解码为一个str字符串，根据默认的或传入的Unicode编码名称。例如，试图以文本模式打开一个真正的二进制数据文件，即便使用了正确的对象类型，也不可能在Python 3.0中有效。

```
>>> file = open('python.exe', 'r')
>>> text = file.read()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 2: ...

>>> file = open('python.exe', 'rb')
>>> data = file.read()
>>> data[:20]
b'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00\x00'
```

这些例子中的第一个可能不会在Python 2.X中失效（常规文件不能解码文本），即便它可能应该失效：读取一个文件可能会以字符串返回毁坏的数据，由于在文本模式中的自动行末转换（读取的时候，任何嵌入的\r\n字节都将在Windows下转换为\n）。在Python 2.6中，要把文件内容当做Unicode文本对待，我们需要使用特殊的工具而不是通用的内置函数open，稍后我们将介绍这些。那么，首先，让我们来看一个更重要的话题。

在Python 3.0中处理BOM

正如本章前面所介绍的，一些编码方式在文件的开始处存储了一个特殊的**字节顺序标记**（BOM）序列，来指定数据的大小尾方式或声明编码类型。如果编码名暗示了BOM的时候，Python在输入和将其写入输出的时候都会忽略该标记，但是有时候必须使用一个特定的编码名称来迫使显式地处理BOM。

例如，当我们把一个文本文件保存到Windows Notepad中的时候，可以在一个下拉列表中指定其编码类型——简单的ASCII文本、UTF-8或者小尾或大尾的UTF-16。例如，如果一个单行的、名为*spam.txt*的文本文件在Notepad中按照编码类型“ANSI”保存，它会编写为一个简单的ASCII文件而没有一个BOM。当这个文件在Python中以二进制模式读取的时候，我们可以看到存储在文件中的真正的bytes。当它作为文本读取的时候，Python默认会执行行尾转换，既然ASCII是UTF-8的一个子集（并且UTF-8是Python 3.0的默认编码），我们可以将其显式地解码为UTF-8文本：

```
c:\misc> C:\Python30\python                          # File saved in Notepad
>>> import sys
>>> sys.getdefaultencoding()
'utf-8'
>>> open('spam.txt', 'rb').read()                    # ASCII (UTF-8) text file
b'spam\r\nSPAM\r\n'
>>> open('spam.txt', 'r').read()                     # Text mode translates line-end
'spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
```

```
'spam\nSPAM\n'
```

如果文件在Notepad中保存为“UTF-8”，预先使用一个3字节UTF-8 BOM序列，并且我们需要给出更多具体编码名称（“utf-8-sig”）来迫使Python跳过标记：

```
>>> open('spam.txt', 'rb').read()                    # UTF-8 with 3-byte BOM
b'\xef\xbb\xbfspam\r\nSPAM\r\n'
>>> open('spam.txt', 'r').read()
'ï»¿spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'\ufeffspam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8-sig').read()
'spam\nSPAM\n'
```

如果文件作为“Unicode大尾”存储在Notepad中，我们得到了文件中的UTF-16格式的数据，预先使用一个两字节的BOM序列——在Python中，编码名“utf-16”忽略BOM，因为它是暗示的（因为所有的UTF-16文件都有一个BOM），并且“utf-16-be”处理大尾格式但不会忽略BOM：

```
>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00s\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n'
>>> open('spam.txt', 'r').read()
UnicodeEncodeError: 'charmap' codec can't encode character '\xfe' in position 1:...
>>> open('spam.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-16-be').read()
'\ufeffspam\nSPAM\n'
```

对于**输出**通常也是这样。当用Python代码写入一个Unicode文件，我们需要一个更加显式的编码名称来强迫UTF-8中带有BOM——“utf-8”不会写入（或忽略）BOM，但“utf-8-sig”会这么做：

```
>>> open('temp.txt', 'w', encoding='utf-8').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()                    # No BOM
b'spam\r\nSPAM\r\n'

>>> open('temp.txt', 'w', encoding='utf-8-sig').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()                    # Wrote BOM
b'\xef\xbb\xbfspam\r\nSPAM\r\n'

>>> open('temp.txt', 'r').read()
'ï»¿spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()   # Keeps BOM
'\ufeffspam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()  # Skips BOM
'spam\nSPAM\n'
```

注意，尽管“utf-8”没有抛弃BOM，但不带BOM的数据可以用“utf-8”和“utf-8-sig”

读取——如果你不确定一个文件中是否有BOM，使用后者进行输入（在机场安全检测线上，不要大声读出这一段）：

```
>>> open('temp.txt', 'w').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()                          # Data without BOM
b'spam\r\nSPAM\r\n'
>>> open('temp.txt', 'r').read()                           # Any utf-8 works
'spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()
'spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()
'spam\nSPAM\n'
```

最后，对于编码名“utf-16”，BOM自动处理：在**输出**上，数据以平台本地的大小尾方式写入，并且，BOM总是会写的；在**输入**上，数据根据每个BOM解码，并且BOM总是会去除掉。更具体的UTF-16编码名称可以指定不同的大小尾，尽管在某些情况下如果需要或显示BOM的话，我们必须自己手动地编写和略过BOM：

```
>>> sys.byteorder
'little'
>>> open('temp.txt', 'w', encoding='utf-16').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()
b'\xff\xfes\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n\x00'
>>> open('temp.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'

>>> open('temp.txt', 'w', encoding='utf-16-be').write('\ufeffspam\nSPAM\n')
11
>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00s\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n'
>>> open('temp.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-16-be').read()
'\ufeffspam\nSPAM\n'
```

更具体的UTF-16编码名称对于缺乏BOM的文件都工作得很好，尽管“utf-16”在输入时需要一个BOM以便确定字节顺序：

```
>>> open('temp.txt', 'w', encoding='utf-16-le').write('SPAM')
4
>>> open('temp.txt', 'rb').read()                  # OK if BOM not present or expected
b'S\x00P\x00A\x00M\x00'
>>> open('temp.txt', 'r', encoding='utf-16-le').read()
'SPAM'
>>> open('temp.txt', 'r', encoding='utf-16').read()
UnicodeError: UTF-16 stream does not start with BOM
```

自己尝试实验这些编码，或者查看Python的库手册，以了解关于BOM的更多细节。

Python 2.6中的Unicode文件

前面的讨论适用于Python 3.0的字符串类型和文件。我们可以针对Python 2.6中的Unicode实现类似的效果，但是，接口是不同的。如果用unicode替代str并且用codecs.open来打开，在Python 2.6中的结果基本相同：

```
C:\misc> c:\python26\python
>>> S = u'A\xc4B\xe8C'
>>> print S
AÄBèC
>>> len(S)
5
>>> S.encode('latin-1')
'A\xc4B\xe8C'
>>> S.encode('utf-8')
'A\xc3\x84B\xc3\xa8C'

>>> import codecs
>>> codecs.open('latindata', 'w', encoding='latin-1').write(S)
>>> codecs.open('utfdata', 'w', encoding='utf-8').write(S)

>>> open('latindata', 'rb').read()
'A\xc4B\xe8C'
>>> open('utfdata', 'rb').read()
'A\xc3\x84B\xc3\xa8C'

>>> codecs.open('latindata', 'r', encoding='latin-1').read()
u'A\xc4B\xe8C'
>>> codecs.open('utfdata', 'r', encoding='utf-8').read()
u'A\xc4B\xe8C'
```

Python 3.0中其他字符串工具的变化

Python标准库中其他一些常用的字符串处理工具，也由于新的str/bytes类型区分而进行了修改。我们无法在这本介绍核心语言的图书里覆盖所有这些面向应用的工具的细节，但是，为了结束本章的讨论，这里快速看一下受到影响的4种主要的工具：re模式匹配模块、struct二进制数据模块、pickle对象序列化模块和用于解析XML文本的xml包。

re模式匹配模块

Python的re模式匹配模块提供的文本处理，比简单的方法调用所提供的查找、分隔、替换等更加通用。借助re，设定搜索和分隔目标的字符串可以用更通用的模式来描述，而不是用绝对文本描述。这个模块已经泛化为可以用于Python 3.0中的任何字符串类型的对象——str、bytes和 bytearray，并且返回同样类型的结果子字符串作为目标字符串。

如下是其在Python 3.0中的引用，从一行文本提取子字符串。在模式字符串中，(.*)表示任何字符(.)、0或多次(*)、，作为一个匹配的子字符串单独保存(())。在成功匹配

之后，根据包含在圆括号中的模式部分而匹配的字符串部分就可以使用，通过group或groups方法：

```
C:\misc> c:\python30\python
>>> import re
>>> S = 'Bugger all down here on earth!'                    # Line of text
>>> B = b'Bugger all down here on earth!'                   # Usually from a file

>>> re.match('(.*) down (.*) on (.*)', S).groups()          # Match line to pattern
('Bugger all', 'here', 'earth!')                            # Matched substrings

>>> re.match(b'(.*) down (.*) on (.*)', B).groups()         # bytes substrings
(b'Bugger all', b'here', b'earth!')
```

Python 2.6中的结果是类似的，但是，unicode类型用于非ASCII文本，并且str处理8位的和二进制文本：

```
C:\misc> c:\python26\python
>>> import re
>>> S = 'Bugger all down here on earth!'                    # Simple text and binary
>>> U = u'Bugger all down here on earth!'                   # Unicode text

>>> re.match('(.*) down (.*) on (.*)', S).groups()
('Bugger all', 'here', 'earth!')

>>> re.match('(.*) down (.*) on (.*)', U).groups()
(u'Bugger all', u'here', u'earth!')
```

由于bytes和str支持基本相同的操作集，所以这种类型差异大部分很明显。但是，注意，像在其他的API中一样，我们不能在Python 3.0调用的参数中混合str和bytes类型（尽管如果你不想在二进制数据上进行模式匹配，可能不需要关心这一点）：

```
C:\misc> c:\python30\python
>>> import re
>>> S = 'Bugger all down here on earth!'
>>> B = b'Bugger all down here on earth!'

>>> re.match('(.*) down (.*) on (.*)', B).groups()
TypeError: can't use a string pattern on a bytes-like object

>>> re.match(b'(.*) down (.*) on (.*)', S).groups()
TypeError: can't use a bytes pattern on a string-like object

>>> re.match(b'(.*) down (.*) on (.*)', bytearray(B)).groups()
(bytearray(b'Bugger all'), bytearray(b'here'), bytearray(b'earth!'))

>>> re.match('(.*) down (.*) on (.*)', bytearray(B)).groups()
TypeError: can't use a string pattern on a bytes-like object
```

Struct二进制数据模块

Python的struct模块，用来从字符串创建和提取打包的二进制数据，它在Python 3.0中也

像在Python 2.X中一样工作，但是，打包的数据只是作为bytes和bytearray对象显示，而不是str对象（这是有意义的，因为它本来就是要处理二进制数据的，而不是处理任意编码的文本）。

下面是在Python的两个版本中的应用，它根据二进制类型声明把3个对象打包到一个字符串中（它们创建了一个4字节的整数、一个4字节的字符串和一个两字节的整数）：

```
C:\misc> c:\python30\python
>>> from struct import pack
>>> pack('>i4sh', 7, 'spam', 8)                # bytes in 3.0 (8-bit string)
b'\x00\x00\x00\x07spam\x00\x08'

C:\misc> c:\python26\python
>>> from struct import pack
>>> pack('>i4sh', 7, 'spam', 8)                # str in 2.6 (8-bit string)
'\x00\x00\x00\x07spam\x00\x08'
```

由于bytes有一个几乎近似于Python 3.0和Python 2.6中的str的接口，然而，大多数程序员可能不需要关心——这一修改与大多数已有的代码无关，特别是由于读取一个二进制文件会自动创建一个bytes。尽管下面例子中最后的测试在一个类型错误匹配上失效，但大多数脚本将从一个文件读取二进制数据，而不是将其创建为一个字符串：

```
C:\misc> c:\python30\python
>>> import struct
>>> B = struct.pack('>i4sh', 7, 'spam', 8)
>>> B
b'\x00\x00\x00\x07spam\x00\x08'

>>> vals = struct.unpack('>i4sh', B)
>>> vals
(7, b'spam', 8)

>>> vals = struct.unpack('>i4sh', B.decode())
TypeError: 'str' does not have the buffer interface
```

除了bytes的新语法之外，创建和读取二进制文件在Python 3.0和Python 2.X中几乎同样地工作。像这里的代码，是程序员将会注意到bytes对象类型的主要地方：

```
C:\misc> c:\python30\python

# Write values to a packed binary file

>>> F = open('data.bin', 'wb')                 # Open binary output file
>>> import struct
>>> data = struct.pack('>i4sh', 7, 'spam', 8)  # Create packed binary data
>>> data                                       # bytes in 3.0, not str
b'\x00\x00\x00\x07spam\x00\x08'
>>> F.write(data)                              # Write to the file
10
>>> F.close()
```

```
# Read values from a packed binary file

>>> F = open('data.bin', 'rb')                  # Open binary input file
>>> data = F.read()                             # Read bytes
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> values = struct.unpack('>i4sh', data)       # Extract packed binary data
>>> values                                      # Back to Python objects
(7, b'spam', 8)
```

一旦像这样把二进制数据提取到Python对象中，如果必须做的话，可以更深入地探索二进制的世界——可以索引和分片字符串以得到单个bytes的值，可以用位操作符来从整数提取单个的位，等等（参见本书前面的内容，了解这里应用的操作的更多细节）：

```
>>> values                                      # Result of struct.unpack
(7, b'spam', 8)

# Accesssing bits of parsed integers

>>> bin(values[0])                              # Can get to bits in ints
'0b111'
>>> values[0] & 0x01                            # Test first (lowest) bit in int
1
>>> values[0] | 0b1010                          # Bitwise or: turn bits on
15
>>> bin(values[0] | 0b1010)                     # 15 decimal is 1111 binary
'0b1111'
>>> bin(values[0] ^ 0b1010)                     # Bitwise xor: off if both true
'0b1101'
>>> bool(values[0] & 0b100)                     # Test if bit 3 is on
True
>>> bool(values[0] & 0b1000)                    # Test if bit 4 is set
False
```

由于解析的bytes字符串是小整数的序列，所以我们可以对其单个的bytes做类似的处理：

```
# Accessing bytes of parsed strings and bits within them

>>> values[1]
b'spam'
>>> values[1][0]                                # bytes string: sequence of ints
115
>>> values[1][1:]                               # Prints as ASCII characters
b'pam'
>>> bin(values[1][0])                           # Can get to bits of bytes in strings
'0b1110011'
>>> bin(values[1][0] | 0b1100)                  # Turn bits on
'0b1111111'
>>> values[1][0] | 0b1100
127
```

当然，大多数Python程序员不会处理二进制位；Python拥有更高级别的对象类型，例如

列表和字典，对于在Python脚本中表示信息，它们通常是一种更好的选择。然而，如果你必须使用或产生供C程序、网络库或其他接口所使用的低层级数据，Python也有辅助的工具。

pickle对象序列化模块

我们在本书第9章和第30章简单介绍了pickle模块。在第27章，我们也使用了shelve模块，它实际上使用了pickle。这里为了完整起见，别忘了pickle模块的Python 3.0版本总是创建一个bytes对象，而不管默认的或传入的“协议”（数据格式化层级）。我们通过使用该模块的dumps调用来返回一个对象的pickle字符串，从而查看这一点：

```
C:\misc> C:\Python30\python
>>> import pickle                              # dumps() returns pickle string

>>> pickle.dumps([1, 2, 3])                   # Python 3.0 default protocol=3=binary
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'

>>> pickle.dumps([1, 2, 3], protocol=0)       # ASCII protocol 0, but still bytes!
b'(lp0\nL1L\naL2L\naL3L\na.'
```

这意味着，用来存储pickle化的对象的文件必须总是在Python 3.0中以二进制模式打开，因为文本文件使用str字符串来表示数据，而不是bytes——dump调用直接试图把pickle字符串写入一个打开的输出文件中:

```
>>> pickle.dump([1, 2, 3], open('temp', 'w'))    # Text files fail on bytes!
TypeError: can't write bytes to text stream      # Despite protocol value

>>> pickle.dump([1, 2, 3], open('temp', 'w'), protocol=0)
TypeError: can't write bytes to text stream

>>> pickle.dump([1, 2, 3], open('temp', 'wb'))   # Always use binary in 3.0

>>> open('temp', 'r').read()
UnicodeEncodeError: 'charmap' codec can't encode character '\u20ac' in ...
```

由于pickle数据不是可解码的Unicode文本，所以对于输出也是如此——在Python 3.0中的正确用法总是要求以二进制模式写入和读取pickle数据：

```
>>> pickle.dump([1, 2, 3], open('temp', 'wb'))
>>> pickle.load(open('temp', 'rb'))
[1, 2, 3]
>>> open('temp', 'rb').read()
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

在Python 2.6（和更早的版本）中，我们可以使用文本模式文件来获得pickle化的数据，只要协议是0层级（Python 2.6中默认的协议）并且我们使用文本模式一致地转换行尾：

```
C:\misc> c:\python26\python
>>> import pickle
>>> pickle.dumps([1, 2, 3])                    # Python 2.6 default=0=ASCII
'(lp0\nI1\naI2\naI3\na.'

>>> pickle.dumps([1, 2, 3], protocol=1)
']q\x00(K\x01K\x02K\x03e.'

>>> pickle.dump([1, 2, 3], open('temp', 'w'))    # Text mode works in 2.6
>>> pickle.load(open('temp'))
[1, 2, 3]
>>> open('temp').read()
'(lp0\nI1\naI2\naI3\na.'
```

如果你关注版本独立，或者不想要关心协议或者其特定版本的默认值，总是对pickle化的数据使用二进制模式文件，如下的做法在Python 3.0和Python 2.6中都有效：

```
>>> import pickle
>>> pickle.dump([1, 2, 3], open('temp', 'wb'))   # Version neutral
>>> pickle.load(open('temp', 'rb'))              # And required in 3.0
[1, 2, 3]
```

由于几乎所有的程序都允许Python自动地pickle或unpikle对象，并且不会处理pickle化的数据自身的内容，总是使用二进制文件模式的要求是Python 3的新pickle化模式中唯一显著的不兼容。参阅参考书籍或Python的手册，来了解关于对象pickle化的更多细节。

XML解析工具

XML是一种基于标签的语言，用于定义结构化的信息，通常用来定义通过Web传输的文档和数据。尽管可以使用基本的字符串方法或`re`模式模块从XML文本提取一些信息，但XML的嵌套式的结构和任意的属性文本使得全面解析更为精确。

由于XML是如此广泛使用的格式，Python自身附带一个完整的XML解析工具包，所以它支持SAX和DOM解析模式，此外，还有一个名为*ElementTree*的包——这是特定于Python的专用于解析和构造XML的一个API。除了基本的解析，开源领域还提供了额外的XML工具，例如XPath、Xquery、XSLT，等等。

XML根据定义以Unicode形式表示文本，以支持国际化。尽管大多数Python的XML解析工具总是返回Unicode字符串，但在Python 3.0中，它们的结果必须从Python 2.X的`unicode`类型转变为Python 3.0通用的`str`字符串类型——这是有意义的，因为Python 3.0的str字符串是Unicode，不管编码是ASCII或是其他的。

我们无法在这里介绍更多细节，但是，示例对于了解这方面内容有帮助，假设我们有一个简单的XML文本文件*mybooks.xml*:

```
<books>
    <date>2009</date>
    <title>Learning Python</title>
    <title>Programming Python</title>
    <title>Python Pocket Reference</title>
    <publisher>O'Reilly Media</publisher>
</books>
```

并且我们想要运行一个脚本来提取并显示所有嵌套的title标记的内容，如下所示：

```
Learning Python
Programming Python
Python Pocket Reference
```

至少有4种基本的方法来做到这点（不包括像XPath这样更高级的工具）。首先，我们可以在文件的文本上运行基本的**模式匹配**，尽管如果文本不可预期的话，这种方法可能不精确。采用这种方法的时候，我们前面介绍的re模块可以完成这项工作，它的match方法从字符串开始查找一个匹配，search方法向前查找一个匹配，这里使用的findall方法找出字符串中和模式匹配的所有地方（返回的结果是，和括号括起来的模式组或多个这样的组的元组对应的匹配子字符串的列表）：

```
# File patternparse.py

import re
text = open('mybooks.xml').read()
found = re.findall('<title>(.*)</title>', text)
for title in found: print(title)
```

其次，为了更加稳健，我们可以用标准库的DOM**解析**支持来执行完整的XML解析。DOM把XML文本解析为一个对象树，并且提供一个接口在树中导航并提取标签属性和值；这个接口是一个正式规范，独立于Python：

```
# File domparse.py

from xml.dom.minidom import parse, Node
xmltree = parse('mybooks.xml')
for node1 in xmltree.getElementsByTagName('title'):
    for node2 in node1.childNodes:
         if node2.nodeType == Node.TEXT_NODE:
             print(node2.data)
```

作为第三种方式，Python的标准库支持SAX**解析**XML。在SAX模式下，类的方法接收回调作为一个解析过程，并且使用状态信息来记录它们在文档中的位置并收集其数据：

```
# File saxparse.py

import xml.sax.handler
class BookHandler(xml.sax.handler.ContentHandler):
    def __init__(self):
```

```
            self.inTitle = False
        def startElement(self, name, attributes):
            if name == 'title':
                self.inTitle = True
        def characters(self, data):
            if self.inTitle:
                print(data)
        def endElement(self, name):
            if name == 'title':
                self.inTitle = False

    import xml.sax
    parser = xml.sax.make_parser()
    handler = BookHandler()
    parser.setContentHandler(handler)
    parser.parse('mybooks.xml')
```

最后，可以使用标准库的etree包中的*ElementTree*系统，它往往用来实现和XML DOM解析器相同的效果，但是，代码更少。这是一种特定于Python的方式，既解析XML文本也可以生成XML文本。在解析之后，其API可以访问文档的组件：

```
# File etreeparse.py

from xml.etree.ElementTree import parse
tree = parse('mybooks.xml')
for E in tree.findall('title'):
    print(E.text)
```

在Python 2.6或Python 3.0中运行的时候，所有这4段脚本都显示相同的结果：

