第 1 章

文本

引言

感谢：Fred L. Drake, Jr.，PythonLabs

对于脚本语言来说，文本处理任务构成了一个重要的组成部分，每个人都会同意文本处理非常有用。每个人都会有一些文本需要重新格式化或者转化为另一种形式。问题是，每个程序都与另一个程序有点不同，无论它们是多么相似，想提取出一些可复用的代码片段并用它来处理不同的文件格式仍然是非常困难的。

什么是文本

看起来问题有点简单得过分了，事实上，我们看到了文本，就知道了什么是文本，文本是一串字符，这正是它与二进制数据之间的不同。二进制数据是一串字节。

不幸的是，所有的数据进入程序中都只是一串字节。没有什么库函数能够帮助我们确定某一个特定的字节串是否代表文本，我们只能自己创造一些试探的方法来判断那些数据是否能够用文本的方式来处理（不一定正确）。第 1.11 节就展示了一种试探的方法。

Python 的字符串是一串不可改变的字节或者字符。绝大多数我们创造的方法以及处理字符串的方式都是把它们当做一串字符来处理的，但是一些字节串也是可以处理的。Unicode 字符串是一串不可改变的 Unicode 字符：由 Unicode 字符串转到普通字符串或者由普通字符串转化为 Unicode 字符串的过程是通过 codecs（编码器-解码器）对象来完成的，在这些对象涉及的很多标准转化方法中，字符串可以被表示为一串字节（也被称为编码和字符集）。但是需要注意的是，Unicode 字符串不像字节串那样可以身兼两职。第 1.20 节、第 1.21 节和第 1.22 节展示了 Python 中 Unicode 的一些基本方法。

现在假设我们的程序通过上下文环境确认了它要处理的是文本。程序一般会接收外部输入，这通常是最好的方法。我们能够识别该文件的原因是，它有个已知的名字和已经定义好了的格式（在 UNIX 世界这很普遍），或者它有个著名的文件扩展名来指示出

它内容的格式（在 Windows 世界这很普遍）。现在有个问题：我们必须使用“格式”这种东西，要不然上面这段话毫无意义。人们不是认为文本很简单吗？

让我们面对这个问题：其实并没有所谓的“纯”文本这样的东西，即使有，我们也可能不会关心（也有例外，计算机语言学家在某些情况下可能会研究纯粹的文本）。我们的程序想处理的东西是包含在文本中的信息。我们关心的文本可能包含了配置命令、控制命令、流程定义命令、供人类阅读的文档、甚至制表信息。包含配置数据和一系列命令的文本通常可以通过严格的语法检查来验证，然后才能决定是否可以依赖其中的信息。向用户提示输入文本中的错误还远远不够，这也不是我们要讨论的主题。

供人类阅读的文档似乎比较简单，但仍然有很多细节需要注意。由于它们通常被写为自然语言的形式，它们的语法和句法很难检查。不同的文本可能会使用不同的字符集和编码，如果事先不知道相关的编码信息，想检查出一段文本使用了何种字符集和编码是极其困难的，甚至是不可能的。然而，对于自然语言文档的正确呈现，这却是非常必要的。自然语言文本也有结构，但是这种结构并不明显而且还需要你至少懂一点该种自然语言。字符组成了单词，单词再形成了句子，然后句子构成了段落，但仍然有更大的结构。单独的一个段落很难定位，除非你知道文档的排版约定：究竟是每行构成一个段，还是多行组成一个段？如果是后者，我们怎么判断哪些行构成了一段？段之间可能会被空白行、缩进或者其他的特殊符号隔开。第 19.10 节就给出了一个读取文本文件中由空白行隔开段落的例子。

制表信息就像自然语言文本一样也有很多值得讨论的地方，它实际上给输入的格式增加了第二个维度：文本不再是线性的，也不再是一串字符，而是一个字符的矩阵，每一个单独的文本块可以被缩进和组织起来。

基本的文本操作

就像其他的数据格式，我们也需要用一些不同的方式在不同的场合下处理文本。总地来说，有三种基本的操作：

- 解析数据并将数据放入程序内部的结构中；
- 将数据以某种方式转化为另一种相似的形式，数据本身发生了改变；
- 生成全新的数据。

有很多方式可以完成解析，一些特别的解析器可以有效处理有诸多限制的数据格式，并可以解析大量其他的格式。比如 RFC 2822 型的邮件头格式（参看 Python 标准库的 rfc822 模块）和由 ConfigParser 模块处理的配置文件格式。netrc 模块则提供了一种解析应用特有格式的解析器例子，这个模块基于 shlex 模块。shlex 提供一个典型的分词器（tokenizer），可应用于一些基本的语言，特别适合用来创建可读的配置文件以及让用户