```
C:\misc> c:\python26\python domparse.py
Learning Python
Programming Python
Python Pocket Reference

C:\misc> c:\python30\python domparse.py
Learning Python
Programming Python
Python Pocket Reference
```

然而，从技术上，在Python 2.6中，这些脚本中的某些产生unicode字符串对象，而在Python 3.0中都产生str字符串，因为该类型包含了Unicode文本（不管是ASCII或其他）：

```
C:\misc> c:\python30\python
>>> from xml.dom.minidom import parse, Node
>>> xmltree = parse('mybooks.xml')
>>> for node in xmltree.getElementsByTagName('title'):
...     for node2 in node.childNodes:
...         if node2.nodeType == Node.TEXT_NODE:
...             node2.data
...
```

```
'Learning Python'
'Programming Python'
'Python Pocket Reference'

C:\misc> c:\python26\python
>>> ...same code...
...
u'Learning Python'
u'Programming Python'
u'Python Pocket Reference'
```

必须处理XML解析的程序，导致了需要针对Python 3.0中的不同对象类型考虑不一般的方式。再次，尽管所有的字符串在Python 2.6和Python 3.0中都有几乎相同的接口，但大多数脚本不会受到这样修改的影响。可用于Python 2.6的`unicode`工具，通常也可用于Python 3.0中的`str`。

遗憾的是，继续深入讨论XML的细节超越了本书的范围。如果你对于文本或XML解析感兴趣，在后续基于应用程序的图书*Programming Python*中介绍了其更多细节。要了解关于`re`、`struct`、`pickle`和XML工具的更多知识，可在Web上搜索，阅读前面提到的书籍或其他书籍，或参阅Python的标准库手册。

本章小结

本章介绍了Python 3.0和Python 2.6中可以用来处理Unicode文本和二进制数据的高级字符串类型。正如我们所介绍的，很多程序员使用ASCII文本，并且使用基本的字符串类型及其操作就够了。对于一些较高级的应用程序，Python的字符串模型全面支持宽字符Unicode文本（通过Python 3.0中的常规字符串类型和Python 2.6中的一种特殊类型）以及面向字节的数据（在Python 3.0中有一个bytes类型表示，在Python 2.6中用常规字符串表示）。

此外，我们学习了Python3.0中的文件对象如何自动编码和解码Unicode文本以及针对二进制模式文件处理字节字符串。最后，我们简单地介绍了Python库中的一些文本和二进制数据工具，并且示例了它们在Python 3.0中的应用。

下一章中，我们将把关注点转移到工具构建器的话题，看看通过插入有趣的自动运行代码来管理对对象属性访问的方法。在继续学习之前，让我们来进行一组测试，复习在本章中学习过的内容。

本章习题

1. Python 3.0中字符串对象类型的名称和角色是什么？

2. Python 2.6中字符串对象类型的名称和角色是什么？
3. Python 2.6和Python 3.0中的字符串类型是如何对应的？
4. Python 3.0的字符串类型在操作上有何不同？
5. 在Python 3.0中，如何把非ASCII Unicode字符编写到字符串中？
6. Python 3.0中的文本模式文件和二进制模式文件之间的主要区别是什么？
7. 如何读取一个Unicode文本文件，它包含按照一种不同于平台默认编码进行编码的文本。
8. 如何以一种特定的编码格式创建一个Unicode文本文件？
9. 为什么把ASCII文本看做是一种Unicode文本？
10. Python 3.0的字符串类型修改对你的代码有多大的影响？

习题解答

1. Python 3.0有3种字符串类型：`str`（用于Unicode文本，包括ASCII）、`bytes`（用于带有绝对字节值的二进制数据）和`bytearray`（`bytes`的一种可变的形式）。`str`类型通常表示存储在文本文件中的内容，其他的两种形式通常表示存储在二进制文件中的内容。
2. Python 2.6有两种主要的字符串类型：`str`（用于8位文本和二进制数据）以及`unicode`（用于宽字符文本）。`str`类型用于文本和二进制文件内容，`unicode`用于通常比8位更复杂的文本文件内容。Python 2.6（但不包括更早的版本）也有Python 3.0的`bytearray`类型，但它主要是一种向后兼容，而且并没有表现出Python 3.0中所表现出来的鲜明的文本/二进制区别。
3. 从Python 2.6到Python 3.0的字符串类型对应并不是直接的，因为Python 2.6的`str`等同于Python 3.0中的`str`和`bytes`，并且Python 3.0中的str等同于Python 2.6中的`str`和`unicode`。Python 3.0中`bytearray`的可变性是唯一的。
4. Python 3.0的字符串类型共享了几乎所有相同的操作：方法调用、序列操作，甚至像模式匹配这样以相同方式工作的更大工具。另一方面，只有`str`支持字符串格式操作，并且`bytearray`有一组额外的操作来执行原处修改。`Str`和`bytes`类型也分别拥有编码和解码文本的方法。
5. 非ASCII Unicode字符可以以十六进制转移（`\xNN`）和Unicode转义（`\uNNNN`，`\UNNNNNNNN`）编写到一个字符串中。在某些键盘上，一些非ASCII字符——例如，某些Latin-1字符，也可以直接录入。

6. 在Python 3.0中，文本模式文件假设其文件内容是Unicode文本（即便它是ASCII），并且当读取的时候自动解码，写入的时候自动编码。对于二进制模式的文件，bytes和文件之间不经修改地转换。文本模式文件的内容通常在脚本中表示为str对象，并且二进制文件的内容表示为bytes（或bytearray）对象。文本模式文件也针对特定编码类型处理bytearray，并且输入和输出时自动在行末序列与单个\n之间转换，除非显式地关闭这一功能。二进制模式的文件不会执行任何这样的步骤。

7. 要读取和平台默认编码方式不同的方式编码的文件，直接把文件编码名传递给Python 3.0的内置函数open（在Python 2.6中是codecs.open()）；当从文件读取数据的时候，数据将针对每种特定编码来解码。我们也可以在二进制模式中读取，并且通过给定一个编码名来手动地把字节解码成一个字符串，但是，这涉及额外的工作，并且对于多字节字符更容易出错（可能偶尔要读取字节序列的一部分）。

8. 要以特定编码格式创建一个Unicode文本文件，把想要的编码名称传递给Python 3.0中的open（在Python 2.6中是codecs.open()）。当字符串写入文件中的时候，将会按照每个想要的编码来进行编码。也可以手动把一个字符串编码为字节，并在二进制模式下将其写入，但是，这通常需要额外的工作。

9. ASCII文本看作是一种Unicode文本，因为其7位范围值只是大多数Unicode编码的一个子集。例如，有效的ASCII文本也是有效的Latin-1文本（Latin-1只是把一个8位字节中其余可能的值分配给额外的字符），并且是有效的UTF-8文本（UTF-8为表示更多的字符定义了一个变量-字节方案，但是ASCII字符仍然用同样的代码表示，即单个的字节）。

10. Python 3.0的字符串类型修改的影响，取决于你所使用的字符串的类型。对于那些使用简单ASCII文本的脚本，可能根本没有影响：在此情况下，str字符串类型在Python 2.6和Python 3.0中的用法相同。此外，尽管标准库中像re、struct、pickle和xml这样字符串相关的工具可能在Python 3.0和Python 2.6中技术上的用法不同，但这一修改很大程度上与大多数程序不相关，因为Python 3.0的str和bytes以及Python 2.6的str都支持几乎相同的接口。如果你处理Unicode数据，只需要直接把Python 2.6的工具集unicode和codecs.open()转换为Python 3.0的str和open。如果你处理二进制数据文件，将需要处理作为bytes对象的内容。由于它们与Python 2.6字符串具有相似的接口，所以影响再次变得很小。

第37章

管理属性

本章将展开介绍前面所提到的**属性拦截**技术，并且还要介绍另一种技术，将它们应用到一些较大的示例中。就像本书中这一部分的其他各章一样，本章也作为高级话题并供读者选择阅读，因为大多数高级程序员不需要关心这里讨论的内容——他们可以获取和设置对象的属性，而不必关心属性的实现。然而，特别是对工具构建者来说，管理属性的访问可能是灵活的API的一个重要部分。

为什么管理属性

对象的属性是大多数Python程序的中心——它们是我们经常存储供脚本处理的相关实体信息的地方。通常，属性只是对象的名称，例如，一个人的`name`属性，可能是一个简单的字符串，通过基本的属性语法来获取和设置：

```
person.name                          # Fetch attribute value
person.name = value                  # Change attribute value
```

在大多数情况下，属性位于对象自身之中，或者继承自对象所派生自的一个类。基本的模式对于我们编写职业生涯中的大多数Python程序来说已经足够了。

然而，有的时候需要更多的灵活性。假设你编写了一个程序来直接使用一个`name`属性，但是随后需要修改——例如，在以某种方式设置或修改名称的时候，需要进行逻辑验证。编写一个方法来管理对属性值的访问（`valid`和`transform`在这里是抽象的），这是很直接的：

```
class Person:
    def getName(self):
        if not valid():
```

```
        raise TypeError('cannot fetch name')
    else:
        return self.name.transform()
    def setName(self, value):
         if not valid(value):
            raise TypeError('cannot change name')
        else:
            self.name = transform(value)

    person = Person()
    person.getName()
    person.setName('value')
```

然而，这需要在整个程序中用到了名称的所有地方都进行修改，这可能不是个小任务。此外，这种方法需要程序知道如何导出值：是作为简单的名称或是作为调用的方法。如果从一开始就使用基于方法的接口来访问数据，客户端不会受到修改的影响；如果没有这么做，它们可能就会变成问题。

这个问题可能比我们想象的要更常见。例如，像电子表格这样的程序中一个单元格的值，可能作为一个简单的离散值开始其存在，但是，随后要变为一个任意的计算。由于一个对象的接口应该足够灵活以支持未来这样的修改，而不会破坏已有的代码，因此以后再切换到方法就不是理想的做法了。

插入在属性访问时运行的代码

一个更好的解决方案是，如果需要的话，在属性访问时自动运行代码。在本书的各种不同位置，我们都遇到过这样的Python工具，当获取属性值以及在存储属性值对其进行验证或修改的时候，这样的工具允许脚本动态地计算属性值。在本章中，我们将展开介绍那些已经遇到过的工具，探讨其他可用的工具，并且学习此领域的一些较大的使用示例。特别强调，本章包括以下内容：

- `__getattr__`和`__setattr__`方法，把未定义的属性获取和所有的属性赋值指向通用的处理器方法。
- `__getattribute__`方法，把所有属性获取都指向Python 2.6的新式类和Python 3.0的所有类中的一个泛型处理器方法。·
- `property`内置函数，把特定属性访问定位到get和set处理器函数，也叫做特性（Property）。
- 描述符协议，把特定属性访问定位到具有任意get和set处理器方法的类的实例。

这里的第一个和第三个方法在第六部分都简单介绍过了；其他两个是这里要介绍的新主题。

正如我们将要看到的，所有4种技术在某种程度上具有同样的目标，并且通常可能对于给定的问题使用任何一种技术来编写代码。然而它们确实存在某些重要的不同。例如，这里列出的最后两种技术适用于特定属性，而前两种则足够通用，可以用于那些必须把任意属性指向包装的对象的、基于委托的类。我们将会看到，所有4种方法在复杂性和优雅性上也都有所不同，在使用中，我们必须通过实际应用来自行判断。

本章除了学习本节列出的4种属性拦截技术背后的细节，还给出了机会来探究比我们在本书前面任何地方所见过的程序更大的程序。例如，最后的CardHolder案例，可以作为使用一个较大的类的自学示例。在下一章中，我们还将使用这里介绍的某些技术来编写装饰器，因此，在继续学习之前，确保对这里介绍的技术至少有个一般性的理解。

特性

特性协议允许我们把一个特定属性的get和set操作指向我们所提供的函数或方法，使得我们能够插入在属性访问的时候自动运行的代码，拦截属性删除，并且如果愿意的话，还可为属性提供文档。

通过property内置函数来创建特性并将其分配给类属性，就像方法函数一样。同样，可以通过子类和实例继承属性，就像任何其他类属性一样。它们的访问拦截功能通过self实例参数提供，该参数确保了在主体实例上访问状态信息和类属性是可行的。

一个特性管理一个单个的、特定的属性；尽管它不能广泛地捕获所有的属性访问，它允许我们控制访问和赋值操作，并且允许我们自由地把一个属性从简单的数据改变为一个计算，而不会影响已有的代码。正如你将看到的，特性和描述符有很大的关系，它们基本上是描述符的一种受限制的形式。

基础知识

可以通过把一个内置函数的结果赋给一个类属性来创建一个特性：

```
attribute = property(fget, fset, fdel, doc)
```

这个内置函数的参数都不是必需的，并且如果没有传递参数的话，所有都取默认值None。这样的操作是不受支持的，并且尝试使用默认值将会引发一个异常。当使用它们的时候，我们向fget传递一个函数来拦截属性访问，给fset传递一个函数进行赋值，并且给fdel传递一个函数进行属性删除；doc参数接收该属性的一个文档字符串，如果想要的话（否则，该特性会赋值fget的文档字符串，如果提供了fget的文档字符串的话，其默认值为None）。fget返回计算过的属性值，并且fset和fdel不返回什么（确实是None）。

这个内置的函数调用返回一个特性对象，我们将它赋给了在类的作用域中要管理的属性的名称，正是在类的作用域中每个实例都继承了类。

第一个例子

为了说明如何把这些转换成有用的代码，如下的类使用一个特性来记录对一个名为name的属性的访问，实际存储的数据名为_name，以便不会和特性搞混了：

```
class Person:                           # Use (object) in 2.6
    def __init__(self, name):
        self._name = name
    def getName(self):
        print('fetch...')
        return self._name
    def setName(self, value):
        print('change...')
        self._name = value
    def delName(self):
        print('remove...')
        del self._name
    name = property(getName, setName, delName, "name property docs")

bob = Person('Bob Smith')               # bob has a managed attribute
print(bob.name)                         # Runs getName
bob.name = 'Robert Smith'               # Runs setName
print(bob.name)
del bob.name                            # Runs delName

print('-'*20)
sue = Person('Sue Jones')               # sue inherits property too
print(sue.name)
print(Person.name.__doc__)              # Or help(Person.name)
```

Python 2.6和Python 3.0中都可以使用特性，但是，它们要求在Python 2.6中派生一个新式对象，才能使赋值正确地工作——为了在Python 2.6中运行代码，这里把对象添加为一个超类（我们在Python 3.0中也可以使用超类，但是，这是暗含的，并且不是必需的）。

这个特定的特性所做的事情并不多——它只是拦截并跟踪了一个属性，这里将它作为展示协议的一个例子。当这段代码运行的时候，两个实例继承了该特性，就好像它们是附加到其类的另外两个属性一样。然而，捕获了它们的属性访问：

```
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
```

```
name property docs
```

就像所有的类属性一样，实例和较低的子类都**继承**特性。如果我们把例子修改为如下所示：

```
class Super:
    ...the original Person class code...
    name = property(getName, setName, delName, 'name property docs')

class Person(Super):
    pass                                # Properties are inherited

bob = Person('Bob Smith')
...rest unchanged...
```

输出是同样的。Person子类从Super继承了name特性，并且bob实例从Person获取了它。关于继承，特性的方式和常规方法是一样的；由于它们能够访问self实例参数，所以它们可以访问像方法这样的实例状态信息，就像下面小节所介绍的一样。

计算的属性

前面小节的例子简单地跟踪了属性访问。然而，通常特性做的更多——例如，当获取属性的时候，动态地计算属性的值。下面的例子展示了这一点：

```
class PropSquare:
    def __init__(self, start):
        self.value = start
    def getX(self):                             # On attr fetch
        return self.value ** 2
    def setX(self, value):                      # On attr assign
        self.value = value
    X = property(getX, setX)                    # No delete or docs

P = PropSquare(3)         # 2 instances of class with property
Q = PropSquare(32)        # Each has different state information

print(P.X)                # 3 ** 2
P.X = 4
print(P.X)                # 4 ** 2
print(Q.X)                # 32 ** 2
```

这个类定义了一个X属性，并且将其当作静态数据一样访问，但实际运行的代码在获取该属性的时候计算了它的值。这种效果很像是一个隐式方法调用。当代码运行的时候，值作为状态信息存储在实例中，但是，每次我们通过管理的属性获取它的时候，它的值都会自动平方：

```
9
16
```

```
1024
```

注意，我们已经创建了两个不同的实例——因为特性方法自动地接收一个self参数，所以它们都访问了存储在实例中的状态信息。在我们的例子中，这意味着获取会计算主体实例的数据的平方。

使用装饰器编写特性

尽管我们直到下一章才会介绍额外细节，但我们更早地在第31章就引入了函数装饰器的基本概念。回忆一下，函数装饰器的语法是：

```
@decorator
def func(args): ...
```

Python会自动将其翻译成对等形式，把函数名重新绑定到可调用的decorator的返回结果上：

```
def func(args): ...
func = decorator(func)
```

由于这一映射，证实了内置函数property可以充当一个装饰器，来定义一个函数，当获取一个属性的时候自动运行该函数：

```
class Person:
    @property
    def name(self): ...                  # Rebinds: name = property(name)
```

运行的时候，装饰的方法自动传递给property内置函数的第一个参数。这其实只是创建一个特性并手动绑定属性名的一种替代语法：

```
class Person:
    def name(self): ...
    name = property(name)
```

对于Python 2.6，property对象也有getter、setter和deleter方法，这些方法指定相应的特性访问器方法赋值并且返回特性自身的一个副本。我们也可以使用这些方法，通过装饰常规方法来指定特性的组成部分，尽管getter部分通常由创建特性自身的行为自动填充：

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

    @property
    def name(self):              # name = property(name)
        "name property docs"
         print('fetch...')
```

```
            return self._name

        @name.setter
        def name(self, value):              # name = name.setter(name)
            print('change...')
            self._name = value

        @name.deleter
        def name(self):                     # name = name.deleter(name)
            print('remove...')
            del self._name

    bob = Person('Bob Smith')               # bob has a managed attribute
    print(bob.name)                         # Runs name getter (name 1)
    bob.name = 'Robert Smith'               # Runs name setter (name 2)
    print(bob.name)
    del bob.name                            # Runs name deleter (name 3)

    print('-'*20)
    sue = Person('Sue Jones')               # sue inherits property too
    print(sue.name)
    print(Person.name.__doc__)              # Or help(Person.name)
```

实际上，这段代码等同于本小节的第一个示例——在这个例子中，装饰只是编写特性的一种替代方法。当运行这段代码时，结果是相同的：

```
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
name property docs
```

和`property`手动赋值的结果相比，这个例子中，使用装饰器来编写特性只需要3行额外的代码（这是无法忽视的差别）。就像替代工具的通常情况一样，在这两种技术之间的选择很大程度上与个人爱好有关。

描述符

描述符提供了拦截属性访问的一种替代方法；它们与前面小节所讨论的特性有很大的关系。实际上，特性是描述符的一种——从技术上讲，`property`内置函数只是创建一个特定类型的描述符的一种简化方式，而这种描述符在属性访问时运行方法函数。

从功能上讲，描述符协议允许我们把一个特定属性的get和set操作指向我们提供的一个单

独类对象的方法：它们提供了一种方式来插入在访问属性的时候自动运行的代码，并且它们允许我们拦截属性删除并且为属性提供文档（如果愿意的话）。

描述符作为独立的**类**创建，并且它们就像方法函数一样分配给类属性。和任何其他的类属性一样，它们可以通过子类和实例继承。通过为描述符自身提供一个`self`，以及提供客户类的实例，都可以提供访问拦截方法。因此，它们可以自己保留和使用状态信息，以及主体实例的状态信息。例如，一个描述符可能调用客户类上可用的方法，以及它所定义的特定于描述符的方法。

和特性一样，描述符也管理一个单个的、特定的属性。尽管它不能广泛地捕获所有的属性访问，但它提供了对获取和赋值访问的控制，并且允许我们自由地把简单的数据修改为计算值从而改变一个属性，而不会影响已有的代码。特性实际上只是创建一种特定描述符的方便方法，并且，正如我们所见到的，它们可以直接作为描述符编写。

然而，特性的应用领域相对狭窄，描述符提供了一种更为通用的解决方案。例如，由于它们编码为常规类，所以描述符拥有自己的状态，可能参与描述符继承层级，可以使用复合来聚合对象，并且为编写内部方法和属性文档字符串提供一种自然的结构。

基础知识

正如前面所提到的，描述符作为单独的类编写，并且针对想要拦截的属性访问操作提供特定命名的访问器方法——当以相应的方式访问分配给描述符类实例的属性时，描述符类中的获取、设置和删除等方法自动运行：

```
class Descriptor:
    "docstring goes here"
    def __get__(self, instance, owner): ...      # Return attr value
    def __set__(self, instance, value): ...      # Return nothing (None)
    def __delete__(self, instance): ...          # Return nothing (None)
```

带有任何这些方法的类都可以看作是描述符，并且当它们的一个实例分配给另一个类的属性的时候，它们的这些方法是特殊的——当访问属性的时候，会自动调用它们。如果这些方法中的任何一个空缺，通常意味着不支持相应类型的访问。然而，和特性不同，省略一个`__set__`意味着允许这个名字在一个实例中重新定义，因此，隐藏了描述符——要使得一个属性是**只读的**，我们必须定义`__set__`来捕获赋值并引发一个异常。

描述符方法参数

在进行任何真正的编程之前，先来回顾一些基础知识。前面小节介绍的所有3种描述符方法，都传递了描述符类实例（`self`）以及描述符实例所附加的客户类的实例（`instance`）。

__get__访问方法还额外地接收一个owner参数，指定了描述符实例要附加到的类。其instance参数要么是访问的属性所属的实例（用于*instance*.attr），要么当所访问的属性直接属于类的时候是None（用于*class*.attr）。前者通常针对实例访问计算一个值；如果描述符对象访问是受支持的，后者通常返回self。

例如，在下面的例子中，当获取X.attr的时候，Python自动运行Descriptor类的__get__方法，Subject.attr类属性分配给该方法（和特性一样，在Python 2.6中，要在这里使用描述符，我们必须派生自对象；在Python 3.0中，这是隐式的，但无伤大雅）：

```
>>> class Descriptor(object):
...     def __get__(self, instance, owner):
...         print(self, instance, owner, sep='\n')
...
>>> class Subject:
...     attr = Descriptor()              # Descriptor instance is class attr
...
>>> X = Subject()

>>> X.attr
<__main__.Descriptor object at 0x0281E690>
<__main__.Subject object at 0x028289B0>
<class '__main__.Subject'>

>>> Subject.attr
<__main__.Descriptor object at 0x0281E690>
None
<class '__main__.Subject'>
```

注意在第一个属性获取中自动传递到__get__方法中的参数，当获取X.attr的时候，就好像发生了如下的转换（尽管这里的Subject.attr没有再次调用__get__）：

```
X.attr -> Descriptor.__get__(Subject.attr, X, Subject)
```

当描述符的实例参数为None的时候，该描述符知道将直接访问它。

只读描述符

正如前面提到的，和特性不同，使用描述符直接忽略__set__方法不足以让属性成为只读的，因为描述符名称可以赋给一个实例。在下面的例子中，对X.a的属性赋值在实例对象X中存储了a，由此，隐藏了存储在类C中的描述符：

```
>>> class D:
...     def __get__(*args): print('get')
...
>>> class C:
...     a = D()
...
```

```
>>> X = C()
>>> X.a                                  # Runs inherited descriptor __get__
get
>>> C.a
get
>>> X.a = 99                             # Stored on X, hiding C.a
>>> X.a
99
>>> list(X.__dict__.keys())
['a']
>>> Y = C()
>>> Y.a                                  # Y still inherits descriptor
get
>>> C.a
get
```

这就是Python中所有实例属性赋值工作的方式，并且它允许在它们的实例中类选择性地覆盖类级默认值。要让基于描述符的属性成为只读的，捕获描述符类中的赋值并引发一个异常来阻止属性赋值——当要赋值的属性是一个描述符的时候，Python有效地绕过了常规实例层级的赋值行为，并且把操作指向描述符对象：

```
>>> class D:
...     def __get__(*args): print('get')
...     def __set__(*args): raise AttributeError('cannot set')
...
>>> class C:
...     a = D()
...
>>> X = C()
>>> X.a                                  # Routed to C.a.__get__
get
>>> X.a = 99                             # Routed to C.a.__set__
AttributeError: cannot set
```

注意：还要注意不要把描述符`__delete__`方法和通用的`__del__`方法搞混淆了。调用前者是试图删除所有者类的一个实例上的管理属性名称；后者是一种通用的实例析构器方法，当任何类的一个实例将要进行垃圾回收的时候调用。`__delete__`与我们将要在本章后面遇到的`__delattr__`泛型属性删除方法关系更近。参见本书第29章了解关于操作符重载的更多内容。

第一个示例

要看这些如何组合到更为实际的代码中，让我们从前面为特性编写的第一个例子开始。如下代码定义了一个描述符，来拦截对其客户类中的名为`name`的一个属性的访问。其方法使用它们的`instance`参数来访问主体实例中的状态信息，其中指定了实际存储的名称字符串。和特性一样，描述符只对新式类能够很好地工作，因此，如果使用Python 2.6

的话，要确保下面例子中的类都派生自对象object：

```
class Name:                                    # Use (object) in 2.6
    "name descriptor docs"
    def __get__(self, instance, owner):
        print('fetch...')
        return instance._name
    def __set__(self, instance, value):
        print('change...')
        instance._name = value
    def __delete__(self, instance):
        print('remove...')
        del instance._name

class Person:                                  # Use (object) in 2.6
    def __init__(self, name):
        self._name = name
    name = Name()                              # Assign descriptor to attr

bob = Person('Bob Smith')                      # bob has a managed attribute
print(bob.name)                                # Runs Name.__get__
bob.name = 'Robert Smith'                      # Runs Name.__set__
print(bob.name)
del bob.name                                   # Runs Name.__delete__

print('-'*20)
sue = Person('Sue Jones')                      # sue inherits descriptor too
print(sue.name)
print(Name.__doc__)                            # Or help(Name)
```

注意，在这段代码中，我们如何把描述符类的一个实例分配给客户类中的一个**类属性**；正因为如此，它为该类的所有实例所继承，就像是一个类方法一样。实际上，我们**必须**像这样把描述符赋给一个类属性——如果赋给一个self实例属性，它将无法工作。当描述符的__get__方法运行的时候，它传递了3个对象来定义其上下文：

- self是Name类实例
- instance是Person类实例
- owner是Person类实例

当这段代码运行描述符的方法来拦截对该属性的访问的时候，和特性的版本十分相似。实际上，输出再一次相同：

```
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
```

```
Sue Jones
name descriptor docs
```

还和特性示例中相似，我们的描述符类实例是一个类属性，并且因此由客户类和任何子类的所有实例所**继承**。例如，如果我们把示例中的Person类修改为如下的样子，脚本的输出是相同的：

```
...
class Super:
    def __init__(self, name):
        self._name = name
    name = Name()

class Person(Super):                    # Descriptors are inherited
   pass
...
```

还要注意到，当一个描述符类在客户类之外无用的话，将描述符的定义嵌入客户类之中，这在语法上是完全合理的。这里，我们的示例看上去就像使用一个**嵌套的类**：

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

    class Name:                         # Using a nested class
        "name descriptor docs"
        def __get__(self, instance, owner):
            print('fetch...')
            return instance._name
        def __set__(self, instance, value):
            print('change...')
            instance._name = value
        def __delete__(self, instance):
            print('remove...')
            del instance._name
   name = Name()
```

当按照这种方式编码时，Name变成了Person类声明的作用域中的一个局部变量，这样，它不会与类之外的任何名称冲突。这个版本和最初的版本一样地工作——我们已经直接把描述符类定义移动到了客户类的作用域中，但是，测试代码的最后一行必须改为从其新位置获取文档字符串：

```
...
print(Person.Name.__doc__)                  # Differs: not Name.__doc__ outside class
```

计算的属性

和使用特性的例子一样，上一小节中我们的第一个描述符例子也没有做太多事情——它直接打印了属性访问的追踪消息。实际上，描述符也可以用来在每次获取属性的时候计

算它们的值。如下例子说明了这点——它是我们为特性编写的同一个例子的改写，这里使用了一个描述符，从而在每次获取属性值的时候对其值自动求平方：

```
class DescSquare:
    def __init__(self, start):                 # Each desc has own state
        self.value = start
    def __get__(self, instance, owner):         # On attr fetch
        return self.value ** 2
    def __set__(self, instance, value):         # On attr assign
        self.value = value                      # No delete or docs

class Client1:
    X = DescSquare(3)                           # Assign descriptor instance to class attr

class Client2:
    X = DescSquare(32)                          # Another instance in another client class

                                                # Could also code 2 instances in same class
c1 = Client1()
c2 = Client2()

print(c1.X)                                     # 3 ** 2
c1.X = 4
print(c1.X)                                     # 4 ** 2
print(c2.X)                                     # 32 ** 2
```

运行这个例子的时候，其输出与最初的基于特性的版本相同，但是在这里，一个描述符类对象正在拦截属性访问：

```
9
16
1024
```

在描述符中使用状态信息

如果已经学习了我们到目前为止编写的两个描述符的例子，你可能会注意到，它们从不同的地方获取其信息——第一个例子（name属性的示例）使用存储在客户**实例**中的数据，第二个例子（属性乘方的例子）使用附加到**描述符**对象本身的数据。实际上，描述符可以使用实例状态和描述符状态，或者二者的任何组合：

- 描述符状态用来管理内部用于描述符工作的数据。
- 实例状态记录了和客户类相关的信息，以及可能由客户类创建的信息。

描述符方法也可以使用，但是描述符状态常常使得不必要使用特定的命名惯例，以避免存储在一个实例上的描述符数据的名称冲突。例如，如下的描述符把信息附加到自己的实例，因此，它不会与客户类的实例上的信息冲突：

```
class DescState:                                # Use descriptor state
    def __init__(self, value):
        self.value = value
    def __get__(self, instance, owner):         # On attr fetch
        print('DescState get')
        return self.value * 10
    def __set__(self, instance, value):         # On attr assign
        print('DescState set')
        self.value = value

# Client class

class CalcAttrs:
    X = DescState(2)                            # Descriptor class attr
    Y = 3                                       # Class attr
    def __init__(self):
        self.Z = 4                              # Instance attr

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)                      # X is computed, others are not
obj.X = 5                                       # X assignment is intercepted
obj.Y = 6
obj.Z = 7
print(obj.X, obj.Y, obj.Z)
```

这段代码的Value信息仅存在于**描述符**之中，因此，如果在客户类的实例中使用相同的名字，也不会有冲突。注意，这里只是管理了描述符的属性——对X的获取和设置访问被拦截，但是对Y和Z的访问没有拦截（Y附加到客户类，Z附加到其实例）。当这段代码运行的时候，X在获取的时候会计算：

```
DescState get
20 3 4
DescState set
DescState get
50 6 7
```

对描述符存储或使用附加到客户类的实例的一个属性，而不是自己的属性，这也是可行的。如下例子中的描述符假设实例有一个属性_Y通过客户类附加，并且使用它来计算它所表示的属性的值：

```
class InstState:                                # Using instance state
    def __get__(self, instance, owner):
        print('InstState get')                  # Assume set by client class
        return instance._Y * 100
    def __set__(self, instance, value):
        print('InstState set')
        instance._Y = value

# Client class

class CalcAttrs:
    X = DescState(2)                            # Descriptor class attr
```

```
    Y = InstState()                          # Descriptor class attr
    def __init__(self):
        self._Y = 3                          # Instance attr
        self.Z =  4                          # Instance attr

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z)                   # X and Y are computed, Z is not
obj.X = 5                                    # X and Y assignments intercepted
obj.Y = 6
obj.Z = 7
print(obj.X, obj.Y, obj.Z)
```

这一次，X和Y都赋给了描述符，并且获取的时候会计算（X是赋给了前面的例子的描述符）。这里新的描述符本身没有信息，但是它使用了一个假设存在于实例中的属性——这个属性名为_Y，以避免与描述符自身的名称冲突。当这个版本的代码运行的时候，结果是类似的，但是，管理的是第二个属性，使用位于实例中的状态而不是描述符：

```
DescState get
InstState get
20 300 4
DescState set
InstState set
DescState get
InstState get
50 600 7
```

描述符和实例状态都有各自的用途。实际上，这是描述符优于特性的一个通用优点——因为它们都有自己的状态，所以可以很容易地在内部保存数据，而不用将数据添加到客户实例对象的命名空间中。

特性和描述符是如何相关的

正如前面提到的，特性和描述符有很强的相关性——property内置函数只是创建描述符的一种方便方式。既然已经知道了二者是如何工作的，我们应该能够看到，可以使用如下的一个描述符类来模拟property内置函数：

```
class Property:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel                                  # Save unbound methods
        self.__doc__ = doc                                # or other callables

    def __get__(self, instance, instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("can't get attribute")
        return self.fget(instance)                         # Pass instance to self
                                                           # in property accessors
```

```
        def __set__(self, instance, value):
            if self.fset is None:
               raise AttributeError("can't set attribute")
            self.fset(instance, value)

        def __delete__(self, instance):
            if self.fdel is None:
               raise AttributeError("can't delete attribute")
            self.fdel(instance)

    class Person:
    def getName(self): ...
    def setName(self, value): ...
    name = Property(getName, setName)                # Use like property()
```

这个Property类捕获了带有描述符协议的属性访问，并且把请求定位到创建类的时候在描述符状态中传入和保存的函数或方法。例如，属性获取从Person类指向Property类的__get__方法，再回到Person类的getName。有了描述符，这“恰好可以工作”。

注意，尽管这个描述符类等同于只是处理基本的特性用法，使用@ *decorator*语法也只是指定了设置和删除操作，但是我们的Property类也必须用setter和deleter方法扩展，这可能会节省装饰的访问器函数并且返回特性对象（self应该足够了）。既然property内置函数已经做了这些，这里，我们将省略这一扩展的正式编码。

还要注意，描述符用来实现Python的__slots__，使用存储在类级别的描述符来截取slot名称，从而避免了实例属性字典。参见第31章了解关于slot的更多介绍。

注意：在第38章中，我们还将使用描述符来实现应用于函数和方法的函数装饰器。正如你将在那里见到的，由于描述符接收描述符和主体类实例，它们在这种情况下工作得很好，尽管嵌套函数通常是一种更简单的解决方案。

__getattr__和__getattribute__

到目前为止，我们已经学习了特性和描述符——管理特定属性的工具。__getattr__和__getattribute__操作符重载方法提供了拦截类实例的属性获取的另一种方法。就像特性和描述符一样，它们也允许我们插入当访问属性的时候自动运行的代码。然而，我们将会看到，这两个方法有更广泛的应用。

属性获取拦截表现为两种形式，可用两个不同的方法来编写：

- __getattr__针对**未定义的**属性运行——也就是说，属性没有存储在实例上，或者没有从其类之一继承。

- `__getattribute__`针对**每个**属性，因此，当使用它的时候，必须小心避免通过把属性访问传递给超类而导致递归循环。

我们在本书第29章中遇到过前一种情况，它在Python的所有版本中都可用。后者对于Python 2.6中的新式类可用，并且对于Python 3.0中的所有类（隐式都是新式类）可用。这两个方法是一组属性拦截方法的代表，这些方法还包括`__setattr__`和`__delattr__`。由于这些方法具有相同的作用，我们在这里将它们通常作为一个单独话题。

与特性和描述符不同，这些方法是Python的**操作符重载**协议的一部分——是类的特殊命名的方法，由子类继承，并且当在隐式的内置操作中使用实例的时候自动调用。和一个类的所有方法一样，它们每一个在调用的时候都接收第一个`self`参数，访问任何请求的实例状态信息或该类的其他方法。

`__getattr__`和`__getattribute__`方法也比特性和描述符更加**通用**——它们可以用来拦截对任何（几乎所有的）实例属性的获取，而不仅仅只是分配给它们的那些特定名称。因此，这两个方法很适合于通用的基于**委托**的编码模式——它们可以用来实现包装对象，该对象管理对一个嵌套对象的所有属性访问。相反，我们必须为想要拦截的每个属性都定义一个特性或描述符。

最后，这两种方法比我们前面考虑的替代方法的应用领域更**专注集中**一些：它们只是拦截属性获取，而不拦截属性赋值。要捕获赋值对属性的更改，我们必须编写一个`__setattr__`方法——这是一个操作符重载方法，只对每个属性获取运行，必须小心避免由于通过实例命名空间字典指向属性赋值而导致的递归循环。

尽管很少用到，我们还是可以编写一个`__delattr__`重载方法（必须以同样的方式避免循环）来拦截属性删除。相反，特性和描述符通过设计捕获访问、设置和删除操作。

大多数这些操作符重载方法在本书前面都介绍过，这里，我们将展开讨论其用法并研究它们在更大的应用环境中的作用。

基础知识

`__getattr__`和`__setattr__`在本书第29章和第31章介绍，并且第31章简单地提到了`__getattribute__`。简而言之，如果一个类定义了或继承了如下方法，那么当一个实例用于后面的注释所提到的情况时，它们将自动运行：

```
def __getattr__(self, name):           # On undefined attribute fetch [obj.name]
def __getattribute__(self, name):      # On all attribute fetch [obj.name]
def __setattr__(self, name, value):    # On all attribute assignment [obj.name=value]
def __delattr__(self, name):           # On all attribute deletion [del obj.name]
```

所有这些之中，self通常是主体实例对象，name是将要访问的属性的字符串名，value是要赋给该属性的对象。两个get方法通常返回一个属性的值，另两个方法不返回什么（None）。例如，要捕获每个属性获取，我们可以使用上面的前两个方法；要捕获属性赋值，可以使用第三个方法：

```
class Catcher:
    def __getattr__(self, name):
        print('Get:', name)
    def __setattr__(self, name, value):
        print('Set:', name, value)

X = Catcher()
X.job                                   # Prints "Get: job"
X.pay                                   # Prints "Get: pay"
X.pay = 99                              # Prints "Set: pay 99"
```

这样的代码结构可以用来实现我们在第30章介绍的**委托**设计模式。由于所有的属性通常都指向我们的拦截方法，所以我们可以验证它们并将其传递到嵌入的、管理的对象中。例如，下面的类（取自第30章）跟踪了对传递给包装类的另一个对象的**每一次**属性获取：

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object                    # Save object
    def __getattr__(self, attrname):
        print('Trace:', attrname)                # Trace fetch
        return getattr(self.wrapped, attrname)   # Delegate fetch
```

特性和描述符没有这样的类似功能，做不到对每个可能的包装对象中每个可能的属性编写访问器。

避免属性拦截方法中的循环

这些方法通常都容易使用，它们唯一复杂的部分就是潜在的**循环**（即递归）。由于__getattr__仅针对未定义的属性调用，所以它可以在自己的代码中自由地获取其他属性。然而，由于__getattribute__和__setattr__针对所有的属性运行，因此，它们的代码要注意在访问其他属性的时候避免再次调用自己并触发一次递归循环。

例如，在一个__getattribute__方法代码内部的另一次属性获取，将会再次触发__getattribute__，并且代码将会循环直到内存耗尽：

```
def __getattribute__(self, name):
    x = self.other                              # LOOPS!
```

要解决这个问题，把获取指向一个更高的超类，而不是跳过这个层级的版本——object类总是一个超类，并且它在这里可以很好地起作用：

```
def __getattribute__(self, name):
    x = object.__getattribute__(self, 'other')          # Force higher to avoid me
```

对于__setattr__，情况是类似的。在这个方法内赋值任何属性，都会再次触发__setattr__并创建一个类似的循环：

```
def __setattr__(self, name, value):
    self.other = value                                  # LOOPS!
```

要解决这个问题，把属性作为实例的__dict__命名空间字典中的一个键赋值。这样就避免了直接的属性赋值：

```
def __setattr__(self, name, value):
    self.__dict__['other'] = value                      # Use atttr dict to avoid me
```

尽管这种方法比较少用到，但__setattr__也可以把自己的属性赋值传递给一个更高的超类而避免循环，就像__getattribute__一样：

```
def __setattr__(self, name, value):
    object.__setattr__(self, 'other', value)            # Force higher to avoid me
```

相反，我们**不能**使用__dict__技巧在__getattribute__中避免循环：

```
def __getattribute__(self, name):
    x = self.__dict__['other']                          # LOOPS!
```

获取__dict__属性本身会再次触发__getattribute__，导致一个递归循环。很奇怪，但确实如此。

__delattr__方法实际中很少用到，但是，当用到的时候，它针对每次属性删除而调用（就像针对每次属性赋值调用__setattr__一样）。因此，我们必须小心，在删除属性的时候要避免循环，通过使用同样的技术：命名空间字典或者超类方法调用。

第一个示例

所有这些并不像我们前一小节所暗示的那样复杂。要看看如何把这些思想付诸应用，这里是与用来说明特性和描述符的示例一样的示例，不过这次是用属性操作符重载方法实现的。由于这些方法如此通用，所以我们在这里测试属性名来获知何时将要访问一个管理的属性；其他的则允许正常传递：

```
class Person:
    def __init__(self, name):                   # On [Person()]
        self._name = name                       # Triggers __setattr__!

    def __getattr__(self, attr):                # On [obj.undefined]
        if attr == 'name':                      # Intercept name: not stored
```

```
            print('fetch...')
            return self._name                    # Does not loop: real attr
        else:                                    # Others are errors
            raise AttributeError(attr)

    def __setattr__(self, attr, value):          # On [obj.any = value]
        if attr == 'name':
           print('change...')
           attr = '_name'                        # Set internal name
        self.__dict__[attr] = value              # Avoid looping here

    def __delattr__(self, attr):                 # On [del obj.any]
        if attr == 'name':
           print('remove...')
           attr = '_name'                        # Avoid looping here too
        del self.__dict__[attr]                  # but much less common

bob = Person('Bob Smith')                        # bob has a managed attribute
print(bob.name)                                  # Runs __getattr__
bob.name = 'Robert Smith'                        # Runs __setattr__
print(bob.name)
del bob.name                                     # Runs __delattr__

print('-'*20)
sue = Person('Sue Jones')                        # sue inherits property too
print(sue.name)
#print(Person.name.__doc__)                      # No equivalent here
```

注意，`__init__`构造函数中的属性赋值也触发了`__setattr__`，这个方法捕获了每次属性赋值，即便是类自身之中的那些。运行这段代码的时候，会产生同样的输出，但这一次，它是Python的常规操作符重载机制与我们的属性拦截方法的结果：

```
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
```

还要注意，与特性和描述符不同，这里没有为属性直接声明指定的**文档**，管理的属性存在于我们拦截方法的代码之中，而不是在不同的对象中。

要实现与`__getattribute__`相同的结果，用下面的代码替换示例中的`__getattr__`，由于它会捕获所有的属性获取，这个版本必须通过把新的获取传递到超类来避免循环，并且通常不能假设未知的名称是错误：

```
# Replace __getattr__ with this

    def __getattribute__(self, attr):                    # On [obj.any]
```

```
        if attr == 'name':                          # Intercept all names
            print('fetch...')
            attr = '_name'                          # Map to internal name
        return object.__getattribute__(self, attr)  # Avoid looping here
```

这个例子与我们为属性和描述符编写的代码相同，但是它有点人为的痕迹，并且它没有真正地强调这些工具的应用。由于它们是通用的，所以__getattr__和__getattribute__可能在基于委托的代码中更为常用（正如前面所介绍的），在那里属性访问验证并指向一个嵌入的对象。在只有**单个的**属性要管理的情况下，特性和描述符可能会做得更好。