在一个具有交互和提示能力的环境下输入命令。Python 标准库中有很多类似的特别解析器，关于它们的使用示例和方法请参看第 2 章和第 13 章。Python 中还有一些正式的解析工具，它们依赖于一些庞大的附加包，可以阅读第 16 章的引言部分以获得一个大致的了解。

当我们提到文本的时候，我们的第一个念头往往是，所谓文本处理，有时候就是文本格式转换。在本章中，我们将会看到一些可以应用于各种场合的转换方法。有时我们要处理的文本是存储于外部文件中的，有时我们则直接处理内存中的字符串。

通过 Python 的 print 语句，文件对象或者类文件对象的 write 方法，我们可以轻易地产生基于程序特定结构的文本数据、对于那些将输出文件作为一个传入参数的应用，我们通常是用该程序的一个方法或者函数来完成此功能。举个例子，函数可以像下面这样使用 print 语句：

```
print >>thefile, sometext
thefile.write(sometext)
```

这就将产生的输出写入到了文件。不过，这并不是通常我们所认为的文本处理，因为在这个过程中并没有文本输入。本书有很多使用 print 和 write 的例子，在进一步的阅读中你会看到更多的示例。

文本的来源

当文本处于内存的时候，如果数据量不是很大，处理起来相对比较容易。对文本的搜索可以轻易和快速地跨越多行，而且不用担心搜索会超出缓存的边界。只要我们能设法让文本处于内存，并以一种普通字符串的形式呈现，就可以充分利用 string 对象的各种内建的操作来简单而快速地处理文本。

基于文件的转化则需要一些特别的对待，因为需要考虑 I/O 性能和可观的开销，以及实际上需要放入内存的数据量。当处理位于磁盘上的数据时，由于数据的尺寸，我们常常避免将整个文件载入内存：把一个 80MB 的文件全部放入内存并不是一件随随便便的事情！当我们的程序只需要处理一部分数据时，尽量让它只处理较小的数据段往往能够得到可观的性能提升，因为这样我们可以让程序拥有更多的资源来运行。相对于涉及大量磁盘读写的大块数据处理，使用谨慎的缓存管理通常能够取得更好的效果。与文件相关的内容请参看第 2 章。

另一个有趣的文本来源是网络。通过 socket 可以从网络中取回文本。我们可以把 socket 看成是一个文件（使用 socket 对象的 makefile 方法），从 socket 中取回的数据可能是完整的一块，也可能是不完整的数据块，这时我们可以继续等待直到更多的数据到达。由于在最后的数据块到达之前文本数据可能是不完整的，所以用 makefile 创建的文件对象可能不太适合被传递给文本处理代码。在处理来自网络连接的文本时，在进一步的处理之前，我们通常需要完全读取连接中的所有数据。如果数据很庞大，我们可以

将其存入一个文件，每当有新数据部分到达，就在文件末尾不断添加，直到最后接收完所有数据，然后再将这个文件用于文本处理。如果要求文本处理这一步要在接收完所有数据之前启动，则需要在具体的处理上采用更加精巧的方法。这方面的解析器例子可以参考标准库中的 htmllib 和 HTMLParser 模块。

字符串基础

Python 提供的用于文本处理的最主要的工具就是字符串——不可改变的字符序列。实际上存在两种字符串：普通字符串，包含了 8 位（ASCII）字符；Unicode 字符串，包含了 Unicode 字符。我们这里不对 Unicode 字符串做太多讨论：它们的处理方法和普通字符串很类似，只不过它们每个字符占用 2（或者 4）个字节，所以它们拥有成千上万（甚至上亿）个不同的字符，而普通字符串却仅有 256 个不同字符。当需要处理一些有不同字母表的文本时，尤其是亚洲的象形文字，Unicode 字符串的价值就体现出来了。而普通字符串对于英文以及一些非亚洲的精简语言已经够用了。比如，所有的欧洲字母表都可以用普通字符串表示，我们采取的典型的方法是使用国际标准编码 ISO-8859-1（或者 ISO-8859-15，如果需要欧洲的货币符号）。

在 Python 中，可以用下列方式表现一个文本字符串：

```
'this is a literal string'
"this is another string"
```

字符串的值被单引号或者双引号圈起。这两种表示法在程序中完全一样，都允许你在字符串内部把另一种表示法的引用符号包括进来，而无须用反斜线符号进行转义：

```
'isn\'t that grand'
"isn't that grand"
```