计算属性

正如前面所述，我们前面的示例除了跟踪属性获取没有做任何事情。当获取属性并计算属性值时，它没有做太多额外工作。与介绍特性和描述符时的情况相同，下面的代码创建了一个虚拟的属性X，当获取的时候自动计算它：

```
class AttrSquare:
    def __init__(self, start):
        self.value = start                      # Triggers __setattr__!

    def __getattr__(self, attr):                # On undefined attr fetch
        if attr == 'X':
            return self.value ** 2              # value is not undefined
        else:
            raise AttributeError(attr)

     def __setattr__(self, attr, value):        # On all attr assignments
         if attr == 'X':
             attr = 'value'
         self.__dict__[attr] = value

A = AttrSquare(3)         # 2 instances of class with overloading
B = AttrSquare(32)        # Each has different state information

print(A.X)                # 3 ** 2
A.X = 4
print(A.X)                # 4 ** 2
print(B.X)                # 32 ** 2
```

运行这段代码，会产生与前面我们使用特性和描述符的时候相同的输出，但是，这段脚本的机制是基于通用的属性拦截方法：

```
9
16
1024
```

如前所述，我们可以用__getattribute__而不是__getattr__实现同样的效果。下面的代码用一个__getattribute__替换了获取方法，并且通过使用直接超类方法调用而不是

__dict__键来修改__setattr__赋值方法从而避免循环：

```
class AttrSquare:
    def __init__(self, start):
        self.value = start                              # Triggers __setattr__!

    def __getattribute__(self, attr):                   # On all attr fetches
        if attr == 'X':
            return self.value ** 2                      # Triggers __getattribute__ again!
        else:
            return object.__getattribute__(self, attr)

    def __setattr__(self, attr, value):                 # On all attr assignments
        if attr == 'X':
            attr = 'value'
        object.__setattr__(self, attr, value)
```

当这个版本运行的时候，结果再次相同。注意，隐式的指向在类的方法内部进行：

- 构造函数中的self.value=start 触发__setattr__。
- __getattribute__中self.value再次触发__getattribute__。

实际上，每次我们获取属性X的时候，__getattribute__都运行了**两次**。这并没有在__getattr__版本中发生，因为value属性没有定义。如果你关心速度并且要避免这一点，修改__getattribute__以使用超类来获取value：

```
def __getattribute__(self, attr):
    if attr == 'X':
        return object.__getattribute__(self, 'value') ** 2
```

当然，这仍然会引发对超类方法的一次调用，但是，在获取值之前没有额外的递归调用。给这些方法添加print调用，以记录它们如何运行和何时运行。

__getattr__和__getattribute__比较

为了概括__getattr__和__getattribute__之间的编码区别，下面的例子使用了这两者来实现3个属性——attr1是一个类属性，attr2是一个实例属性，attr3是一个虚拟的管理属性，当获取时计算它：

```
class GetAttr:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattr__(self, attr):                # On undefined attrs only
        print('get: ' + attr)                   # Not attr1: inherited from class
        return 3                                # Not attr2: stored on instance

X = GetAttr()
```

```
print(X.attr1)
print(X.attr2)
print(X.attr3)

print('-'*40)

class GetAttribute(object):                         # (object) needed in 2.6 only
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattribute__(self, attr):               # On all attr fetches
        print('get: ' + attr)                       # Use superclass to avoid looping here
        if attr == 'attr3':
            return 3
        else:
            return object.__getattribute__(self, attr)

X = GetAttribute()
print(X.attr1)
print(X.attr2)
print(X.attr3)
```

运行时，__getattr__版本拦截对attr3的访问，因为它是未定义的。另一方面，__getattribute__版本拦截所有的属性获取，并且必须将那些没有管理的属性访问指向超类获取器以避免循环：

```
1
2
get: attr3
3
----------------------------------------
get: attr1
1
get: attr2
2
get: attr3
3
```

尽管__getattribute__可以捕获比__getattr__更多的属性获取，但是实际上，它们只是一个主题的不同变体——如果属性没有物理地存储，二者具有相同的效果。

管理技术比较

为了概括我们在本章介绍的4种属性管理方法之间的编码区别，让我们快速地来看看使用每种技术的一个更全面的计算属性的示例。如下的版本使用特性来拦截并计算名为square和cube的属性。注意它们的基本值是如何存储到以下划线开头的名称中的，因此，它们不会与特性本身的名称冲突：

```
# 2 dynamically computed attributes with properties
```

```
class Powers:
    def __init__(self, square, cube):
        self._square = square                   # _square is the base value
        self._cube = cube                       # square is the property name

    def getSquare(self):
        return self._square ** 2
    def setSquare(self, value):
        self._square = value
    square = property(getSquare, setSquare)

    def getCube(self):
        return self._cube ** 3
    cube = property(getCube)

X = Powers(3, 4)
print(X.square)       # 3 ** 2 = 9
print(X.cube)         # 4 ** 3 = 64
X.square = 5
print(X.square)       # 5 ** 2 = 25
```

要用描述符做到同样的事情，我们用完整的类定义了属性。注意，描述符把基础值存储为实例状态，因此，它们必须再次使用下划线开头，以便不会与描述符的名称冲突（正如我们将在本章最后的示例中见到的，我们可以通过把基础值存储为描述符状态，从而避免必须重新命名）：

```
# Same, but with descriptors

class DescSquare:
    def __get__(self, instance, owner):
        return instance._square ** 2
    def __set__(self, instance, value):
        instance._square = value

class DescCube:
    def __get__(self, instance, owner):
        return instance._cube ** 3

class Powers:                                   # Use (object) in 2.6
    square = DescSquare()
    cube = DescCube()
    def __init__(self, square, cube):
        self._square = square                   # "self.square = square" works too,
        self._cube = cube                       # because it triggers desc __set__!

X = Powers(3, 4)
print(X.square)       # 3 ** 2 = 9
print(X.cube)         # 4 ** 3 = 64
X.square = 5
print(X.square)       # 5 ** 2 = 25
```

要使用__getattr__访问拦截来实现同样的结果，我们再次用下划线开头的名称存储基础

值，这样对被管理的名称访问是未定义的，并且由此调用我们的方法。我们还需要编写一个__setattrr__来拦截赋值，并且注意避免其潜在的循环：

```
# Same, but with generic __getattr__ undefined attribute interception

class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube =   cube
    def __getattr__(self, name):
        if name == 'square':
            return self._square ** 2
        elif name == 'cube':
            return self._cube ** 3
        else:
            raise TypeError('unknown attr:' + name)
        def __setattr__(self, name, value):
            if name == 'square':
                self.__dict__['_square'] = value
            else:
                self.__dict__[name] = value

X = Powers(3, 4)
print(X.square)       # 3 ** 2 = 9
print(X.cube)         # 4 ** 3 = 64
X.square = 5
print(X.square)       # 5 ** 2 = 25
```

最后一个选项，使用__getattribute__来编写，类似于前一个版本。由于我们现在捕获了每一个属性，因此必须把基础值获取指向超类以避免循环：

```
# Same, but with generic __getattribute__ all attribute interception

class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube
    def __getattribute__(self, name):
        if name == 'square':
            return object.__getattribute__(self, '_square') ** 2
        elif name == 'cube':
            return object.__getattribute__(self, '_cube') ** 3
        else:
            return object.__getattribute__(self, name)
     def __setattr__(self, name, value):
        if name == 'square':
           self.__dict__['_square'] = value
        else:
           self.__dict__[name] = value

X = Powers(3, 4)
print(X.square)       # 3 ** 2 = 9
print(X.cube)         # 4 ** 3 = 64
X.square = 5
```

```
print(X.square)       # 5 ** 2 = 25
```

正如你所见到的，每种技术的编码形式都有所不同，但是，所有4种方法在运行的时候都产生同样的结果：

```
9
64
25
```

要了解如何比较这些替代方案以及其他编码选项的更多内容，在本章后面“示例：属性验证”节的属性验证示例中，我们会更多地尝试它们的实际应用。在此之前，我们需要先学习和这些工具中的两种相关的一个缺点。

拦截内置操作属性

在介绍__getattr__和__getattribute__的时候，我说它们分别拦截未定义的以及所有的属性获取，这使得它们很适合用于基于委托的编码模式。尽管对于常规命名的属性来说是这样，但它们的行为需要一些额外的澄清：对于隐式地使用内置操作获取的方法名属性，这些方法可能**根本不会运行**。这意味着操作符重载方法调用不能委托给被包装的对象，除非包装类自己重新定义这些方法。

例如，针对__str__、__add__和__getitem__方法的属性获取分别通过打印、+表达式和索引隐式地运行，而不会指向Python 3.0中的类属性拦截方法。特别是：

- 在Python 3.0中，__getattr__和__getattribute__**都不会**针对这样的属性而运行。
- 在Python 2.6中，如果属性在类中未定义的话，__getattr__会针对这样的属性运行。
- 在Python 2.6中，__getattribute__只对于新式类可用，并且在Python 3.0中也可以使用。

换句话说，在Python 3.0的类中（以及Python 2.6的新式类中），没有直接的方法来通用地拦截像打印和加法这样的内置操作。在Python 2.X中，这样的操作调用的方法在运行时从实例中查找，就像所有其他属性一样；在Python 3.0中，这样的方法在**类**中查找。

这种修改使得基于委托的编码模式在Python 3.0中更为复杂，因为它们不能通用地拦截操作符重载方法调用并将它们指向一个嵌入的对象。这不是一个严重的错误——包装类可以通过在自身中重新定义所有相关的操作符重载方法，从而委托调用以解决这一约束。这些额外的方法可以手动添加，用工具添加，或者通过在共同超类中定义并从共同超类继承。然而，相对于操作符重载方法是被包装对象接口的一部分的情况，这种方法确实使包装更有用处。

记住，这个问题只适用于__getattr__和__getattribute__。由于特性和描述符只针对特定属性定义，所以它们根本不能真正应用于基于代理的类——单个特性或描述符不能用于拦截任意属性。此外，定义操作符重载方法和属性拦截的一个类将能够正确地工作，而不管定义的属性拦截的类型。我们在这里只是关心没有定义操作符重载方法但是力图通用地拦截它们的类。

考虑如下的例子，即文件*getattr.py*，它在包含了__getattr__和__getattribute__方法的类的实例上，测试各种属性类型和内置操作：

```
class GetAttr:
    eggs = 88                           # eggs stored on class, spam on instance
    def __init__(self):
        self.spam = 77
    def __len__(self):                  # len here, else __getattr__ called with __len__
        print('__len__: 42')
        return 42
    def __getattr__(self, attr):        # Provide __str__ if asked, else dummy func
        print('getattr: ' + attr)
        if attr == '__str__':
            return lambda *args: '[Getattr str]'
        else:
            return lambda *args: None

class GetAttribute(object):             # object required in 2.6, implied in 3.0
    eggs = 88                           # In 2.6 all are isinstance(object) auto
    def __init__(self):                 # But must derive to get new-style tools,
        self.spam = 77                  # incl __getattribute__, some __X__ defaults
    def __len__(self):
        print('__len__: 42')
        return 42
    def __getattribute__(self, attr):
        print('getattribute: ' + attr)
        if attr == '__str__':
            return lambda *args: '[GetAttribute str]'
        else:
            return lambda *args: None

for Class in GetAttr, GetAttribute:
    print('\n' + Class.__name__.ljust(50, '='))

    X = Class()
    X.eggs                      # Class attr
    X.spam                      # Instance attr
    X.other                     # Missing attr
    len(X)                      # __len__ defined explicitly

    try:                        # New-styles must support [], +, call directly: redefine
        X[0]                    # __getitem__?
    except:
         print('fail []')
```

```
try:
    X + 99                          # __add__?
except:
   print('fail +')

try:
    X()                             # __call__? (implicit via built-in)
except:
    print('fail ()')
X.__call__()                        # __call__? (explicit, not inherited)

print(X.__str__())                  # __str__? (explicit, inherited from type)
print(X)                            # __str__? (implicit via built-in)
```

在Python 2.6下运行的时候，__getattr__的确接收针对内置操作的各种隐式属性获取，因为Python通常在实例中查询这样的属性。相反，对于任何操作符重载名，__getattribute__**不会**运行，因为这样的名称只在类中查找：

```
C:\misc> c:\python26\python getattr.py

GetAttr=========================================
getattr: other
__len__: 42
getattr: __getitem__
getattr: __coerce__
getattr: __add__
getattr: __call__
getattr: __call__
getattr: __str__
[Getattr str]
getattr: __str__
[Getattr str]

GetAttribute====================================
getattribute: eggs
getattribute: spam
getattribute: other
__len__: 42
fail []
fail +
fail ()
getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x025EA1D0>
```

注意，在Python 2.6中，这里的__getattr__如何拦截__call__和__str__的隐式和显式获取。相反，对于内置操作的任何属性名，__getattribute__不能捕捉隐式获取。

确实，__getattribute__例子在Python 2.6中与其在Python 3.0中是相同的，因为在Python 2.6中类必须通过派生自object成为新式类，才能使用这个方法。这段代码的object派生在Python 3.0中是可选的，因为其中所有的类都是新式的。

然而，在Python 3.0下运行的时候，__getattr__的结果不同——当通过内置操作获取属性的时候，**没有**隐式运行的操作符重载方法会触发哪个属性拦截方法。在解析这样的名称的时候，Python 3.0省略了常规实例查找机制：

```
C:\misc> c:\python30\python getattr.py

GetAttr=========================================
getattr: other
__len__: 42
fail []
fail +
fail ()
getattr: __call__
<__main__.GetAttr object at 0x025D17F0>
<__main__.GetAttr object at 0x025D17F0>

GetAttribute====================================
getattribute: eggs
getattribute: spam
getattribute: other
__len__: 42
fail []
fail +
fail ()
getattribute: __call__
getattribute: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x025D1870>
```

我们可以跟踪这些输出，从而了解到脚本中的打印，看看这是如何工作的：

- 在Python 3.0中，__str__访问有两次未能被__getattr__捕获：一次是针对内置打印，一次是针对显式获取，因为从该类继承了一个默认方法（实际上，该类来自内置object，它是每个类的一个超类）。
- __str__只有一次未能被__getattribute__捕获，在内置打印操作中，显式获取绕过了继承的版本。
- __call__在Python 3.0中用于内置调用表达式的两次都没有捕获，但是，当显式获取的时候，它两次都拦截到了；和__str__不同，没有继承的__call__默认版本能够超越__getattr__。
- __len__被两个类都捕获了，直接原因是，它在类自身中是一个显式定义的方法——它的名称指明了，在Python 3.0中，如果我们删除了类的__len__方法，它不会指向__getattr__或__getattribute__。
- 所有其他的内置操作在Python 3.0中都没有被两种方案拦截。

再一次，直接的效果是，由内置操作隐式地运行的操作符重载方法始终不会通过Python

3.0中的某个属性拦截方法指向：Python 3.0在**类**中查找这样的属性，并且完全忽略了实例查找。

这使得基于委托的包装类在Python 3.0中更难以编写，如果被包装的类可能包含操作符重载方法，这些方法必须冗余地在包装类中重新定义，从而能够委托给被包装的对象。在一般的委托工具中，这可能会增加很多额外的方法。

当然，这些方法的增加可能一部分是工具自动进行的，通过用新的方法来扩展类做到（这里，下两章将要介绍的类装饰器和元类可能会有帮助）。此外，超类可能能够一次性定义所有这些额外方法，以便在基于委托的类中继承。然而，在Python 3.0中，委托编码模式需要额外的工作。

要了解关于这一现象的更实际的说明及其解决方法，参见下一章中的`Private`装饰器示例。在那里，我们将看到，也可能在客户类中插入一个`__getattribute__`从而保留其最初的类型，尽管这个方法仍然不会为了操作符重载方法而调用；例如，打印仍然直接运行在这样的一个类中定义的`__str__`，而不是通过`__getattribute__`指向请求。

作为另一个例子，下一小节重复了我们的类教程示例。既然理解了属性拦截是如何工作的，我们将能够解释稍微奇怪一点的一个例子。

注意：要查看这一Python 3.0改变在Python自身中工作的一个例子，参见本书第14章中Python 3.0的`os.popen`对象的介绍。由于它用一个包装类实现，而该包装类使用`__getattr__`把属性获取委托给一个嵌入的对象，它没有拦截Python 3.0中的`next(X)`内置迭代器函数，该函数定义为运行`__next__`。然而，它确实拦截并委托显式`X.__next__()`调用，因为这些不是通过内置函数指向的，并且没有像`__str__`那样继承自一个超类。

这等同于我们的例子中的`__call__`——对内置函数的隐式调用不会触发`__getattr__`，但对于不是继承自类类型的显式调用，则会触发`__getattr__`。换句话说，这一改变不仅影响到我们的委托者，而且会影响到Python标准库中的那些类！既然这一修改有了一定的范围，这些行为未来可能会发展，因此，确保在以后的版本中验证这个问题。

重访基于委托的Manager

第27章的面向对象教程展示了一个`Manager`类，它使用对象嵌套和方法委托来定制它的超类，而不是使用继承。这里再次引用那段代码，删除了一些不相关的测试：

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
```

```
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)          # Embed a Person object
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)          # Intercept and delegate
    def __getattr__(self, attr):
        return getattr(self.person, attr)               # Delegate all other attrs
    def __str__(self):
        return str(self.person)                         # Must overload again (in 3.0)

if __name__ == '__main__':
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(sue.lastName())
    sue.giveRaise(.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)  # Manager.__init__
    print(tom.lastName())              # Manager.__getattr__ -> Person.lastName
    tom.giveRaise(.10)                 # Manager.giveRaise -> Person.giveRaise
    print(tom)                         # Manager.__str__ -> Person.__str__
```

这个文件末尾的注释展示了对于一行的操作调用哪个方法。特别是，注意`lastName`调用如何在`Manager`中是未定义的，并由此指向通用的`__getattr__`，又从那里到了嵌入的`Person`对象。如下是这段脚本的输出——Sue从`Person`那里接收了10%的加薪，但Tom得到了20%的加薪，因为`giveRaise`在`Manager`中定制了：

```
C:\misc> c:\python30\python getattr.py
Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

相反，注意当我们在这段脚本末尾**打印**`Manager`的时候发生了什么：调用了包装类的`__str__`，并且它委托到嵌入的`Person`对象的`__str__`。记住这一点，看看如果我们在代码中**删除**了`Manager.__str__`方法，将会发生什么事情：

```
# Delete the Manager __str__ method

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)          # Embed a Person object
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)          # Intercept and delegate
    def __getattr__(self, attr):
        return getattr(self.person, attr)               # Delegate all other attrs
```

现在，在Python 3.0下，针对Manager对象，打印**不会**通过通用的__getattr__拦截器指向其属性获取。相反，继承自类的隐式object超类的一个默认__str__ 显示方法，被查找到并运行（sue仍然正确地打印，因为Person有一个显式的__str__）：

```
C:\misc> c:\python30\python person.py
Jones
[Person: Sue Jones, 110000]
Jones
<__main__.Manager object at 0x02A5AE30>
```

奇怪的是，像这样没有一个__str__运行，在Python 2.6中**却会**触发__getattr__，因为操作符重载属性通过这个方法指向，并且类不会为__str__继承一个默认版本：

```
C:\misc> c:\python26\python person.py
Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

切换到__getattribute__在这里也不会对Python 3.0有所帮助——像__getattr__一样，对于Python 2.6和Python 3.0中内置操作所暗示的操作符重载属性，它都不会运行：

```
# Replace __getattr_ with __getattribute__

class Manager:                                          # Use (object) in 2.6
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)          # Embed a Person object
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)          # Intercept and delegate
    def __getattribute__(self, attr):
        print('**', attr)
        if attr in ['person', 'giveRaise']:
            return object.__getattribute__(self, attr)  # Fetch my attrs
        else:
            return getattr(self.person, attr)           # Delegate all others
```

不管在Python 3.0中使用哪个属性拦截方法，我们仍然必须在Manager中包含一个重新定义的__str__（如上面所示），以便拦截打印操作并将它们指向嵌入的Person对象：

```
C:\misc> c:\python30\python person.py
Jones
[Person: Sue Jones, 110000]
** lastName
** person
Jones
** giveRaise
** person
<__main__.Manager object at 0x028E0590>
```

注意，__getattribute__针对方法获得**两次**调用——一次针对方法名，另一次针对

self.person嵌入对象获取。我们可以用一种不同的编码方式来避免如此，但是，我们仍然必须重定义__str__以捕获打印，尽管这里有所不同（self.person将导致__getattribute__失效）：

```
# Code __getattribute__ differently to minimize extra calls

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'mgr', pay)
    def __getattribute__(self, attr):
        print('**', attr)
        person = object.__getattribute__(self, 'person')
        if attr == 'giveRaise':
            return lambda percent: person.giveRaise(percent+.10)
        else:
            return getattr(person, attr)
        def __str__(self):
            person = object.__getattribute__(self, 'person')
            return str(person)
```

运行这一替代方案的时候，我们的对象正确地打印了，但是只是因为我们已经在包装类中添加了一个显式的__str__——这个属性仍然没有指向通用的属性拦截方法：