为了让一个文本字符串扩展到多行，可以在一行的末尾使用反斜线符号，那意味着下面的一行仍是上面字符串的延续：

```
big = "This is a long string\
that spans two lines."
```

如果想让字符串的输出分为两行，可以在字符串中嵌入换行符：

```
big = "This is a long string\n\
that prints on two lines."
```

还有一种方式是用一对连续的三引用符将字符串圈起：

```
bigger = """
This is an even
bigger string that
spans three lines.
"""
```

使用这种三引用符，无须在文本中加入续行符和换行符，因为文本将按照原貌被储存

在 Python 的字符串对象中。也可以在字符串前面加一个 r 或者 R，表示该字符串是一个真正的“原”字符串，需要它的原貌：

```
big = r"This is a long string\
with a backslash and a newline in it"
```

使用“原”字符串，反斜线转义被完全忽略了。还可以在字符串前面加一个 u 或者 U 来使之成为一个 Unicode 字符串：

```
hello = u'Hello\u0020World'
```

字符串是无法改变的，那意味着无论你对它进行什么操作，你总是创建了一个新的字符串对象，而不是改变了原有的字符串。字符串是字符的序列，所以也可以通过索引的方法访问单个字符：

```
mystr = "my string"
mystr[0]       # 'm'
mystr[-2]      # 'n'
```

也可以用切片的方法访问字符串的一个部分：

```
mystr[1:4]     # 'y s'
mystr[3:]      # 'string'
mystr[-3:]     # 'ing'
```

切片的方法还可以扩展，增加第三个参数，作为切片的步长：

```
mystr[:3:-1]    # 'gnirt'
mystr[1::2]     # 'ysrn'
```

可以通过循环遍历整个字符串：

```
for c in mystr:
```

上述方法将 c 依次绑定到了 mystr 中的每一个字符。还可以构建另一个序列：

```
list(mystr)     # 返回 ['m','y',' ','s','t','r','i','n','g']
```

通过简单的加法，还可以实现字符串的拼接：

```
mystr+'oid'     # 'my stringoid'
```

乘法则完成了对字符串多次重复：

```
'xo'*3         # 'xoxoxo'
```

总之，可以对字符串做任何你能够对其他序列所做的操作，前提是不能试图改变字符序列，因为字符串是不能改变的。

字符串对象有很多有用的方法。比如，可以用 s.isdigit()来测试字符串的内容，如果 s 不是空的而且所有的字符都是数字，该方法将会返回 true（否则返回 false）。还可以用 s.upper()来创建一个修改过的字符串，该字符串很像原字符串 s，不过其中的每一个字符都是原对应字符的大写形式。可以用 haystack.count('needle')在一个字符串中搜索另一

个字符串，该方法返回了子串‘needle’在字符串 haystack 中出现的次数。如果有一个庞大的包含多行文本的字符串，可以用 splitlines 来将其分隔为多个单行字符串并置入一个列表中：

```
list_of_lines = one_large_string.splitlines( )
```

然后还可以用 join 来重新生成一个庞大的单个字符串：

```
one_large_string = '\n'.join(list_of_lines)
```

本章展示了字符串对象的许多方法。可以在 Python 的 *Library Reference* 和 *Python in a Nutshell* 中读到更多的相关内容。

Python 中的字符串还可以通过 re 模块用正则表达式来操作。正则表达式是一种强大无比（但也很复杂）的工具，你可能已经通过其他语言（比如 Perl）、使用 vi 编辑器、或者使用命令行方式的工具（比如 grep）对它有所了解了。在本章的下半章中你会看到一些使用正则表达式的例子。更多相关文档，请参考 *Library Reference* 和 *Python in a Nutshell*。如果想掌握这方面的知识，J.E.F. Friedl 的 *Mastering Regular Expressions*（O’Reilly）也是值得推荐的读本。Python 的正则表达式和 Perl 的正则表达式基本相同，Friedl 的书对这方面的内容介绍得非常全面。

Python 的标准模块 string 提供的很多功能和字符串方法提供的功能基本相同，后者被打包成一系列函数，而非方法。string 模块还提供一些附加的功能，比如 string.maketrans 函数，本章中有几节展示了其用法；还有一些有用的字符串常量（比如，string.digits，值为’0123456789’），再比如 Python 2.4 中引入的新类 Template，提供了一种简单而灵活的方式来格式化带有内嵌变量的字符串，你将在本章中看到其应用。字符串格式化操作符，%，提供了一种简洁的方法将字符串拼接并根据某些对象（比如浮点数）来精确格式化字符串。在本章中你同样会看到大量相关例子并学会如何根据自己的目的来使用%。Python 还有一些标准模块或者扩展模块，可处理一些特定种类的字符串。本章并未覆盖这样的特殊领域，不过本书第 12 章将全面而深入地讨论 XML 处理方面的内容。