```
Jones
[Person: Sue Jones, 110000]
** lastName
Jones
** giveRaise
[Person: Tom Jones, 60000]
```

这里简单介绍了像Manager这样的基于委托的类，在Python 3.0中必须重定义某些操作符重载方法（例如__str__）才能将它们指向嵌套的对象，但是，在Python 2.6中不必，除非使用了新式类。我们唯一的直接选择似乎是使用__getattr__和Python 2.6，或者在Python 3.0中在包装类中冗余地重定义操作符重载方法。

再一次说明，这不是一个不可能的任务。很多包装类可以预计所需的操作符重载方法的集合，并且工具和超类可以将这个任务的一部分自动化。此外，并非所有的类都使用操作符重载方法（实际上，大多数应用程序类通常不会使用）。然而，对于在Python 3.0中使用的委托编码模式，需要记住一些事情。当操作符重载方法是一个对象的接口的一部分时，包装类必须通过在本地重新定义它们来容纳它们。

示例：属性验证

为了结束本章的内容，让我们来看一个更实际的示例，以所有的4种属性管理方案来编写代码。我们将要使用的这个示例定义了一个CardHolder对象，它带有4个属性，其中3个属性是要管理的。管理的属性在获取或存储的时候要验证或转换值。对于同样的测试

代码，所有4个版本都产生同样的结果，但是，它们以不同的方式实现了它们的属性。这个示例包含了很大一部分需要自学的内容。然而我们不会详细介绍其代码，因为它们都使用了我们在本章中已经介绍过的概念。

使用特性来验证

我们的第一段代码使用了特性来管理3个属性。与通常一样，我们可以使用简单的方法而不是管理属性，但是，如果我们在已有的代码中已经使用了属性，特性就能帮忙了。特性根据属性访问自动运行代码，但是关注属性的一个特定集合，它们不会用来广泛地拦截所有属性。

要理解这段代码，关键是要注意到，`__init__`构造函数方法内部的属性赋值也触发了特性的`setter`方法。例如，当这个方法分配给`self.name`时，它自动调用`setName`方法，该方法转换值并将其赋给一个叫做`__name`的实例属性，以便它不会与特性的名称冲突。

这一重命名（有时候叫做**名称压缩**）是必要的，因为特性使用公用的实例状态并且没有自己的实例状态。存储在一个属性中的数据叫做`__name`，而叫做`name`的属性总是特性，而非数据。

最后，这个类管理了叫做`name`、`age`和`acct`的属性；允许直接访问属性`addr`，并且提供了一个名为`remain`的只读属性，该属性完全是虚拟的并且根据需要计算。为了进行比较，这个基于特性的程序包含了39行代码：

```
class CardHolder:
    acctlen = 8                                 # Class data
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                        # Instance data
        self.name = name                        # These trigger prop setters too
        self.age = age                          # __X mangled to have class name
        self.addr = addr                        # addr is not managed
                                                # remain has no data
    def getName(self):
        return self.__name
    def setName(self, value):
        value = value.lower().replace(' ', '_')
        self.__name = value
    name = property(getName, setName)

    def getAge(self):
        return self.__age

    def setAge(self, value):
        if value < 0 or value > 150:
            raise ValueError('invalid age')
        else:
```

```
            self.__age = value
    age = property(getAge, setAge)

    def getAcct(self):
        return self.__acct[:-3] + '***'
    def setAcct(self, value):
        value = value.replace('-', '')
        if len(value) != self.acctlen:
            raise TypeError('invald acct number')
        else:
            self.__acct = value
    acct = property(getAcct, setAcct)

    def remainGet(self):                        # Could be a method, not attr
        return self.retireage - self.age        # Unless already using as attr
    remain = property(remainGet)
```

self测试代码

如下的代码测试我们的类；将这段代码添加到文件的底部，或者把类放到一个模块中并先导入它。我们将对这个例子的所有4个版本使用这段同样的测试代码。当它运行的时候，我们创建了管理的属性类的两个实例，并且获取和修改其各种属性。期待失效的操作包装在try语句中：

```
bob = CardHolder('1234-5678', 'Bob Smith', 40, '123 main st')
print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=' / ')
bob.name = 'Bob Q. Smith'
bob.age = 50
bob.acct = '23-45-67-89'
print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=' / ')

sue = CardHolder('5678-12-34', 'Sue Jones', 35, '124 main st')
print(sue.acct, sue.name, sue.age, sue.remain, sue.addr, sep=' / ')
try:
    sue.age = 200
except:
    print('Bad age for Sue')

try:
    sue.remain = 5
except:
    print("Can't set sue.remain")

try:
    sue.acct = '1234567'
except:
    print('Bad acct for Sue')
```

如下是我们的self测试代码的输出。再一次说明，这对这个示例的所有版本都是一样的。分析这段代码，看看类的方法是如何调用的。账户显示出来，其中一些数字隐藏

了，名称转换为一种标准格式，并且使用一个类属性访问拦截的时候，截止到退休的剩余时间就计算了出来：

```
12345*** / bob_smith / 40 / 19.5 / 123 main st
23456*** / bob_q._smith / 50 / 9.5 / 123 main st
56781*** / sue_jones / 35 / 24.5 / 124 main st
Bad age for Sue
Can't set sue.remain
Bad acct for Sue
```

使用描述符验证

现在，让我们使用描述符而不是使用特性来重新编写示例。正如我们已经看到的，描述符在功能和角色上与特性类似。实际上，特性基本上是描述符的一种受限制的形式。和特性一样，描述符设计来处理特定的属性访问，而不是通用的属性访问。和特性不同，描述符有自己的状态，并且它们是一种更为通用的方案。

要理解这段代码，注意`__init__`构造函数方法内部的属性赋值触发了描述符的`__set__`操作符方法，这一点还是很重要。例如，当构造函数方法分配给`self.name`时，它自动调用`Name.__set__()`方法，该方法转换值，并且将其赋给了叫做`name`的一个描述符属性。

和前面基于特性的变体不同，在这个例子中，实际的`name`值附加到了**描述符**对象，而不是客户类实例。尽管我们可以把这个值存储在实例状态或描述符状态中，后者避免了需要用下划线压缩名称以避免冲突。在`CardHolder`客户类中，名为`name`的属性总是一个描述符对象，而不是数据。

最后，这个类实现了和前面的版本同样的属性：它管理名为`name`、`age`和`acct`的属性。允许直接访问属性`addr`，并且提供一个名为`remain`的只读属性，`remain`是完全虚拟的并且根据需要计算。注意我们为何在其描述符中捕获对`remain`的赋值，并引发一个异常。正如我们前面所介绍的，如果没有这么做，对一个实例这一属性的赋值，将会默默地创建一个实例属性而隐藏了类属性描述符。为了进行比较，这个基于描述符的代码占了45行：

```
class CardHolder:
    acctlen = 8                                 # Class data
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                        # Instance data
        self.name = name                        # These trigger __set__ calls too
    self.age = age                              # __X not needed: in descriptor
    self.addr = addr                            # addr is not managed
                                                # remain has no data
class Name:
```

```
    def __get__(self, instance, owner):            # Class names: CardHolder locals
        return self.name
    def __set__(self, instance, value):
        value = value.lower().replace(' ', '_')
        self.name = value
name = Name()

class Age:
    def __get__(self, instance, owner):
        return self.age                            # Use descriptor data
    def __set__(self, instance, value):
        if value < 0 or value > 150:
            raise ValueError('invalid age')
        else:
            self.age = value
age = Age()

class Acct:
    def __get__(self, instance, owner):
        return self.acct[:-3] + '***'
    def __set__(self, instance, value):
        value = value.replace('-', '')
        if len(value) != instance.acctlen:         # Use instance class data
            raise TypeError('invald acct number')
        else:
            self.acct = value
acct = Acct()

class Remain:
    def __get__(self, instance, owner):
        return instance.retireage - instance.age   # Triggers Age.__get__
    def __set__(self, instance, value):
        raise TypeError('cannot set remain')       # Else set allowed here
remain = Remain()
```

使用__getattr__来验证

正如我们已经见到的，__getattr__方法拦截所有未定义的属性，因此，它可能比使用特性或描述符更为通用。在我们的例子中，当获取一个管理的属性的时候，我们通过直接测试属性名来获知。其他的属性物理地存储在实例中，因而无法达到__getattr__。尽管这种方法比使用特性或描述符更为通用，但需要额外的工作来模拟专门关注属性的其他工具。我们需要在运行时检查名称，并且必须编写一个__setattr__以拦截并验证属性赋值。

对于这个例子的特性和描述符版本，注意__init__构造函数方法中的属性赋值触发了类的__setattr__方法，这还是很关键的。例如，当这个方法分配给self.name时，它自动地调用_setattr_方法，该方法转换值，并将其分配给一个名为name的实例属性。通过在该实例上存储name，它确保了未来的访问不会触发__getattr__。相反，acct存储为_acct，因此随后对acct的访问会调用__getattr__。

最后，像前两个例子中的情况一样，这个类管理名为name、age和acct的属性。允许直接访问属性addr；并且提供一个名为remain的只读属性，它是完全虚拟的并且根据需要计算。

为了进行比较，这个替代方法有32行代码——比基于特性的版本少了7行，比使用描述符的版本少了13行。当然，清晰与否比代码大小更重要，但额外的代码有时候意味着额外的开发和维护工作。可能这里更重要的是**角色**：像__getattr__这样的通用工具可能更适合于通用委托，而特性和描述符更直接是为了管理特定属性而设计。

还要注意，当设置未管理的属性（例如，addr）的时候，这里的代码引发**额外调用**，然而获取未管理的属性并不会引发额外调用，因为它们是定义了的。尽管这可能对大多数程序都会导致不可忽视的额外开销，但只有当访问管理的属性的时候，特性和描述符才会引发额外调用。

下面是代码的__getattr__版本：

```
class CardHolder:
    acctlen = 8                                     # Class data
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                            # Instance data
        self.name = name                            # These trigger __setattr__ too
        self.age = age                              # _acct not mangled: name tested
        self.addr = addr                            # addr is not managed
                                                    # remain has no data
    def __getattr__(self, name):
        if name == 'acct':                          # On undefined attr fetches
            return self._acct[:-3] + '***'          # name, age, addr are defined
        elif name == 'remain':
            return self.retireage - self.age        # Doesn't trigger __getattr__
        else:
            raise AttributeError(name)

    def __setattr__(self, name, value):
        if name == 'name':                          # On all attr assignments
            value = value.lower().replace(' ', '_') # addr stored directly
        elif name == 'age':                         # acct mangled to _acct
            if value < 0 or value > 150:
                raise ValueError('invalid age')
        elif name == 'acct':
            name = '_acct'
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('invald acct number')
        elif name == 'remain':
            raise TypeError('cannot set remain')
        self.__dict__[name] = value                 # Avoid looping
```

使用__getattribute__验证

最后的变体使用__getattribute__在需要的时候拦截属性获取并管理它们。这里，每次属性获取都会捕获，因此，我们测试属性名称来检测管理的属性，并将所有其他的属性指向超类以实现常规的获取过程。这个版本和前面的版本一样，使用了同样的__setattr__来捕获赋值。

这段代码的工作和__getattr__版本很相似，因此，我不想在这里重复整个介绍。然而，注意，由于每个属性获取都指向了__getattribute__，所以这里我们不需要压缩名称以拦截它们（acct存储为acct）。另一方面，这段代码必须负责把未压缩的属性获取指向一个超类以避免循环。

还要注意，对于设置和获取未管理的属性（例如，addr），这个版本都会引发额外调用。如果速度极为重要，这个替代方法可能会是所有方案中最慢的。为了进行比较，这个版本也有32行代码，和前面的版本一样：

```
class CardHolder:
    acctlen = 8                                 # Class data
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                        # Instance data
        self.name = name                        # These trigger __setattr__ too
        self.age = age                          # acct not mangled: name tested
        self.addr = addr                        # addr is not managed
                                                # remain has no data
    def __getattribute__(self, name):
        superget = object.__getattribute__              # Don't loop: one level up
        if name == 'acct':                              # On all attr fetches
            return superget(self, 'acct')[:-3] + '***'
        elif name == 'remain':
            return superget(self, 'retireage') - superget(self, 'age')
        else:
            return superget(self, name)                 # name, age, addr: stored

    def __setattr__(self, name, value):
        if name == 'name':                              # On all attr assignments
            value = value.lower().replace(' ', '_')     # addr stored directly
        elif name == 'age':
            if value < 0 or value > 150:
                raise ValueError('invalid age')
        elif name == 'acct':
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('invald acct number')
       elif name == 'remain':
           raise TypeError('cannot set remain')
       self.__dict__[name] = value      # Avoid loops, orig names
```

要确保自己学习和运行本节中的代码，以获得关于管理属性编码技术的更多技巧。

本章小结

本章介绍了在Python中管理对属性进行访问的各种技术，包括__getattr__和__getattribute__操作符重载方法、类特性和属性描述符。此外，还比较和对比了这些工具，并给出了一些用例来展示它们的行为。

第38章继续我们的构建工具学习，来看看装饰器，即在函数和类创建的时候而不是属性访问的时候运行的代码。在继续学习之前，让我们来看一组练习题，以复习在本章学习的内容。

本章习题

1. __getattr__和 __getattribute__有何区别？
2. 特性和描述符有何区别？
3. 特性和装饰器有何关联？
4. __getattr__和__getattribute__以及特性和描述符之间主要的功能区别是什么？
5. 所有这些功能的比较只是某种争论吗？

习题解答

1. __getattr__方法只针对**未定义**属性的获取运行，即那些没有在一个实例上显示的属性，以及没有从它的任何类继承的属性。相反，__getattribute__方法针对所有的属性获取运行，不管属性是否定义了。因此，__getattr__中的代码可以自由地获取其他属性，如果它们定义了的话；而__getattribute__针对所有这样的属性获取使用特定代码以避免循环（它必须把获取指向一个超类以跳过自身）。
2. 特性充当一个特定角色，而描述符更为通用。特性定义了特定属性的获取、设置和删除功能。描述符也提供了一个类，带有完成这些操作的方法，但是，它们提供了额外的灵活性以支持更多任意行为。实际上，特性真的只是创建特定描述符的一种简单方法——即在属性访问上运行的一个描述符。编码上也有区别：特性通过一个内置函数创建，而描述符用一个类来编码；同样，描述符可以利用类的所有常用OOP功能，例如继承。此外，除了实例的状态信息，描述符有它们自己的本地状态，因此，它们可以避免在实例中的名称冲突。

3. 特性可以用装饰器语法编写。由于property内置函数接受一个单个的函数参数，它可以直接用作一个函数装饰器来定义一个获取访问特性。由于名称重新绑定装饰器的行为，所以被装饰的函数的名称分配给了一个特性，而特性的获取访问器设置为最初装饰的函数（name = property(name)）。特性的setter和deleter属性允许我们进一步用装饰器语法添加设置和删除访问器——它们把访问器设置为装饰的函数并且返回扩展的特性。

4. __getattr__和 __getattribute__方法更为通用：它们用来捕获任意多的属性。相反，每个特性或描述符只针对一个特定属性提供访问拦截——我们不能用一个单个的特性或描述符捕获每个属性获取。另一方面，特性和描述符都通过设计来处理属性获取和赋值：__getattr__和__getattribute__只处理获取；要拦截赋值，必须编写__setattr__。实现也是不同的：__getattr__和__getattribute__是操作符重载方法，而特性和描述符是手动赋给类属性的对象。

5. 并非如此。引用Python同名的喜剧*Monty Python's Flying Circus*中的台词：

```
An argument is a connected series of statements intended to establish a
proposition.
No it isn't.
Yes it is! It's not just contradiction.
Look, if I argue with you, I must take up a contrary position.
Yes, but that's not just saying "No it isn't."
Yes it is!
No it isn't!
Yes it is!
No it isn't. Argument is an intellectual process. Contradiction is just
the automatic gainsaying of any statement the other person makes.
(short pause)
No it isn't.
It is.
Not at all.
Now look...
```

第38章

装饰器

在本书的高级类主题一章中（第31章），我们介绍了静态方法和类方法，并且快速浏览了Python提供的声明装饰器的语法`@ decorator`。我们还在上一章（第37章）遇到过函数装饰器，并探讨了`property`内置函数充当装饰器的能力，在第28章中，我们还学习了抽象超类的概念。

本章选择了前面对装饰器介绍所没有涉及的内容。这里，我们将深入装饰器的内部工作机制，并学习自己编写新的装饰器的更多高级方法。正如我们将要了解到的，我们在前面各章学习到的很多概念（例如状态保持），会在装饰器中常规出现。

这是一个颇为高级的话题，并且装饰器的构建对于工具构架者比对于应用程序员的意义更大。此外，装饰器在流行Python框架中变得越来越常见，对其基本的理解有助于认识它们的作用，即便你只是一个装饰器的用户。

除了详细介绍装饰器的构建，本章还作为装饰器在Python中应用的更为实际的**案例研究**。由于本章的示例比我们在本书其他各章见到的示例都要大，它们更好地说明了代码如何组合为较为完整的系统和工具。作为额外的好处，我们在这里编写的很多代码可以作为通用的工具用于日常编程之中。

什么是装饰器

装饰是为函数和类指定管理代码的一种方式。装饰器本身的形式是处理其他的可调用对象的可调用的对象（如函数）。正如我们在本书前面所见到过的，Python装饰器以两种相关的形式呈现：

- *函数装饰器*在函数定义的时候进行名称重绑定，提供一个逻辑层来管理函数和方法或随后对它们的调用。
- *类装饰器*在类定义的时候进行名称重绑定，提供一个逻辑层来管理类，或管理随后调用它们所创建的示例。

简而言之，装饰器提供了一种方法，在函数和类定义语句的末尾插入**自动运行代码**——对于函数装饰器，在def的末尾；对于类装饰器，在class的末尾。这样的代码可以扮演不同的角色，参见后面小节的介绍。

管理调用和实例

例如，通常的用法中，这种自动运行的代码可能用来增强对函数和类的调用。它通过针对随后的调用安装**包装器对象**来实现这一点：

- 函数装饰器安装包装器对象，以在需要的时候拦截随后的**函数调用**并处理它们。
- 类装饰器安装包装器对象，以在需要的时候拦截随后的**实例创建调用**并处理它们。

装饰器通过自动把函数和类名重绑定到其他的可调用对象来实现这些效果，在def和class语句的末尾做到这点。当随后调用的时候，这些可调用对象可以执行诸如对函数调用跟踪和计时、管理对类实例属性的访问等任务。

管理函数和类

尽管本章的大多数实例都使用包装器来拦截随后对函数和类的调用，但这并非使用装饰器的唯一方法：

- *函数装饰器*也可以用来管理**函数**对象，而不是随后对它们的调用——例如，把一个函数注册到一个API。然而，我们在这里主要关注更为常见的用法，即调用包装器应用程序。
- *类装饰器*也可以用来直接管理类对象，而不是实例创建调用——例如，用新的方法扩展类。因为这些用法和**元类**有很大的重合（实际上，都是在类创建过程的最后运行），我们将在下一章看到更多的例子。

换句话说，函数装饰器可以用来管理函数调用和函数对象，类装饰器可以用来管理类实例和类自身。通过返回装饰的对象自身而不是一个包装器，装饰器变成了针对函数和类的一种简单的后创建步骤。

不管扮演什么样的角色，装饰器都提供了一种方便而明确的方法，来编写在程序开发阶段和现实产品系统中都有用的工具。

使用和定义装饰器

根据你的工作形式，你可能成为装饰器的用户或提供者。正如我们所看到的，Python本身带有具有特定角色的内置装饰器——静态方法装饰器、属性装饰器以及更多。此外，很多流行的Python工具包括了执行管理数据库或用户接口逻辑等任务的装饰器。在这样的情况中，我们不需要知道装饰器如何编码就可以完成任务。

对于更为通用的任务，程序员可以编写自己的任意装饰器。例如，函数装饰器可能通过添加跟踪调用、在调试时执行参数验证测试、自动获取和释放线程锁、统计调用函数的次数以进行优化等的代码来扩展函数。你可以想象添加到函数调用中的任何行为，都可以作为定制函数装饰器的备选。

另外一方面，函数装饰器设计用来只增强一个特定函数或方法**调用**，而不是一个完整的对象**接口**。类装饰器更好地充当后一种角色——因为它们可以拦截实例创建调用，它们可以用来实现任意的对象接口扩展或管理任务。例如，定制的类装饰器可以跟踪或验证对一个对象的每个属性引用。它们也可以用来实现代理对象、单体类以及其他常用的编程模式。实际上，我们将会发现很多类装饰器与在第30章中见到的**委托**编程模式有很大的相似之处。

为什么使用装饰器

像很多高级Python工具一样，从纯技术的视角来看，并不是严格需要装饰器：它们的功能往往可以使用简单的辅助函数调用或其他的技术来实现（并且从基本的层面出发，我们总是可以手动地编写装饰器所自动执行的名称重绑定）。

也就是说，装饰器为这样的任务提供了一种显式的语法，它使得意图明确，可以最小化扩展代码的冗余，并且有助于确保正确的API使用：

- 装饰器有一种非常**明确的**语法，这使得它们比那些可能任意地远离主体函数或类的辅助函数调用更容易为人们发现。
- 当主体函数或类定义的时候，装饰器应用**一次**；在对类或函数的每次调用的时候，不必添加额外的代码（在未来可能必须改变）。
- 由于前面两点，装饰器使得一个API的用户不太可能**忘记**根据API需求扩展一个函数或类。

换句话说，除了其技术模型之外，装饰器提供了一些和代码维护性和审美相关的优点。此外，作为结构化工具，装饰器自然地促进了代码的**封装**，这减少了冗余性并使得未来变得更容易。

装饰器确实也有一些潜在的**缺点**——当它们插入包装类的逻辑，它们可以修改装饰的对象的类型，并且它们可能引发额外的调用。另外一方面，同样的考虑也适用于任何为对象添加包装逻辑的技术。

我们将在本章随后的真实代码中说明这些权衡。尽管选择使用装饰器仍然多少有些主观性，但它们的优点引人注目，足以使其快速成为Python世界中的最佳实践。为了帮助你做出决定，让我们来看一些细节。

基础知识

让我们首先从一个符号的角度来第一次看一看装饰行为。我们很快将编写真正的代码，但是，由于装饰器的很多神奇之处可归结为自动重绑定操作，所以首先理解这一映射是很重要的。

函数装饰器

函数装饰器已经从Python 2.5开始可用。正如我们在本书前面所见到的，它们主要只是一种语法糖：通过在一个函数的`def`语句的末尾来运行另一个函数，把最初的函数名重新绑定到结果。

用法

函数装饰器是一种关于函数的**运行时声明**，函数的定义需要遵守此声明。装饰器在紧挨着定义一个函数或方法的`def`语句之前的一行编写，并且它由`@`符号以及紧随其后的对于**元函数**的一个引用组成——这是管理另一个函数的一个函数（或其他的可调用对象）。

在编码方面，函数装饰器自动将如下的语法：

```
@decorator              # Decorate function
def F(arg):
    ...

F(99)                   # Call function
```

映射为这一对等的形式，其中装饰器是一个单参数的可调用对象，它返回与`F`具有相同数目的参数的一个可调用对象：

```
def F(arg):
    ...
F = decorator(F)        # Rebind function name to decorator result

F(99)                   # Essentially calls decorator(F)(99)
```

这一自动名称重绑定在`def`语句上有效，不管它针对一个简单的函数或是类中的一个方

法。当随后调用F函数的时候，它自动调用装饰器所**返回的**对象，该对象可能是实现了所需的包装逻辑的另一个对象，或者是最初的函数本身。

换句话说，装饰实际把如下的第一行映射为第二行（尽管装饰器实际上只运行一次，在装饰的时候）：

```
func(6, 7)
decorator(func)(6, 7)
```

这一自动名称重绑定说明了我们在本书前面遇到的静态方法和正确的装饰语法的原因：

```
class C:
    @staticmethod
    def meth(...): ...          # meth = staticmethod(meth)

class C:
    @property
    def name(self): ...         # name = property(name)
```

在这两个例子中，在`def`语句的末尾，方法名重新绑定到一个内置函数装饰器的结果。随后再调用最初的名称，将会调用装饰器所返回的对象。

实现

装饰器自身是一个**返回可调用对象的可调用对象**。也就是说，它返回了一个对象，当随后装饰的函数通过其最初的名称调用的时候，将会调用这个对象——不管是拦截了随后调用的一个包装器对象，还是最初的函数以某种方式的扩展。实际上，装饰器可以是任意类型的可调用对象，并且返回任意类型的可调用对象：函数和类的任何组合都可以使用，尽管一些组合更适合于特定的背景。

例如，要在一个函数创建之后接入装饰协议以管理函数，我们需要编写如下形式的装饰器：

```
def decorator(F):
    # Process function F
    return F

@decorator
def func(): ...                 # func = decorator(func)
```

由于最初的装饰函数分配回给其名称，这么做将直接向函数的定义添加创建之后的步骤。这样的一个结构可能会用来把一个函数注册到一个API、赋值函数属性，等等。

更典型的用法，是插入逻辑以拦截对函数的随后调用，我们可以编写一个装饰器来返回和最初函数不同的一个对象：

```
def decorator(F):
```

```
    # Save or use function F
    # Return a different callable: nested def, class with __call__, etc.

@decorator
def func(): ...                 # func = decorator(func)
```

这个装饰器在装饰的时候调用，并且当随后调用最初的函数名的时候，它所返回的调用对象将被调用。装饰器自身接受被装饰的函数，返回的调用对象会接受随后传递给被装饰函数的名称的任何参数。这和类方法的工作方式相同：隐含的实例对象只是在返回的可调用对象的第一个参数中出现。

更概括地说，有一种常用的编码模式可以包含这一思想——装饰器返回了一个包装器，包装器把最初的函数保持到一个封闭的作用域中：

```
def decorator(F):                       # On @ decoration
    def wrapper(*args):                 # On wrapped function call
        # Use F and args
        # F(*args) calls original function
    return wrapper

@decorator                              # func = decorator(func)
def func(x, y):                         # func is passed to decorator's F
    ...

func(6, 7)                              # 6, 7 are passed to wrapper's *args
```

当随后调用名称`func`的时候，它确实调用装饰器所返回的包装器函数；随后包装器函数可能会运行最初的`func`，因为它在一个**封闭的作用域**中仍然可以使用。当以这种方式编码的时候，每个装饰的函数都会产生一个新的作用域来保持状态。

为了对类做同样的事情，我们可以重载调用操作，并且使用实例属性而不是封闭的作用域：

```
class decorator:
    def __init__(self, func):          # On @ decoration
        self.func = func
    def __call__(self, *args):         # On wrapped function call
        # Use self.func and args
        # self.func(*args) calls original function

@decorator
def func(x, y):                        # func = decorator(func)
    ...                                # func is passed to __init__

func(6, 7)                             # 6, 7 are passed to __call__'s *args
```

现在，随后再调用`func`的时候，它确实会调用装饰器所创建的实例的`__call__`运算符重载方法；然后，`__call__`方法可能运行最初的`func`，因为它在一个**实例属性**中仍然可

用。当按照这种方式编写代码的时候，每个装饰的函数都会产生一个新的实例来保持状态。

支持方法装饰

关于前面的基于类的代码的细微的一点是，尽管它对于拦截简单函数调用有效，但当它应用于类方法函数的时候，并不是很有效：

```
class decorator:
    def __init__(self, func):              # func is method without instance
        self.func = func
    def __call__(self, *args):             # self is decorator instance
        # self.func(*args) fails! # C instance not in args!

class C:
    @decorator
    def method(self, x, y):                # method = decorator(method)
        ...                                # Rebound to decorator instance
```

当按照这种方式编码的时候，装饰的方法重绑定到装饰器类的一个实例，而不是一个简单的函数。

这一点带来的问题是，当装饰器的`__call__`方法随后运行的时候，其中的`self`接收装饰器类实例，并且类C的实例不会包含到一个`*args`中。这使得有可能把调用分派给最初的方法——即保持了最初的方法函数的装饰器对象，但是，没有实例传递给它。

为了支持函数和方法，嵌套函数的替代方法工作得更好：

```
def decorator(F):                          # F is func or method without instance
    def wrapper(*args):                    # class instance in args[0] for method
        # F(*args) runs func or method
    return wrapper

@decorator
def func(x, y):                            # func = decorator(func)
    ...
func(6, 7)                                 # Really calls wrapper(6, 7)

class C:
    @decorator
    def method(self, x, y):                # method = decorator(method)
        ...                                # Rebound to simple function

X = C()
X.method(6, 7)                             # Really calls wrapper(X, 6, 7)
```

当按照这种方法编写的包装类在其第一个参数里接收了C类实例的时候，它可以分派到最初的方法和访问状态信息。

从技术上讲，这种嵌套函数版本是有效的，因为Python创建了一个绑定的方法对象，并

且由此只有当一个方法属性引用一个简单的函数的时候，才把主体类实例传递给self参数；相反，当它引用可调用的类的一个实例的时候，可调用的类的实例传递给self，以允许可调用的类访问自己的状态信息。在本章随后，我们还将看到这一细微的区别在实际实例中的作用。

还要注意，嵌套函数可能是支持函数和方法的装饰的最直接方式，但是不一定是唯一的方式。例如，上一章中的描述符，调用的时候接收了描述符和主体类实例。然而，更为复杂的是，在本章稍后，我们将看到这一工具如何在这一背景下起作用。

类装饰器

函数装饰器已经证明了是如此有用，以至于这一模式在Python 2.6和Python 3.0中扩展为允许类装饰器。类装饰器与函数装饰器密切相关，实际上，它们使用相同的语法和非常相似的编码模式。然而，不是包装单个的函数或方法，类装饰器是管理类的一种方式，或者用管理或扩展类所创建的实例的额外逻辑，来包装实例构建调用。

用法

从语法上讲，类装饰器就像前面的class语句一样（就像前面函数定义中出现的函数装饰器）。在语法上，假设装饰器是返回一个可调用对象的一个单参数的函数，类装饰器语法：

```
@decorator                      # Decorate class
class C:
      ...

x = C(99)                       # Make an instance
```

等同于下面的语法——类自动地传递给装饰器函数，并且装饰器的结果返回来分配给类名：

```
class C:
    ...
C = decorator(C)                # Rebind class name to decorator result

x = C(99)                       # Essentially calls decorator(C)(99)
```

直接的效果就是，随后调用类名会创建一个实例，该实例会触发装饰器所返回的可调用对象，而不是调用最初的类自身。

实现

新的类装饰器使用函数装饰器所使用的众多相同的技术来编码。由于类装饰器也是**返回一个可调用对象的一个可调用对象**，因此大多数函数和类的组合已经足够了。

尽管先编码，但装饰器的结果是当随后创建一个实例的时候才运行的。例如，要在一个类创建之后直接管理它，返回最初的类自身：

```
def decorator(C):
    # Process class C
    return C

@decorator
class C: ...                                    # C = decorator(C)
```

不是插入一个包装器层来拦截随后的实例创建调用，而是返回一个不同的可调用对象：

```
def decorator(C):
    # Save or use class C
    # Return a different callable: nested def, class with __call__, etc.

@decorator
class C: ...                                    # C = decorator(C)
```

这样的一个类装饰器返回的可调用对象，通常创建并返回最初的类的一个新的实例，以某种方式来扩展对其接口的管理。例如，下面的实例插入一个对象来拦截一个类实例的未定义的属性：

```
def decorator(cls):                             # On @ decoration
    class Wrapper:
        def __init__(self, *args):              # On instance creation
            self.wrapped = cls(*args)
        def __getattr__(self, name):            # On attribute fetch
            return getattr(self.wrapped, name)
    return Wrapper

@decorator
class C:                                # C = decorator(C)
    def __init__(self, x, y):           # Run by Wrapper.__init__
        self.attr = 'spam'

x = C(6, 7)                             # Really calls Wrapper(6, 7)
print(x.attr)                           # Runs Wrapper.__getattr__, prints "spam"
```

在这个例子中，装饰器把类的名称重新绑定到另一个类，这个类在一个封闭的作用域中保持了最初的类，并且当调用它的时候，创建并嵌入了最初的类的一个实例。当随后从该实例获取一个属性的时候，包装器的`__getattr__`拦截了它，并且将其委托给最初的类的嵌入的实例。此外，每个被装饰的类都创建一个新的作用域，它记住了最初的类。在本章后面，我们将用一些更有用的代码来充实这个例子。

就像函数装饰器一样，类装饰器通常可以编写为一个创建并返回可调用的对象的“工厂”函数，或者使用`__init__`或`__call__`方法来拦截所有调用操作的类，或者是由此产生的一些组合。工厂函数通常在封闭的作用域引用中保持状态，类通常在属性中保持状态。

支持多个实例

和函数装饰器一样，使用类装饰器的时候，一些可调用对象组合比另一些工作得更好。考虑前面例子的类装饰器的一个如下的无效替代方式：

```
class Decorator:
    def __init__(self, C):                          # On @ decoration
        self.C = C
    def __call__(self, *args):                      # On instance creation
        self.wrapped = self.C(*args)
        return self
    def __getattr__(self, attrname):                # On atrribute fetch
        return getattr(self.wrapped, attrname)

@Decorator
class C: ...                                        # C = Decorator(C)

x = C()
y = C()                                             # Overwrites x!
```

这段代码处理多个被装饰的类（每个都产生一个新的`Decorator`实例），并且会拦截实例创建调用（每个运行`__call__`方法）。然而，和前面的版本不同，这个版本没有能够处理给定的类的**多个实例**——每个实例创建调用都覆盖了前面保存的实例。最初的版本确实支持多个实例，因为每个实例创建调用产生了一个新的独立的包装器对象。更通俗地说，如下模式中的每一个都支持多个包装的实例：

```
def decorator(C):                                   # On @ decoration
    class Wrapper:
    def __init__(self, *args):                      # On instance creation
        self.wrapped = C(*args)
    return Wrapper

class Wrapper: ...
def decorator(C):                                   # On @ decoration
    def onCall(*args):                              # On instance creation
        return Wrapper(C(*args))                    # Embed instance in instance
    return onCall
```

我们将在本章随后一个更为实用的环境中研究这一现象，然而，在实际中，我们必须小心地正确组合可调用类型以支持自己的意图。

装饰器嵌套

有的时候，一个装饰器不够。为了支持多步骤的扩展，装饰器语法允许我们向一个装饰的函数或方法添加包装器逻辑的多个层。当使用这一功能的时候，每个装饰器必须出现在自己的一行中。这种形式的装饰器语法：

```
@A
@B
```

```
@C
def f(...):
    ...
```

如下这样运行：

```
def f(...):
    ...
f = A(B(C(f)))
```

这里，最初的函数通过3个不同的装饰器传递，并且最终的可调用对象返回来分配给最初的名称。每个装饰器处理前一个的结果，这可能是最初的函数或一个插入的包装器。

如果所有的装饰器都插入包装器，直接的效果就是，当调用最初的函数名时，将会调用包装对象逻辑的3个不同的层，从而以3种不同的方式扩展最初的函数。列出的最后的装饰器是第一次应用的并且最深层次的嵌套。

就像对函数一样，多个类装饰器导致了多个嵌套的函数调用，并且可能导致围绕实例创建调用的包装器逻辑的多个层。例如，如下的代码：

```
@spam
@eggs
class C:
    ...

X = C()
```

等同于如下的代码：

```
class C:
    ...
C = spam(eggs(C))

X = C()
```

再次，每个装饰器都自由地返回最初的类或者一个插入的包装器对象。有了包装器，当最终请求最初C类的一个实例的时候，这一调用会重定向到spam和eggs装饰器提供的包装层对象，二者可能有任意的不同角色。

例如，如下的什么也不做的装饰器只是返回被装饰的函数：

```
def d1(F): return F
def d2(F): return F
def d3(F): return F

@d1
@d2
@d3
```

```
def func():                    # func = d1(d2(d3(func)))
    print('spam')

func()                         # Prints "spam"
```

同样的语法在类上也有效，就像这里什么也不做的装饰器一样。

然而，当装饰器插入包装器函数对象，调用的时候它们可能扩展最初的函数——如下的代码将其结果连接到一个装饰器层中，随着它从内向外地运行层：

```
def d1(F): return lambda: 'X' + F()
def d2(F): return lambda: 'Y' + F()
def d3(F): return lambda: 'Z' + F()

@d1
@d2
@d3
def func():                # func = d1(d2(d3(func)))
    return 'spam'

print(func())              # Prints "XYZspam"
```

我们在这里使用了`lambda`函数来实现包装器层（每个层在一个封闭的作用域里保持了包装的函数）。实际上，包装器可以采取函数、可调用的类以及更多形式。当设计良好的时候，装饰器嵌套允许我们以种类多样的方式来组合扩展步骤。

装饰器参数

函数装饰器和类装饰器似乎都能接受**参数**，尽管实际上这些参数传递给了真正返回装饰器的一个可调用对象，而装饰器反过来又**返回**一个可调用对象。例如，如下代码：

```
@decorator(A, B)
def F(arg):
    ...
F(99)
```

自动地映射到其对等的形式，其中装饰器是一个可调用对象，它返回实际的装饰器。返回的装饰器反过来返回可调用的对象，这个对象随后运行以调用最初的函数名：

```
def F(arg):
    ...
F = decorator(A, B)(F)         # Rebind F to result of decorator's return value

F(99)                          # Essentially calls decorator(A, B)(F)(99)
```

装饰器参数在装饰发生之前就解析了，并且它们通常用来保持状态信息供随后的调用使用。例如，这个例子中的装饰器函数，可能采用如下的形式：

```
def decorator(A, B):
```

```
# Save or use A, B
def actualDecorator(F):
    # Save or use function F
    # Return a callable: nested def, class with __call__, etc.
    return callable
return actualDecorator
```

这个结构中的外围函数通常会把装饰器参数与状态信息分开保存，以便在实际的装饰器中使用，或者在它所返回的可调用对象中使用，或者在二者中都使用。这段代码在封闭的函数作用域引用中保存了状态信息参数，但是通常也可以使用类属性。

换句话说，装饰器参数往往意味着**可调用对象的3个层级**：接受装饰器参数的一个可调用对象，它返回一个可调用对象以作为装饰器，该装饰器返回一个可调用对象来处理对最初的函数或类的调用。这3个层级的每一个都可能是一个函数或类，并且可能以作用域或类属性的形式保存了状态。我们将在本章后面看到应用装饰器参数的实际例子。

装饰器管理函数和类

尽管本章剩下的很大篇幅集中在包装对函数和类的随后调用，但我应该强调装饰器机制比这更加通用——它是在函数和类创建之后通过一个可调用对象传递它们的一种协议。因此，它可以用来调用任意的创建后处理：

```
def decorate(O):
    # Save or augment function or class O
    return O

@decorator
def F(): ...                    # F = decorator(F)

@decorator
class C: ...                    # C = decorator(C)
```

只要以这种方式返回最初装饰的对象，而不是返回一个包装器，我们就可以管理函数和类自身，而不只是管理随后对它们的调用。在本章稍后，我们将看到使用这一思想的更为实际的例子，它们用装饰把可调用对象注册到一个API，并且在创建函数的时候为它们赋值属性。

编写函数装饰器

现回到代码层面。在本章剩下的内容里，我们将学习实际的例子来展示刚刚介绍的装饰器概念。本小节展示几个函数装饰器的实际例子，下一小节展示实际的类装饰器例子。此后，我们将通过使用类和函数装饰器的一些较大的例子来结束本章。

跟踪调用

首先，让我们回顾一下第31章介绍的跟踪器的例子。如下的代码定义并应用一个函数装饰器，来统计对装饰的函数的调用次数，并且针对每一次调用打印跟踪信息：

```
class tracer:
    def __init__(self, func):          # On @ decoration: save original func
        self.calls = 0
        self.func = func
    def __call__(self, *args):         # On later calls: run original func
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        self.func(*args)

@tracer
def spam(a, b, c):                     # spam = tracer(spam)
    print(a + b + c)                   # Wraps spam in a decorator object
```

注意，用这个类装饰的每个函数将如何创建一个新的实例，带有自己保存的函数对象和调用计数器。还要注意观察，`*args`参数语法如何用来打包和解包任意的多个传入参数。这一通用性使得这个装饰器可以用来包装带有任意多个参数的任何函数（这个版本还不能在类方法上工作，但是，我们将在本小节稍后修改这一点）。

现在，如果导入这个模块的函数并交互地测试它，将会得到如下的一种行为——每次调用都初始地产生一条跟踪信息，因为装饰器类拦截了调用。这段代码在Python 2.6和Python 3.0下都能运行，就像本章中所有其他的代码一样，除非特别提示：

```
>>> from decorator1 import spam

>>> spam(1, 2, 3)                  # Really calls the tracer wrapper object
call 1 to spam
6

>>> spam('a', 'b', 'c')            # Invokes __call__ in class
call 2 to spam
abc

>>> spam.calls                     # Number calls in wrapper state information
2

>>> spam
<decorator1.tracer object at 0x02D9A730>
```

运行的时候，`tracer`类和装饰的函数分开保存，并且拦截对装饰的函数随后的调用，以便添加一个逻辑层来统计和打印每次调用。注意，调用的总数如何作为装饰的函数的一个属性显示——装饰的时候，`spam`实际上是`tracer`类的一个实例（对于进行类型检查的程序，可能还会衍生一次查找，但是通常是有益的）。

对于函数调用，@装饰语法可能比修改每次调用来说明额外的逻辑层要更加方便，并且它避免了意外地直接调用最初的函数。考虑如下所示的非装饰器的对等代码：

```
calls = 0
def tracer(func, *args):
    global calls
    calls += 1
    print('call %s to %s' % (calls, func.__name__))
    func(*args)

def spam(a, b, c):
    print(a, b, c)

>>> spam(1, 2, 3)              # Normal non-traced call: accidental?
1 2 3

>>> tracer(spam, 1, 2, 3)      # Special traced call without decorators
call 1 to spam
1 2 3
```

这一替代方法可以用在任何函数上，且不需要特殊的@语法，但是和装饰器版本不同，它在代码中调用函数的每个地方需要额外的语法。此外，它的意图可能不够明显，并且它不能确保额外的层将会针对常规调用而调用。尽管装饰器不是必需的（我们总是可以手动地重新绑定名称），它们通常是最为方便的。

状态信息保持选项

前面小节的最后一个例子引发了一个重要的问题。函数装饰器有各种选项来保持装饰的时候所提供的状态信息，以便在实际函数调用过程中使用。它们通常需要支持多个装饰的对象以及多个调用，但是，有多种方法来实现这些目标：实例属性、全局变量、非局部变量和函数属性，都可以用于保持状态。

类实例属性

例如，这里是前面的例子的一个扩展版本，其中添加了对**关键字**参数的支持，并且返回包装函数的结果，以支持更多的用例：

```
class tracer:                                   # State via instance attributes
    def __init__(self, func):                   # On @ decorator
        self.calls = 0                          # Save func for later call
        self.func = func
    def __call__(self, *args, **kwargs):        # On call to original function
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)

@tracer
def spam(a, b, c):            # Same as: spam = tracer(spam)
```

```
    print(a + b + c)          # Triggers tracer.__init__

@tracer
def eggs(x, y):               # Same as: eggs = tracer(eggs)
    print(x ** y)             # Wraps eggs in a tracer object

spam(1, 2, 3)                 # Really calls tracer instance: runs tracer.__call__
spam(a=4, b=5, c=6)           # spam is an instance attribute

eggs(2, 16)                   # Really calls tracer instance, self.func is eggs
eggs(4, y=4)                  # self.calls is per-function here (need 3.0 nonlocal)
```

就像最初的版本一样，这里的代码使用**类实例属性**来显式地保存状态。包装的函数和调用计数器都是针对**每个实例**的信息——每个装饰都有自己的拷贝。当在Python 2.6和Python 3.0下运行一段脚本的时候，这个版本的输出如下所示。注意spam和eggs函数的每一个是如何有自己的调用计数器的，因为每个装饰都创建一个新的类实例：

```
call 1 to spam
6
call 2 to spam
15
call 1 to eggs
65536
call 2 to eggs
256
```

尽管对于装饰函数有用，但是当应用于方法的时候，这种编码方案也有问题（随后更为详细地介绍）。

封闭作用域和全局作用域

封闭def作用域引用和嵌套的def常常可以实现相同的效果，特别是对于装饰的最初函数这样的静态数据。然而，在这个例子中，我们也需要封闭的作用域中的一个计数器，它随着每次调用而更改，并且，这在Python 2.6中是不可能的。在Python 2.6中，我们可以使用类和属性，正如我们前面所做的那样，或者使用全局声明把状态变量移出到**全局作用域**：

```
calls = 0
def tracer(func):                       # State via enclosing scope and global
    def wrapper(*args, **kwargs):       # Instead of class attributes
        global calls                    # calls is global, not per-function
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
     return wrapper

@tracer
def spam(a, b, c):                      # Same as: spam = tracer(spam)
    print(a + b + c)
```