1.1 每次处理一个字符

感谢：Luther Blissett

任务

用每次处理一个字符的方式处理字符串。

解决方案

可以创建一个列表，列表的子项是字符串的字符（意思是每个子项是一个字符串，长度为一。Python 实际上并没有一个特别的类型来对应“字符”并以此和字符串区分开

来）。我们可以调用内建的 list，用字符串作为参数，如下：

```
thelist = list(thestring)
```

也可以不创建一个列表，而直接用 for 语句完成对该字符串的循环遍历：

```
for c in thestring:
    do_something_with(c)
```

或者用列表推导中的 for 来遍历：

```
results = [do_something_with(c) for c in thestring]
```

再或者，和列表推导效果完全一样，可以用内建的 map 函数，每次取得一个字符就调用一次处理函数：

```
results = map(do_something, thestring)
```

讨论

在 Python 中，字符就是长度为 1 的字符串。可以循环遍历一个字符串，依次访问它的每个字符。也可以用 map 来实现差不多的功能，只要你愿意每取到一个字符就调用一次处理函数。还可以用内建的 list 类型来获得该字符串的所有长度为 1 的子串列表（该字符串的字符）。如果想获得的是该字符串的所有字符的集合，还可以调用 sets.Set，并将该字符串作为参数（在 Python 2.4 中，可以用同样的方式直接调用内建的 set）：

```
import sets
magic_chars = sets.Set('abracadabra')
poppins_chars = sets.Set('supercalifragilisticexpialidocious')
print ''.join(magic_chars & poppins_chars)   # 集合的交集
acrd
```

更多资料

Library Reference 中关于序列的章节；*Perl Cookbook*，1.5 节。

1.2 字符和字符值之间的转换

感谢：Luther Blissett

任务

将一个字符转化为相应的 ASCII（ISO）或者 Unicode 码，或者反其道而行之。

解决方案

这正是内建的函数 ord 和 chr 擅长的任务：

```
>>> print ord('a')
97
>>> print chr(97)
a
```

内建函数 ord 同样也接收长度为 1 的 Unicode 字符串作为参数，此时它返回一个 Unicode 的码值，最大到 65535。如果想把一个数字的 Unicode 码值转化为一个长度为 1 的 Unicode 字符串，可以用内建函数 unichr：

```
>>> print ord(u'\u2020')
8224
>>> print repr(unichr(8224))
u'\u2020'
```

讨论

这是个很普通的任务，有时我们需要将字符（在 Python 中是长度为 1 的字符串）转换为 ASCII 或者 Unicode 码，有时则反其道而行之，很常见也很有用。内建的 ord、chr 和 unichr 函数完全满足了相关的需求。但请注意，新手们常常混淆 chr(n)和 str(n)之间的区别：

```
>>> print repr(chr(97))
'a'
>>> print repr(str(97))
'97'
```

chr 将一个小整数作为参数并返回对应于 ASCII 单字符的字符串，而 str，能够以任何整数为参数，返回一个该整数的文本形式的字符串。

如果想把一个字符串转化为一个包含各个字符的值的列表，可以像下面这样同时使用内建的 map 和 ord 函数：

```
>>> print map(ord, 'ciao')
[99, 105, 97, 111]
```

若想通过一个包含了字符值的列表创建字符串，可以使用".join、map 和 chr；比如：

```
>>> print ''.join(map(chr, range(97, 100)))
abc
```

更多资料

参见 *Library Reference* 中的内建函数 chr，ord 和 unichr 有关章节以及 *Python in a Nutshell*。

1.3 测试一个对象是否是类字符串

感谢：Luther Blissett

任务

有时候需要测试一个对象，尤其是当你在写一个函数或者方法的时候，经常需要测试传入的参数是否是一个字符串（或者更准确地说，这个对象是否具有类似于字符串的行为模式）。

解决方案

下面给出一个利用内建的 isinstance 和 basestring 来简单快速地检查某个对象是否是字符串或者 Unicode 对象的方法，如下：

```
def isAString(anobj):
      return isinstance(anobj, basestring)
```

讨论

很多遇到这个问题的程序员第一反应是进行类型测试：

```
def isExactlyAString(anobj):
      return type(anobj) is type('')
```

然而，这种方法非常糟糕，因为它破坏了 Python 强大力量的源泉——平滑的、基于签名的多态机制。很明显 Unicode 对象无法通过这个测试，用户自己编写的 str 的子类也不行，甚至任何一种行为表现类似于字符串的用户自定义类型的实例都无法通过测试。

本节推荐的内建函数 isinstance 则要好很多。内建类型 basestring 的存在使得这个方法成为可能。basestring 是 str 和 unicode 类型的共同基类，任何类字符串的用户自定义类型都应该从基类 basestring 派生，这样能保证 isinstance 的测试按照预期工作。本质上 basestring 是一个“空”的类型，就像 object，所以从它派生子类并没有什么开销。

不幸的是，这个似乎完美的 isinstance 检查方案，对于 Python 标准库中的 UserString 模块提供的 UserString 类的实例，完全无能为力。而 UserString 对象是非常明显的类字符串对象，只不过它不是从 basestring 派生的。如果想支持这种类型，可以直接检查一个对象的行为是否真的像字符串一样，比如：

```
def isStringLike(anobj):
      try: anobj + ''
      except: return False
      else: return True
```

这个 isStringLike 函数比方案中给出的 isAString 函数慢且复杂得多，但它的确适用于 UserString（以及其他的类字符串的类型）的实例，也适用于 str 和 unicode。

Python 中通常的类型检查方法是所谓的鸭子判断法：如果它走路像鸭子，叫声也像鸭子，那么对于我们的应用而言，就可以认为它是鸭子了。IsStringLike 函数只不过检查了叫声部分，那其实还不够。如果需要检查 anobj 对象的更多的类字符串特征，可以改造 try 子句，让它检查更多细节，比如：

```
try: anobj.lower( ) + anobj + ''
```

根据我的经验，isStringLike 函数的测试通常就已经满足需要了。

进行类型验证（或者任何验证任务）的最具 Python 特色的方法是根据自己的预期去执行任务，在此过程中检测并处理由于不匹配产生的所有错误和异常。这是一个著名的处理方式，叫做“获得事后原谅总是比事先得到许可要容易得多（It's easier to ask forgiveness than permission）”，或简称 EAFP。try/except 是保证 EAFP 处理风格的关键工具。有时，像本节中的例子一样，可以选择一个简单的判断方法，比如拼接一个空字符串，作为对一系列属性的集合（字符串对象提供的各种操作和方法）的一个替代性判断。

更多资料

参见 *Library Reference* 中内建的 isinstance 和 basestring 的文档以及 *Python in a Nutshell*。

1.4 字符串对齐

感谢：Luther Blissett

任务

实现字符串对齐：左对齐，居中对齐，或者右对齐。

解决方案

这正是 string 对象的 ljust、rjust 和 center 方法要解决的问题。每个方法都需要一个参数，指出生成的字符串的宽度，之后返回一个在左端、右端、或者两端都添加了空格的字符串拷贝：

```
>>> print '|', 'hej'.ljust(20), '|', 'hej'.rjust(20), '|', 'hej'.center(20), '|'
| hej            |                  hej | hej       |
```

讨论

我们常常能够碰到居中、左对齐或右对齐的文本——比如，你可能会打印一个简单的报告，并以 monospace 字体居中显示页码。正因为这种需求很常见，Python 的 string 对象提供了三个简单好用的方法。在 Python 2.3 中，填充字符只能是空格。在 Python 2.4 中，默认情况下仍然使用空格，但是可以给这三种方法第二个参数，指定一个填充字符：

```
>>> print 'hej'.center(20, '+')
++++++++hej+++++++++
```

更多资料

Library Reference 有关 string 的方法；Java Cookbook，第 3.5 节。

1.5 去除字符串两端的空格

感谢：Luther Blissett

任务

获得一个开头和末尾都没有多余空格的字符串。

解决方案

字符串对象的 lstrip、rstrip 和 strip 方法正是为这种任务而设计的。这几个方法都不需要参数，它们会直接返回一个删除了开头、末尾或者两端的空格的原字符串的拷贝：

```
>>> x = '   hej   '
>>> print '|', x.lstrip( ), '|', x.rstrip( ), '|', x.strip( ), '|'
| hej    |    hej | hej |
```

讨论

有时候需要给字符串添加一些空格，让其符合预先规定的固定宽度，以完成左右对齐或居中对齐（如前面 1.4 节所介绍的），但有时也需要从两端移除所有的空格（空白、制表符、换行符等）。因为这种需求是如此常见，Python 的字符串对象给出了 3 个方法来提供这种功能。也可以选择去除其他字符，只需提供一个字符串作为这 3 种方法的参数即可：

```
>>> x = 'xyxxyy hejyx  yyx'
>>> print '|'+x.strip('xy')+'|'
| hejyx  |
```