```
@tracer
def eggs(x, y):                 # Same as: eggs = tracer(eggs)
    print(x ** y)

spam(1, 2, 3)                   # Really calls wrapper, bound to func
spam(a=4, b=5, c=6)             # wrapper calls spam

eggs(2, 16)                     # Really calls wrapper, bound to eggs
eggs(4, y=4)                    # Global calls is not per-function here!
```

遗憾的是，把计数器移出到共同的全局作用域允许像这样修改它们，也意味着它们将为每个包装的函数所共享。和类实例属性不同，全局计数器是跨程序的，而不是针对每个函数的——对于任何跟踪的函数调用，计数器都会递增。如果你比较这个版本与前一个版本的输出，就可以看出其中的区别——单个的、共享的全局调用计数器根据每次装饰函数的调用不正确地更新：

```
call 1 to spam
6
call 2 to spam
15
call 3 to eggs
65536
call 4 to eggs
256
```

封闭作用域和nonlocal

共享全局状态可能是我们在某些情况下想要的。如果我们真的想要一个针对每个函数的计数器，要么像前面那样使用类，要么使用Python 3.0中新的`nonlocal`语句，第17章曾介绍过该语句。由于这一新的语句允许修改封闭的函数作用域变量，所以它们可以充当针对每次装饰的、可修改的数据：

```
def tracer(func):                       # State via enclosing scope and nonlocal
    calls = 0                           # Instead of class attrs or global
    def wrapper(*args, **kwargs):       # calls is per-function, not global
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return wrapper

@tracer
def spam(a, b, c):              # Same as: spam = tracer(spam)
    print(a + b + c)

@tracer
def eggs(x, y):                 # Same as: eggs = tracer(eggs)
    print(x ** y)

spam(1, 2, 3)                   # Really calls wrapper, bound to func
spam(a=4, b=5, c=6)             # wrapper calls spam
```

```
eggs(2, 16)                 # Really calls wrapper, bound to eggs
eggs(4, y=4)                # Nonlocal calls _is_ not per-function here
```

现在，由于封装的作用域变量不能跨程序而成为全局的，所以每个包装的函数再次有了自己的计数器，就像是针对类和属性一样。这里是在Python 3.0下运行时新的输出：

```
call 1 to spam
6
call 2 to spam
15
call 1 to eggs
65536
call 2 to eggs
256
```

函数属性

最后，如果你没有使用Python 3.X并且没有一条nonlocal语句，可能仍然能够针对某些可改变的状态使用**函数属性**来避免全局和类。在最新的Pythons中，我们可以把任意属性分配给函数以附加它们，使用func.*attr=value*就可以了。在我们的例子中，可以直接对状态使用wrapper.calls。如下的代码与前面的nonlocal版本一样地工作，因为计数器再一次是针对每个装饰的函数的，但是，它也可以在Python 2.6下运行：

```
def tracer(func):                       # State via enclosing scope and func attr
    def wrapper(*args, **kwargs):       # calls is per-function, not global
        wrapper.calls += 1
        print('call %s to %s' % (wrapper.calls, func.__name__))
        return func(*args, **kwargs)
    wrapper.calls = 0
    return wrapper
```

注意，这种方法有效，只是因为名称wrapper保持在封闭的tracer函数的作用域中。当我们随后增加wrapper.calls时，并不是在修改名称wrapper本身，因此，不需要nonlocal声明。

这种方案几乎作为一个脚注来介绍，因为它比Python 3.0中的nonlocal要隐晦得多，并且可能留待其他方案无济于事的情况下使用更好。然而，我们将解答本章末尾一个问题的时候使用它，那里，我们需要从装饰器代码的**外部**访问保存的状态；nonlocal只能从嵌套函数自身的内部看到，但是函数属性有更广泛的可见性。

由于装饰器往往意味着可调用对象的多个层级，所以我们可以用封闭的作用域和带有属性的类来组合函数，以实现各种各样的编码结构。正如我们随后将见到的，这有时候可能比我们所期待的要细微——每个装饰的函数应该有自己的状态，并且每个装饰的类都应该需要针对自己的状态和针对每个产生实例的状态。

实际上，正如下一小节所介绍的，如果我们也想要对一个类方法应用函数装饰器，必须小心Python在作为可调用类实例对象的装饰器编码和作为函数的装饰器编码之间的区分。

类错误之一：装饰类方法

当我编写上面的第一个tracer函数的时候，我幼稚地假设它也应该适用于任何方法——装饰的方法应该同样地工作，但是，自动的self实例参数应该直接包含在*args的前面。遗憾的是，我错了：当应用于类方法的时候，tracer的第一个版本失效了，因为self是装饰器类的实例，并且装饰的主体类的实例没有包含在*args中。在Python 3.0和Python 2.6中都是如此。

我在本章前面介绍了这一现象，但是现在，我们可以在真实的工作代码环境中看到它。假设基于类的跟踪装饰器如下：

```
class tracer:
      def __init__(self, func):                  # On @ decorator
          self.calls = 0                         # Save func for later call
          self.func = func
      def __call__(self, *args, **kwargs):       # On call to original function
          self.calls += 1
          print('call %s to %s' % (self.calls, self.func.__name__))
          return self.func(*args, **kwargs)
```

简单函数的装饰与前面介绍的一样：

```
@tracer
def spam(a, b, c):                               # spam = tracer(spam)
    print(a + b + c)                             # Triggers tracer.__init__

spam(1, 2, 3)                                    # Runs tracer.__call__
spam(a=4, b=5, c=6)                              # spam is an instance attribute
```

然而，类方法的装饰失效了（更明白的读者可能会认识到，这是我们在第27章面向对象教程中的Person类的再现）：

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

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

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

```
bob = Person('Bob Smith', 50000)                # tracer remembers method funcs
bob.giveRaise(.25)                              # Runs tracer.__call__(???, .25)
print(bob.lastName())                           # Runs tracer.__call__(???)
```

这里问题的根源在于，tracer类的__call__方法的self——它是一个tracer实例，还是一个Person实例？我们真的需要将其编写为**两者都是**：tracer用于装饰器状态，Person用于指向最初的方法。实际上，self**必须**是tracer对象，以提供对tracer的状态信息的访问；不管装饰一个简单的函数还是一个方法，都是如此。

遗憾的是，当我们用__call__把装饰方法名重绑定到一个类实例对象的时候，Python只向self传递了tracer**实例**；它根本没有在参数列表中传递Person主体。此外，由于tracer不知道我们要用方法调用处理的Person实例的任何信息，没有办法创建一个带有一个实例的绑定的方法，因此，没有办法正确地分配调用。

实际上，前面的列表最终传递了太少的参数给装饰的方法，并且导致了一个错误。在装饰器的__call__方法添加一行，以打印所有的参数来验证这一点。正如你所看到的，self是一个tracer，并且Person实例完全缺失：

```
<__main__.tracer object at 0x02D6AD90> (0.25,) {}
call 1 to giveRaise
Traceback (most recent call last):
  File "C:/misc/tracer.py", line 56, in <module>
    bob.giveRaise(.25)
  File "C:/misc/tracer.py", line 9, in __call__
    return self.func(*args, **kwargs)
TypeError: giveRaise() takes exactly 2 positional arguments (1 given)
```

正如前面提到的，出现这种情况是因为：当一个方法名绑定只是绑定到一个简单的函数，Python向self传递了隐含的主体实例；当它是一个可调用类的实例的时候，就传递这个类的实例。从技术上讲，当方法是一个简单函数的时候，Python只是创建了一个绑定的方法对象，其中包含了主体实例。

使用嵌套函数来装饰方法

如果想要函数装饰器在简单函数和类方法上**都能**工作，最直接的解决方法在于使用前面介绍的状态保持方法之一——把自己的函数装饰器编写为嵌套的def，以便对于包装器类实例和主体类实例都不需要依赖于单个的self实例参数。

如下的替代方案使用Python 3.0的nonlocal。由于装饰的方法重新绑定到简单的函数而不是实例对象，所以Python正确地传递了Person对象作为第一个参数，并且装饰器将其从*args中的第一项传递给真正的、装饰的方法的self参数：

```
# A decorator for both functions and methods
```

```
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

# Applies to simple functions

@tracer
def spam(a, b, c):                      # spam = tracer(spam)
    print(a + b + c)                    # onCall remembers spam

spam(1, 2, 3)                           # Runs onCall(1, 2, 3)
spam(a=4, b=5, c=6)

# Applies to class method functions too!

class Person:
    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]

print('methods...')
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), lastName in scopes
```

这个版本在函数和方法上都有效：

```
call 1 to spam
6
call 2 to spam
15
methods...
Bob Smith Sue Jones
call 1 to giveRaise
110000.0
call 1 to lastName
call 2 to lastName
Smith Jones
```

使用描述符装饰方法

尽管前一小节介绍的嵌套函数的解决方案是支持应用于函数和类方法的装饰器的最直接方法，其他的方法也是可能的。例如，我们在上一章中介绍的描述符功能，在这里也能派上用场。

还记得我们在前一章的讨论中，描述符可能是分配给对象的一个类属性，该对象带有一个__get__方法，当引用或获取该属性的时候自动运行该方法（在Python 2.6中需要对象派生，但在Python 3.0中不需要）：

```
class Descriptor(object):
    def __get__(self, instance, owner): ...

class Subject:
    attr = Descriptor()

X = Subject()
X.attr                # Roughly runs Descriptor.__get__(Subject.attr, X, Subject)
```

描述符也能够拥有__set__和__del__访问方法，但是，我们在这里不需要它们。现在，由于描述符的__get__方法在调用的时候接收描述符类和主体类实例，因此当我们需要装饰器的状态以及最初的类实例来分派调用的时候，它很适合于装饰方法。考虑如下的替代的跟踪装饰器，它也是一个描述符：

```
class tracer(object):
    def __init__(self, func):                    # On @ decorator
        self.calls = 0                           # Save func for later call
        self.func = func
    def __call__(self, *args, **kwargs):         # On call to original func
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):          # On method attribute fetch
        return wrapper(self, instance)

class wrapper:
    def __init__(self, desc, subj):              # Save both instances
        self.desc = desc                         # Route calls back to decr
        self.subj = subj
    def __call__(self, *args, **kwargs):
        return self.desc(self.subj, *args, **kwargs)     # Runs tracer.__call__

@tracer
def spam(a, b, c):                               # spam = tracer(spam)
    ...same as prior...                          # Uses __call__ only

class Person:
    @tracer
    def giveRaise(self, percent):                # giveRaise = tracer(giverRaise)
        ...same as prior...                      # Makes giveRaise a descriptor
```

这和前面的嵌套的函数代码一样有效。装饰的函数只调用其__call__，而装饰的方法首先调用其__get__来解析方法名获取（在*instance.method*上）；__get__返回的对象保持主体类实例并且随后调用以完成调用表达式，由此触发__call__。例如，要测试代码的调用：

```
sue.giveRaise(.10)                              # Runs __get__ then __call__
```

首先运行tracer.__get__，因为Person类的giveRaise属性已经通过函数装饰器重新绑定到了一个描述符。然后，调用表达式触发返回的包装器对象的__call__方法，它返回来调用tracer.__call__。

包装器对象同时保持描述符和主体实例，因此，它可以将控制指回到最初的装饰器/描述符类实例。实际上，在方法属性获取过程中，包装的对象保持了主体类实例可用，并且将其添加到了随后调用的参数列表，该参数列表会传递给__call__。在这个应用程序中，用这种方法把调用路由到描述符类实例是需要的，由此对包装方法的所有调用都使用描述符实例对象中的同样的调用计数器状态信息。

此外，我们也可以使用一个嵌套的函数和封闭的作用域引用来实现同样的效果——如下的版本和前面的版本一样的有效，通过为一个嵌套函数和作用域引用交换类和对象属性，但是，它所需的代码显著减少：

```
class tracer(object):
    def __init__(self, func):                  # On @ decorator
        self.calls = 0                         # Save func for later call
        self.func = func
    def __call__(self, *args, **kwargs):       # On call to original func
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):                # On method fetch
        def wrapper(*args, **kwargs):                  # Retain both inst
        return self(instance, *args, **kwargs)         # Runs __call__
    return wrapper
```

为这些替代方法添加print语句是为了自己跟踪获取/调用过程的两个步骤，用前面嵌套函数替代方法中同样的测试代码来运行它们。在两种编码中，基于描述符的方法也比嵌套函数的选项要细致得多，因此，它可能是这里的又一种选择。在其他的环境中，它也可能是一种有用的编码模式。

在本章剩余的内容中，我们将相当随意地使用类或函数来编写函数装饰器，只要它们都只适用于函数。一些装饰器可能并不需要最初的类的实例，并且如果编写为一个类，它将在函数和方法上都有效——例如Python自己的静态方法装饰器，就不需要主体类的一个实例（实际上，它主要是从调用中删除实例）。

然而，这里的叙述的教训是，如果你想要装饰器在简单函数和类方法上都有效，最好使用基于嵌套函数的编码模式，而不是带有调用拦截的类。

计时调用

为了展示函数装饰器的各种各样能力的一个特殊样例，让我们来看一种不同的应用场景。下一个装饰器将对一个装饰的函数的调用进行计时——既有针对一次调用的时间，也有所有调用的总的时间。该装饰器应用于两个函数，以便比较列表解析和map内置调用所需的时间（为了便于比较，参见本书第20章中另一个非装饰器的示例，它可以作为这里的计时迭代替代方案）：

```
import time

class timer:
    def __init__(self, func):
        self.func   = func
        self.alltime = 0
    def __call__(self, *args, **kargs):
        start = time.clock()
        result = self.func(*args, **kargs)
        elapsed = time.clock() - start
        self.alltime += elapsed
        print('%s: %.5f, %.5f' % (self.func.__name__, elapsed, self.alltime))
        return result

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer
def mapcall(N):
    return map((lambda x: x * 2), range(N))

result = listcomp(5)                    # Time for this call, all calls, return value
listcomp(50000)
listcomp(500000)
listcomp(1000000)
print(result)
print('allTime = %s' % listcomp.alltime)        # Total time for all listcomp calls

print('')
result = mapcall(5)
mapcall(50000)
mapcall(500000)
mapcall(1000000)
print(result)
print('allTime = %s' % mapcall.alltime)         # Total time for all mapcall calls

print('map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))
```

在这个例子中，一种非装饰器的方法允许主体函数用于计时或不用于计时，但是，当需

要计时的时候，它也会使调用签名变得复杂（我们需要在每个调用的时候添加代码，而不是在def中添加一次代码），并且可能没有直接的方法来保证一个程序中的所有列表生成器调用可以通过计时器逻辑路由，在找到所有签名并潜在地修改它们方面有所不足。

在Python 2.6中运行的时候，这个文件的self测试代码的输出如下：

```
listcomp: 0.00002, 0.00002
listcomp: 0.00910, 0.00912
listcomp: 0.09105, 0.10017
listcomp: 0.17605, 0.27622
[0, 2, 4, 6, 8]
allTime = 0.276223304917

mapcall: 0.00003, 0.00003
mapcall: 0.01363, 0.01366
mapcall: 0.13579, 0.14945
mapcall: 0.27648, 0.42593
[0, 2, 4, 6, 8]
allTime = 0.425933533452
map/comp = 1.542
```

测试细微差别：我没有在Python 3.0下运行这段代码，因为正如第14章所介绍的，map内置函数在Python 3.0中返回一个**迭代器**，而不是像在Python 2.6中那样返回一个实际的列表。由此，Python 3.0的map不能和一个列表解析的工作直接对应（即，map测试实际上在Python 3.0中没有花时间）。

如果你想要在Python 3.0下运行这段代码，那么就使用list(map())来迫使它像列表解析那样构建一个列表，否则，就不是真正地进行同类比较。然而，不要在Python 2.6中这么做，如果这么做了，map测试将会负责构建两个列表，而不是一个。

如下的代码可能会对Python 2.6和Python 3.0都公平，然而要注意，尽管这会使得Python 2.6和Python 3.0中列表解析和map之间的比较更公平，但因为range在Python 3.0中也是一个迭代器，因此Python 2.6和Python 3.0的结果不会直接比较：

```
...
import sys

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

if sys.version_info[0] == 2:
   @timer
   def mapcall(N):
       return map((lambda x: x * 2), range(N))
else:
    @timer
```

```
    def mapcall(N):
        return list(map((lambda x: x * 2), range(N)))
    ...
```

最后，正如我们在本书的模块部分所了解到的，如果你想要能够在其他模块中重用这个装饰器，应该在文件的底部一个__name__ == '__main__'测试的下面缩进self测试代码，以便只有当文件运行的时候才运行它，而不是当导入它的时候运行。然而，我们不会这么做，因为打算给代码添加另一个功能。

添加装饰器参数

前面小节介绍的计时器装饰器有效，但是如果它更加可配置的话，那会更好——例如，提供一个输出标签并且可以打开或关闭跟踪消息，这些在一个通用目的的工具中可能很有用。装饰器参数在这里派上了用场：对它们适当编码后，我们可以使用它们来指定配置选项，这些选项可以根据每个装饰的函数而编码。例如，可以像下面这样添加标签：

```
def timer(label=''):
    def decorator(func):
        def onCall(*args):                      # args passed to function
            ...                                 # func retained in enclosing scope
            print(label, ...                    # label retained in enclosing scope
        return onCall
    return decorator                            # Returns that actual decorator

@timer('==>')                                   # Like listcomp = timer('==>')(listcomp)
def listcomp(N): ...                            # listcomp is rebound to decorator

listcomp(...)                                   # Really calls decorator
```

这段代码添加了一个封闭的作用域来保持一个装饰器参数，以便随后真正调用的时候使用。当定义了listcomp函数的时候，它真的调用decorator（timer的结果，在真正装饰发生之前运行），带有其封闭的作用域内可用的label值。也就是说，timer返回decorator，后者记住了装饰器参数和最初的函数，并且返回一个可调用的对象，这个可调用对象在随后的调用时调用最初的函数。

我们可以把这种结构用于定时器之中，来允许在装饰的时候传入一个标签和一个跟踪控制标志。下面是这么做的一个例子，编码在一个名为*mytools.py*的模块文件中，以便它可以作为一个通用工具导入：

```
import time

def timer(label='', trace=True):                # On decorator args: retain args
    class Timer:
        def __init__(self, func):               # On @: retain decorated func
            self.func = func
            self.alltime = 0
```

```
        def __call__(self, *args, **kargs):      # On calls: call original
            start = time.clock()
            result = self.func(*args, **kargs)
            elapsed = time.clock() - start
            self.alltime += elapsed
            if trace:
               format = '%s %s: %.5f, %.5f'
               values = (label, self.func.__name__, elapsed, self.alltime)
               print(format % values)
           return result
    return Timer
```

我们在这里做的主要是把最初的`Timer`类嵌入一个封闭的函数中，以便创建一个作用域以保持装饰器参数。外围的`timer`函数在装饰发生前调用，并且它只是返回`Timer`类作为实际的装饰器。在装饰时，创建了`Timer`的一个实例来记住装饰函数自身，而且访问了位于封闭的函数作用域中的装饰器参数。

这一次，不是把`self`测试代码嵌入这个文件，我们将在一个不同的文件中运行装饰器。下面是时间装饰器的一个客户，模块文件*testseqs.py*，再次将其应用于序列迭代器替代方案：

```
from mytools import timer

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

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

for func in (listcomp, mapcall):
    print('')
    result = func(5)           # Time for this call, all calls, return value
    func(50000)
    func(500000)
    func(1000000)
    print(result)
    print('allTime = %s' % func.alltime)         # Total time for all calls

print('map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))
```

再一次说明，如果想要在Python 3.0中正常地运行这段代码，把`map`函数包装到一个`list`调用中。当在Python 2.6中运行的时候，这文件打印出如下的输出——每个装饰的函数现在都有了一个子集的标签，该标签由装饰器参数定义：

```
[CCC]==> listcomp: 0.00003, 0.00003
[CCC]==> listcomp: 0.00640, 0.00643
[CCC]==> listcomp: 0.08687, 0.09330
[CCC]==> listcomp: 0.17911, 0.27241
[0, 2, 4, 6, 8]
```

```
allTime = 0.272407666337

[MMM]==> mapcall: 0.00004, 0.00004
[MMM]==> mapcall: 0.01340, 0.01343
[MMM]==> mapcall: 0.13907, 0.15250
[MMM]==> mapcall: 0.27907, 0.43157
[0, 2, 4, 6, 8]
allTime = 0.431572169089
map/comp = 1.584
```

与通常一样，我们也可以交互地测试它，看看配置参数是如何应用的：

```
>>> from mytools import timer
>>> @timer(trace=False)                    # No tracing, collect total time
... def listcomp(N):
...     return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
>>> x = listcomp(5000)
>>> x = listcomp(5000)
>>> listcomp
<mytools.Timer instance at 0x025C77B0>
>>> listcomp.alltime
0.0051938863738243413

>>> @timer(trace=True, label='\t=>')     # Turn on tracing
... def listcomp(N):
...     return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
        => listcomp: 0.00155, 0.00155
>>> x = listcomp(5000)
        => listcomp: 0.00156, 0.00311
>>> x = listcomp(5000)
        => listcomp: 0.00174, 0.00486
>>> listcomp.alltime
0.0048562736325408196
```

这个计时函数装饰器可以用于任何函数，在模块中和在交互模式下都可以。换句话，它自动获得作为脚本中计时代码的资格。查看本章后面的“实现私有属性”小节的装饰器参数的示例，以及在本章后面的“针对位置参数的一个基本范围测试装饰器”小节。

注意：计时方法：本小节的计数器装饰器在任何函数上都有效，但是，要将其应用于类方法上，需要进行细小的改写。简而言之，正如我们在本章前面的“类错误之一：装饰类方法”小节所介绍的，必须避免使用一个嵌套的类。由于这一变化将会是我们本章末尾测试题的主题，所以在这里，我们暂时不会给出完整的解答。

编写类装饰器

到目前为止，我们已经编写了函数装饰器来管理函数调用，但是，正如我们已经见到的，Python 2.6和Python 3.0扩展了装饰器使其也能在类上有效。如同前面所提到的，尽管类似于函数装饰器的概念，但类装饰器应用于类——它们可以用于管理类自身，或者用来拦截实例创建调用以管理实例。和函数装饰器一样，类装饰器其实只是可选的语法糖，尽管很多人相信，它们使得程序员的意图更为明显并且能使不正确的调用最小化。

单体类

由于类装饰器可以拦截实例创建调用，所以它们可以用来管理一个类的所有实例，或者扩展这些实例的接口。为了说明这点，这里的第一个类装饰器示例做了前面一项工作——管理一个类的所有实例。这段代码实现了传统的**单体**编码模式，其中最多只有一个类的一个实例存在。其单体函数为管理的属性定义并返回一个函数，并且@语法自动在这个函数中包装了一个主体类：