注意，上面例子中最后获得的字符串的开头和结尾的空格都被保留下来，因为“yx”后面接着的是一些空格：只有开头和结尾的“x”和“y”被真正移除了。

更多资料

Library Reference 中关于字符串的方法；第 1.4 节；*Java Cookbook* 3.12。

1.6 合并字符串

感谢：Luther Blissett

任务

有一些小的字符串，想把这些字符串合并成一个大字符串。

解决方案

要把一系列小字符串连接成一个大字符串，可以使用字符串操作符 join。假如 pieces 是一个字符串列表，想把列表中所有的字符串按顺序拼接成一个大字符串，可以这么做：

```
largeString = ''.join(pieces)
```

如果想把存储在一些变量中的字符串片段拼接起来，那么使用字符串格式化操作符%会更好一些：

```
largeString = '%s%s something %s yet more' % (small1, small2, small3)
```

讨论

Python 中，+操作符也能够将字符串拼接起来，从而实现类似的功能。假如有一些保存在变量中的字符串片段，使用下面这种代码似乎是一种很自然的方式：

```
largeString = small1 + small2 + ' something ' + small3 + ' yet more'
```

类似地，如果有一个小字符串序列，假设叫做 pieces，那么很自然地，可以像这样编写代码：

```
largeString = ''
for piece in pieces:
     largeString += piece
```

或者，用完全等同但却更加漂亮和紧凑的方式：

```
import operator
largeString = reduce(operator.add, pieces, '')
```

不过，不要认为上述例子中给出的方法已经足够好了，上面给出的方法都有许多值得推敲的地方。

Python 中的字符串对象是无法改变的。任何对字符串的操作，包括字符串拼接，都将产生一个新的字符串对象，而不是修改原有的对象。因此拼接 N 个字符串将涉及创建并丢弃 N-1 个中间结果。当然，不创建中间结果的操作会有更佳的性能，但往往不能一步到位地取得最终结果。

如果有少量字符串（尤其是那些绑定到变量上的）需要拼接，甚至有时还需要添加一些额外的信息，Python 的字符串格式化操作符%通常是更好的选择。性能对这种操作完全不是一个问题。和使用多个+操作符相比，%操作符还有一些其他的潜在优点。一旦习惯了它，%也会让你的代码的可读性更好。也无须再对所有的非字符串（如数字）部分调用 str，因为格式指定符%s 已经暗中做完了这些工作。另一个优点是，还可以使用除%s 之外的其他格式指定符，这样可以实现更多的格式要求，比如将浮点数转化为字符串的表示时，可以控制它的有效位数。

什么是“序列”？

Python 并没有一个特别的类型叫做 sequence，但序列是 Python 中一个非常常用的术语。序列，严格地讲，是一个可以迭代的容器，可以从中取出一定数目的子项，也可以一次取出一个，而且它还支持索引、切片，还可以传递给内建函数 len（返回容器中子项的数目）。Python 的 list 就是“序列”，你已经见过多次了，但还有很多其他的“序列”（string、unicode 对象、tuple、array.array 等）。

通常，一个对象即使不支持索引、切片和 len。只要具有一次获得一项的迭代能力，对应用而言就已经够用了。这叫做可迭代对象（或者，如果我们把范围限定在拥有有限子项的情况下，那就叫做有边界可迭代对象）。可迭代对象不是序列，而是如字典（迭代操作将以任意顺序每次取得一个 key）、文件对象（迭代操作将给出文本文件的行数）等，还有其他一些，包括迭代器和生成器等。任何可迭代对象都能用在 for 循环语句以及一些等价的环境中（Python 2.4 的生成器表达式、列表推导中的 for 子句、以及很多内建的方法，比如 min、max、zip、sum、str.join 等）。

在 http://www.python.org/moin/PythonGlossary，可以发现一个词汇表，Python Glossary，它能够帮助你了解很多术语。虽然本书的编辑尽量严格按照那个词汇表的描述来使用术语，仍可能发现本书在很多地方提到了序列、可迭代对象、甚至列表，实际上，严格地讲，我们都应该指明为有边界的可迭代对象。比如，在本节的开头，我们说“一个小字符串的序列”，实际上那是一个有边界的字符串的可迭代对象。在全书使用“有边界的可迭代对象”这样的术语给人的感觉像是在读一本数学书，而不是一本实践性很强的编程书！所以我们决定采用略微偏离严格术语系统的词，这样有助于获得更好的可读性，同时也能够更好地体现本书的多元化。最后结果还不错，根据上下文环境，那些不怎么严密的术语的真实含义仍然能够被清晰地表达出来。

当一个序列中包含了很多的小字符串的时候，性能就变成了一个很现实的问题。在内部使用了+或者+=（和内建函数 reduce 作用相同，但是更漂亮）的循环所需要的时间跟需要累加的字符数的平方成正比，因为分配空间并填充一个大字符串所需要的时间大致正比于该字符串的长度。幸好 Python 提供了另一个更好的选择。对于字符串对象 s 的 join 方法，我们可以传入一个字符串序列作为其参数，它将返回一个由字符串序列中所有子项字符串拼接而成的大字符串，而且这个过程中只使用了一个 s 的拷贝用于串接所有的子项。举个例子，''.join(pieces)把 pieces 中所有的子项一口吞下，而无须产生子项之间的中间结果，再比如，', '.join(pieces)拼接了所有的子项字符串，并在邻接的两项之间插入了一个逗号和空格。这是一种快速、整洁、优雅且兼具良好可读性的合并大字符串的方法。

但有时并不是所有的数据在一开始就已经就位，比如数据可能来自于输入或计算，这

时可以使用一个 list 作为中间数据结构来容纳它们（可以使用 list 的 append 或 extend 方法在末尾添加新的数据）。在取得了所有的数据之后，再调用".join(thelist)就可以得到合并之后的大字符串。在我能教给你的 Python 的字符串处理的各种技巧和方法中，这是最重要的一条：很多 Python 程序效能低下的原因是由于它们使用了+和+=来创建大字符串。因此，一定要提醒自己永远不要使用那种做法，而应该使用本节推荐的".join 方法。

Python 2.4 在这个问题的改善上做了很多工作，在 Python 2.4 中使用+=，性能损失要比以前的版本小一些。但".join 仍然要快许多，而且在各方面都更具优势，至少对于新人和粗心的开发人员来讲，它消耗的时钟周期也更少。类似地，psyco（一种特制的 just-in-time[JIT] Python 编译器，请查看 http://psyco.sourceforge.net/）能更大幅度地减少+=带来的性能损失。不过，我还是要强调，’’.join 依然是最值得选择的方式。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于字符串的方法、字符串格式化操作以及 operator 模块的章节。

1.7 将字符串逐字符或逐词反转

感谢：Alex Martelli

任务

把字符串逐字符或逐词反转过来。

解决方案

字符串无法改变，所以，反转一个字符串需要创建一个拷贝。最简单的方法是使用一种“步长”为-1 的特别的切片方法，这样可立即产生一个完全反转的效果：

```
revchars = astring[::-1]
```

如果要按照单词来反转字符串，我们需要先创建一个单词的列表，将这个列表反转，最后再用 join 方法将其合并，并在相邻两词之间都插入一个空格：

```
revwords = astring.split( )       # 字符串->单词列表
revwords.reverse( )               # 反转列表
revwords = ' '.join(revwords)     # 单词列表->字符串
```

或者，如果喜欢简练而紧凑的“一行解决”的代码：

```
revwords = ' '.join(astring.split( )[::-1])
```

如果想逐词反转但又同时不改变原先的空格，可以用正则表达式来分隔原字符串：

```
import re
revwords = re.split(r'(\s+)', astring)          # 切割字符串为单词列表
revwords.reverse( )                             # 反转列表
revwords = ''.join(revwords)                    # 单词列表 -> 字符串
```

注意，最后的 join 操作要使用空字符串，因为空格分隔符已经被保存在 revwords 列表中了（通过 re.split，使用了一个带括弧的组的正则表达式）。当然，也可以写成“一行解决”的形式，只要你乐意：

```
revwords = ''.join(re.split(r'(\s+)', astring)[::-1])
```

不过这样显得过于紧凑，也失去了可读性，不是好的 Python 代码。

讨论

在 Python 2.4 中，可以改写那个“一行解决”的逐词反转的代码，使用新的内建函数 reversed 来替代原先的可读性略差的切片指示符[::-1]：

```
revwords = ' '.join(reversed(astring.split( )))
revwords = ''.join(reversed(re.split(r'(\s+)', astring)))
```

至于逐字符反转，astring[::-1]仍然是最好的方式，即使在 Python 2.4 中，因为如果要使用 reversed，还得调用".join：

```
revchars = ''.join(reversed(astring))
```