```
instances = {}
def getInstance(aClass, *args):                 # Manage global table
    if aClass not in instances:                 # Add **kargs for keywords
        instances[aClass] = aClass(*args)       # One dict entry per class
    return instances[aClass]

def singleton(aClass):                          # On @ decoration
    def onCall(*args):                          # On instance creation
        return getInstance(aClass, *args)
    return onCall
```

为了使用它，装饰用来强化单体模型的类：

```
@singleton                                      # Person = singleton(Person)
class Person:                                   # Rebinds Person to onCall
      def __init__(self, name, hours, rate):    # onCall remembers Person
          self.name = name
          self.hours = hours
          self.rate = rate
      def pay(self):
          return self.hours * self.rate

@singleton                                      # Spam = singleton(Spam)
class Spam:                                     # Rebinds Spam to onCall
    def __init__(self, val):                    # onCall remembers Spam
        self.attr = val

bob = Person('Bob', 40, 10)                     # Really calls onCall
print(bob.name, bob.pay())

sue = Person('Sue', 50, 20)                     # Same, single object
print(sue.name, sue.pay())
```

```
X = Spam(42)                                # One Person, one Spam
Y = Spam(99)
print(X.attr, Y.attr)
```

现在，当Person或Spam类稍后用来创建一个实例的时候，装饰器提供的包装逻辑层把实例构建调用指向了onCall，它反过来调用getInstance，以针对每个类管理并分享一个单个实例，而不管进行了多少次构建调用。这段代码的输出如下：

```
Bob 400
Bob 400
42 42
```

有趣的是，这里如果能像前面介绍的那样，使用nonlocal语句（在Python 3.0及其以后版本中可用）来改变封闭的作用域名称，我们在这里可以编写一个更为自包含的解决方案——后面的替代方案实现了同样的效果，它为每个类使用了一个**封闭作用域**，而不是为每个类使用一个全局表入口：

```
def singleton(aClass):                      # On @ decoration
    instance = None
    def onCall(*args):                      # On instance creation
        nonlocal instance                   # 3.0 and later nonlocal
        if instance == None:
           instance = aClass(*args)         # One scope per class
        return instance
    return onCall
```

这个版本同样地工作，但是，它不依赖于装饰器之外的全局作用域中的名称。在Python 2.6或Python 3.0版本中，我们也可以用类编写一个自包含的解决方案——如下代码对每个类使用一个实例，而不是使用一个封闭作用域或全局表，并且它和其他的两个版本一样地工作（实际上，依赖于我们随后会见到的同样的编码模式是一个公用的装饰器类错误，这里我们只想要一个实例，但并不总是这样的情况）：

```
class singleton:
    def __init__(self, aClass):             # On @ decoration
        self.aClass = aClass
        self.instance = None
    def __call__(self, *args):              # On instance creation
        if self.instance == None:
           self.instance = self.aClass(*args)   # One instance per class
        return self.instance
```

要让这个装饰器成为一个完全通用的工具，可将其存储在一个可导入的模块文件中，在一个__name__检查下缩进self测试代码，并且在构建调用中使用**kargs语法支持关键字参数（我们将在建议的练习中讨论它）。

跟踪对象接口

前面小节的单体示例使用类装饰器来管理一个类的**所有**实例。类装饰器的另一个常用场景是**每个**产生实例的接口。类装饰器基本上可以在实例上安装一个包装器逻辑层，来以某种方式管理对其接口的访问。

例如，在第30章中，__getattr__运算符重载方法作为包装嵌入的实例的整个对象接口的一种方法，以便实现委托编码模式。我们在前一章介绍的管理的属性中看到过类似的例子。还记得吧，当获取未定义的属性名的时候，__getattr__会运行；我们可以使用这个钩子来拦截一个控制器类中的方法调用，并将它们传递给一个嵌入的对象。

为了便于参考，这里给出最初的非装饰器委托示例，它在两个内置类型对象上工作：

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object                    # Save object
    def __getattr__(self, attrname):
        print('Trace:', attrname)                # Trace fetch
        return getattr(self.wrapped, attrname)  # Delegate fetch

>>> x = Wrapper([1,2,3])                         # Wrap a list
>>> x.append(4)                                  # Delegate to list method
Trace: append
>>> x.wrapped                                    # Print my member
[1, 2, 3, 4]

>>> x = Wrapper({"a": 1, "b": 2})                # Wrap a dictionary
>>> list(x.keys())                               # Delegate to dictionary method
Trace: keys                                      # Use list() in 3.0
['a', 'b']
```

在这段代码中，Wrapper类拦截了对任何包装的对象的属性的访问，打印出一条跟踪信息，并且使用内置函数getattr来终止对包装对象的请求。它特别跟踪包装的对象的类**之外**发出的属性访问。在包装的对象内部访问其方法不会被捕获，并且会按照设计正常运行。这种整体接口模型和函数装饰器的行为不同，装饰器只包装一个特定的方法。

类装饰器为编写这种__getattr__技术来包装一个完整接口提供了一个替代的、方便的方法。例如，在Python 2.6和Python 3.0中，前面的类示例可能编写为一个类装饰器，来触发包装的实例创建，而不是把一个预产生的实例传递到包装器的构造函数中（在这里也用**kargs扩展了，以支持关键字参数，并且统计进行访问的次数）：

```
def Tracer(aClass):                                  # On @ decorator
    class Wrapper:
        def __init__(self, *args, **kargs):          # On instance creation
            self.fetches = 0
            self.wrapped = aClass(*args, **kargs)    # Use enclosing scope name
        def __getattr__(self, attrname):
```

```
            print('Trace: ' + attrname)              # Catches all but own attrs
            self.fetches += 1
            return getattr(self.wrapped, attrname)   # Delegate to wrapped obj
    return Wrapper

@Tracer
class Spam:                                          # Spam = Tracer(Spam)
    def display(self):                               # Spam is rebound to Wrapper
        print('Spam!' * 8)

@Tracer
class Person:                                        # Person = Tracer(Person)
    def __init__(self, name, hours, rate):           # Wrapper remembers Person
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):                                   # Accesses outside class traced
        return self.hours * self.rate                # In-method accesses not traced

food = Spam()                                        # Triggers Wrapper()
food.display()                                       # Triggers __getattr__
print([food.fetches])

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

print('')
sue = Person('Sue', rate=100, hours=60)              # sue is a different Wrapper
print(sue.name)                                      # with a different Person
print(sue.pay())

print(bob.name)                                      # bob has different state
print(bob.pay())
print([bob.fetches, sue.fetches])                    # Wrapper attrs not traced
```

这里与我们前面在“编写函数装饰器”一节中遇到的跟踪器装饰器有很大不同，注意到这点很重要，在那里，我们看到了装饰器可以使我们跟踪和计时对一个给定函数或方法的调用。相反，通过拦截实例创建调用，这里的类装饰器允许我们跟踪整个对象接口，例如，对其任何属性的访问。

下面是这段代码在Python 2.6和Python 3.0下的输出：Spam和Person类的实例上的属性获取都会调用Wrapper类中的__getattr__逻辑，由于food和bob确实都是Wrapper的实例，得益于装饰器的实例创建调用重定向：

```
Trace: display
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
[1]
Trace: name
Bob
Trace: pay
2000
```

```
Trace: name
Sue
Trace: pay
6000
Trace: name
Bob
Trace: pay
2000
[4, 2]
```

注意，前面的代码装饰了一个用户定义的类。就像是在本书第30章最初的例子中一样，我们也可以使用装饰器来包装一个内置的类型，例如列表，只要我们的子类允许装饰器语法，或者手动地执行装饰——装饰器语法对于@行需要一条class语句。

在下面的代码中，由于装饰的间接作用，x实际是一个Wrapper（我把装饰器类放到了模块文件*tracer.py*中，以便以这种方式重用它）：

```
>>> from tracer import Tracer           # Decorator moved to a module file

>>> @Tracer
... class MyList(list): pass           # MyList = Tracer(MyList)

>>> x = MyList([1, 2, 3])              # Triggers Wrapper()
>>> x.append(4)                        # Triggers __getattr__, append
Trace: append
>>> x.wrapped
[1, 2, 3, 4]

>>> WrapList = Tracer(list)            # Or perform decoration manually
>>> x = WrapList([4, 5, 6])            # Else subclass statement required
>>> x.append(7)
Trace: append
>>> x.wrapped
[4, 5, 6, 7]
```

这种装饰器方法允许我们把实例创建移动到装饰器自身之中，而不是要求传入一个预先生成的对象。尽管这好像是一个细小的差别，它允许我们保留常规的实例创建语法并且通常实现装饰器的所有优点。我们只需要用装饰器语法来扩展类，而不是要求所有的实例创建调用都通过一个包装器来手动地指向对象：

```
@Tracer                                         # Decorator approach
class Person: ...
bob = Person('Bob', 40, 50)
sue = Person('Sue', rate=100, hours=60)

class Person: ...                               # Non-decorator approach
bob = Wrapper(Person('Bob', 40, 50))
sue = Wrapper(Person('Sue', rate=100, hours=60))
```

假设你将会产生类的多个实例，装饰器通常将会在代码大小和代码可维护性上双赢。

注意： 属性版本差异：正如我们在本书第37章所了解到的，在Python 2.6中，__getattr__将会拦截对__str__和__repr__这样的运算符重载方法的访问，但是，在Python 3.0中不会这样。

在Python 3.0中，类实例会从类中继承这些方法中的一些（而不是全部）的默认形式（实际上，是从自动对象超类），因为所有的类都是“新式的”。此外，在Python 3.0中，针对打印和+这样的内置操作显式地调用属性并**不会**通过__getattr__（或其近亲__getattribute__）路由。新式类在类中查找这样的方法，并且完全省略常规的实例查找。

此处意味着，在Python 2.6中，基于__getattr__的跟踪包装器将会自动跟踪和传递运算符重载，但是，在Python 3.0中不会如此。要看到这一点，直接在交互式会话的前面的末尾显示“x”，在Python 2.6中，属性__repr__被跟踪并且该列表如预期的那样打印出来，但是在Python 3.0中，不会发生跟踪并且列表打印为Wrapper类使用一个默认显示：

```
>>> x                                    # 2.6
Trace: __repr__
[4, 5, 6, 7]
>>> x                                    # 3.0
<tracer.Wrapper object at 0x026C07D0>
```

要在Python 3.0中同样工作，运算符重载方法通常需要在包装类中冗余地重新定义，要么手动定义，要么通过工具定义，或者通过在超类中定义。只有简单命名的属性会在两种版本中都同样工作。我们将在本章稍后的一个Private装饰器中再次看到版本差异的作用。

类错误之二：保持多个实例

令人好奇的是，这个例子中的装饰器函数**几乎**可以编写为一个类而不是一个函数，使用正确的运算符重载协议。如下的略微简化的替代版本类似地工作，因为当@装饰器应用于类的时候，触发__init__，并且当创建了一个主体类实例的时候触发其__call__。这次，我们的对象实际是Tracer的实例，并且这里，我们实际上只是为避免使用对一个实例属性的封闭作用域引用：

```
class Tracer:
    def __init__(self, aClass):                 # On @decorator
        self.aClass = aClass                    # Use instance attribute
    def __call__(self, *args):                  # On instance creation
        self.wrapped = self.aClass(*args)       # ONE (LAST) INSTANCE PER CLASS!
        return self
    def __getattr__(self, attrname):
        print('Trace: ' + attrname)
        return getattr(self.wrapped, attrname)

@Tracer                                         # Triggers __init__
class Spam:                                     # Like: Spam = Tracer(Spam)
    def display(self):
        print('Spam!' * 8)

...
```

```
food = Spam()                                   # Triggers __call__
food.display()                                  # Triggers __getattr__
```

正如我们在前面见到的，这个仅针对类的替代方法像前面一样处理多个类，但是，它对于一个给定的类的**多个实例**并不是很有效：每个实例构建调用会触发`__call__`，这会覆盖前面的实例。直接效果是`Tracer`只保存了一个实例，即最后创建的一个实例。自行体验一下看看这是如何发生的，但是，这里给出该问题的一个示例：

```
@Tracer
class Person:                           # Person = Tracer(Person)
    def __init__(self, name):           # Wrapper bound to Person
        self.name = name

bob = Person('Bob')                     # bob is really a Wrapper
print(bob.name)                         # Wrapper embeds a Person
Sue = Person('Sue')
print(sue.name)                         # sue overwrites bob
print(bob.name)                         # OOPS: now bob's name is 'Sue'!
```

这段代码输出如下——由于这个跟踪器只有一个共享的实例，所以第二个实例覆盖了第一个实例：

```
Trace: name
Bob
Trace: name
Sue
Trace: name
Sue
```

这里的问题是一个糟糕的**状态保持**——我们为每个类创建了一个装饰器实例，但是不是针对每个类实例，这样一来，只有最后一个实例保持住了。其解决方案就像我们在前面针对装饰方法的类错误一样，在于放弃基于类的装饰器。

前面的基于函数的`Tracer`版本确实可用于多个实例，因为每个实例构建调用都会创建一个新的`Wrapper`实例，而不是覆盖一个单个的共享的`Tracer`实例的状态。由于同样的原因，最初的非装饰器版本正确地处理多个实例。装饰器不仅仅具有无可争辩的魔力，而且微妙的程度令人难以置信。

装饰器与管理器函数的关系

无顾这样的微妙性，`Tracer`类装饰器示例最终仍然是依赖于`__getattr__`来拦截对一个包装和嵌入实例对象的获取。正如我们在前面见到的，我们真正需要完成的只是，把实例创建调用移入一个类的内部，而不是把实例传递一个管理器函数。对于最初的非装饰器跟踪实例，我们将直接有差异地编写实例创建：

```
class Spam:                                  # Non-decorator version
    ...                                      # Any class will do
food = Wrapper(Spam())                       # Special creation syntax

@Tracer
class Spam:                                  # Decorator version
    ...                                      # Requires @ syntax at class
food = Spam()                                # Normal creation syntax
```

基本上，**类装饰器**把特殊语法需求从实例创建调用迁移到了类语句自身。对于本节前面的单体示例来说也是如此，我们直接将类及其构建参数传递到了一个管理器函数中，而不是装饰一个类并使用常规的实例创建调用：

```
instances = {}
def getInstance(aClass, *args):
    if aClass not in instances:
        instances[aClass] = aClass(*args)
    return instances[aClass]

bob = getInstance(Person, 'Bob', 40, 10)        # Versus: bob = Person('Bob', 40, 10)
```

作为替代方案，我们可以使用Python的内省工具来从一个已准备创建好的实例来获取类（假设创建一个初始化实例是可以接受的）：

```
instances = {}
def getInstance(object):
    aClass = object.__class__
    if aClass not in instances:
        instances[aClass] = object
    return instances[aClass]

bob = getInstance(Person('Bob', 40, 10))        # Versus: bob = Person('Bob', 40, 10)
```

这对于我们前面所编写的跟踪器那样的**函数装饰器**也是成立的：我们可以直接把函数及其参数传递到负责分配调用的一个管理器中，而不是用拦截随后调用的逻辑来装饰一个函数：

```
def func(x, y):                              # Nondecorator version
    ...                                      # def tracer(func, args): ... func(*args)
result = tracer(func, (1, 2))                # Special call syntax

@tracer
def func(x, y):                              # Decorator version
    ...                                      # Rebinds name: func = tracer(func)
result = func(1, 2)                          # Normal call syntax
```

像这样的管理器函数的方法把使用特殊语法的负担放到了调用上，而不是期待在函数和类定义上使用装饰语法。

为什么使用装饰器（重访）

那么，为什么我们只是展示不使用装饰器的方法来实现单体呢？正如我在本章开始的时候提到的，装饰器展示给我们利弊权衡。尽管语法意义重大，当面对新工具的时候，我们通常都忘了问“为什么要用”的问题。既然已经看到了装饰器实际是如何工作的，让我们花点时间在这里看看更大的问题。

就像大多数语言功能一样，装饰器也有优点和缺点。例如，从负面的角度讲，**类装饰器**有两个潜在的缺陷：

类型修改

正如我们所见到的，当插入包装器的时候，一个装饰器函数或类不会保持其**最初的类型**——其名称重新绑定到一个包装器对象，在使用对象名称或测试对象类型的程序中，这可能会很重要。在单体的例子中，装饰器和管理函数的方法都为实例保持了最初的类类型；在跟踪器的代码中，没有一种方法这么做，因为需要有包装器。

额外调用

通过装饰添加一个包装层，在每次调用装饰对象的时候，会引发一次**额外调用**所需的额外性能成本——调用是相对耗费时间的操作，因此，装饰包装器可能会使程序变慢。在跟踪器代码中，两种方法都需要每个属性通过一个包装器层来指向；单体的示例通过保持最初的类类型而避免了额外调用。

类似的问题也适用于**函数装饰器**：装饰和管理器函数都会导致额外调用，并且当装饰的时候通常会发生类型变化（不装饰的时候就没有）。

也就是说，这二者都不是非常严重的问题。对于大多数程序来说，类型差异问题不可能有关系，并且额外调用对速度的影响也不显著；此外，只有当使用包装器的时候才会产生后一个问题，且这个问题常常可以忽略，因为需要优化性能的时候可以直接删除装饰器，并且添加包装逻辑的非装饰器解决方案也会导致额外调用的问题（包括我们将在第39章学习的**元类**）。

相反，正如我们在本章开始所见到的，装饰器有3个主要优点。与前面小节的管理器（即辅助）函数解决方案相比，装饰器提供：

明确的语法

装饰器使得扩展明确而显然。它们的@比可能在源文件中任何地方出现的特殊代码要容易识别，例如，在单体和跟踪器实例中，装饰器行似乎比额外代码更容易被注意到。此外，装饰器允许函数和实例创建调用使用所有Python程序员所熟悉的常规语法。

代码可维护性

装饰器避免了在每个函数或类调用中重复扩展的代码。由于它们只出现一次，在类或者函数自身的定义中，它们排除了冗余性并简化了未来的代码维护。对于我们的单体和跟踪器示例，要使用管理器函数的方法，我们需要在每次调用的时候使用特殊的代码——最初以及未来必须做出的任何修改都需要额外的工作。

一致性

装饰器使得程序员忘记使用必需的包装逻辑的可能性大大减少。这主要得益于两个优点——由于装饰是显式的并且只出现一次，出现在装饰的对象自身中，与必须包含在每次调用中的特殊代码相比较，装饰器促进了更加一致和统一的API使用。例如，在单体示例中，可能更容易忘了通过特殊代码来执行所有类创建调用，而这将会破坏单体的一致性管理。

装饰器还促进了代码的**封装**以减少冗余性，并使得未来的维护代价最小化。尽管其他的编码结构化工具也能做到这些，但装饰器使得这对于扩展任务来说更自然。

然而，这三个优点还不是使用装饰器语法的必需的原因，装饰器的用法最终还是一个格式选择。也就是说，大多数程序员发现了一个纯粹的好处，特别是它作为正确使用库和API的一个工具。

我还记得类中的构造函数函数的支持者和反对者也有过类似的争论——在介绍`__init__`方法之前，创建它的时候通过一个方法手动地运行一个实例，往往也能实现同样的效果（例如，`X=Class().init()`）。然而，随着时间的流逝，尽管这基本上是一个格式的选择，但`__init__`语法也变成了广泛的首选，因为它更为明确、一致和可维护。尽管这应该由你来决定，但装饰器似乎把很多同样的成功摆到了桌面上。

直接管理函数和类

本章中，我们的大多数示例都设计来拦截函数和实例创建调用。尽管这对于装饰器来说很典型，它们并不限于这一角色。因为装饰器通过装饰器代码来运行新的函数和类，从而有效地工作，它们也可以用来管理函数和类对象自身，而不只是管理对它们随后的调用。

例如，假设你需要被另一个应用程序使用的方法或类注册到一个API，以便随后处理（可能该API随后将会调用该对象，以响应事件）。尽管你可能提供一个注册函数，在对象定义之后手动地调用该函数，但装饰器使得你的意图更为明显。

这一思路如下的简单实现定义了一个装饰器，它既应用于函数也应用于类，把对象添加

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
```

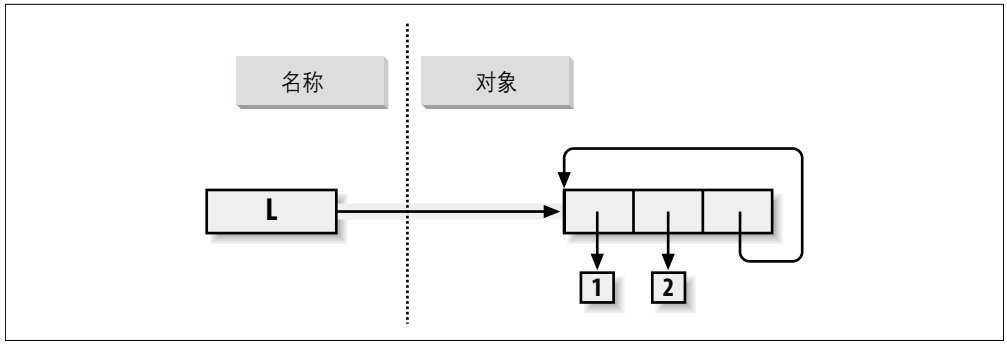
6. *中断和循环*。当你输入以下代码的时候：

```
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()
```