新的内建函数 reversed 返回一个迭代器（iterator），该对象可以被用于循环或者传递给其他的“累加器”，比如".join，但它并不是一个已经完成的字符串。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于序列的切片和类型，以及内建的 reversed（Python 2.4）的内容；*Perl Cookbook* 1.6。

1.8 检查字符串中是否包含某字符集合中的字符

感谢：Jürgen Hermann、Horst Hansen

任务

检查字符串中是否出现了某字符集合中的字符。

解决方案

最简单的方法如下，兼具清晰、快速、通用（适用于任何序列，不仅仅是字符串，也适用于任何容器，不仅仅是集合）：

```
def containsAny(seq, aset):
    """ 检查序列 seq 是否含有 aset 中的项 """
    for c in seq:
        if c in aset: return True
    return False
```

也可以使用更高级和更复杂的基于标准库 itertools 模块的方法来提高一点性能，不过它们本质上其实是同一种方法：

```
import itertools
def containsAny(seq, aset):
    for item in itertools.ifilter(aset.__contains__, seq):
        return True
    return False
```

讨论

对于大多数涉及集合的问题，我们最好使用 Python 2.4 中引入的内建类型 set（如果还在使用 Python 2.3，可以使用 Python 标准库中的等价的 sets.Set 类型）。然而，总是有例外的情况。比如，一个纯粹的基于集合的方法应该是像这样的：

```
def containsAny(seq, aset):
    return bool(set(aset).intersection(seq))
```

不过这种方法就意味着 seq 中的每个成员都不可避免地要被检查。而本节方法中给出的函数，从某种角度讲，可以被叫做“短路法”：它一知道答案就迅速返回。如果答案是 False，它必须检查 seq 中的每个子项——因为除非检查所有的子项，否则我们无法确保 seq 中不含有 aset 中的元素。不过如果答案是 True，我们通常可以很快知道，因为只要找到一个子项是 aset 的成员就可以了。当然这通常是依赖于数据的。如果 seq 很短或者答案是 False，实际上并没有多大区别，但如果 seq 很长，用哪种方式来检查就变得极其关键了（尤其是答案是 True 而且又可以很快探知结果的情况）。

本节给出的 containsAny 的第一个版本有着简洁和清晰的优点：它直观地表达了它背后蕴藏的思想。而第二个版本则可能显得有点“聪明”，这个词在 Python 的世界中可不是一个正面的表达赞美的形容词，因为简洁和清晰才是这个世界的核心价值。不过，第二个版本也有值得思考的地方，因为它展示了一种高级的、基于标准库的 itertools 模块的方法。大多数情况下高级方法总是比低级方法更好（当然在本节的这个特别的例子中，情况正好相反）。itertools.ifilter 要求传入一个断定（译者注：Predicate，原意为断言、谓词或述词）和一个可迭代对象，然后筛选出可迭代对象中的满足该“断定”的描述的所有子项。这里，所谓的“断定”，我们用的是 anyset._ _contains_ _，当我们编写 in anyset 这样的代码来做检查的时候，其内部调用的就是 anyset._ _contains_ _。所以，如果 ifilter 找到了什么东西，比如它找出一个 seq 的子项，同时也正是 anyset 的成员，函数就会立刻返回 True。如果程序运行到了 for 语句之后，那肯定表示 return

True 根本没有被执行，因为 seq 中的任何子项都不是 anyset 的成员，此时只能返回 False。

> 什么是“断定”？
>
> 在一些编程的讨论中你常常可以看到这个词（predicate）：意为一个返回 True 或 False 的函数（或者其他可调用对象）。如果它返回 True，我们就称这个断定成立。

如果你的程序需要用到像 containsAny 这样的函数来检查一个字符串（或其他序列）是否包含了某个集合的成员，也可能会写出这样的变种：

```
def containsOnly(seq, aset):
    """ 检查序列 seq 是否含有 aset 中的项 """
    for c in seq:
          if c not in aset: return False
    return True
```

containsOnly 和 containsAny 完全一样，只不过正好逻辑颠倒了一下。下面还有一个类似的例子，完全没办法短路（必须检查所有的子项），我们最好使用于内建的 set 类型（Python 2.4；或 Python 2.3 中的 sets.Set，使用方法一样）：

```
def containsAll(seq, aset):
     """ 检查序列 seq 是否含有 aset 的所有的项 """
     return not set(aset).difference(seq)
```

如果不习惯用 set（或 sets.Set）的方法 difference，可以记住它的语义：任何一个 set 对象 a，a.difference(b)（就像 a-set(b)）返回 a 中所有不属于 b 的元素。比如：

```
>>> L1 = [1, 2, 3, 3]
>>> L2 = [1, 2, 3, 4]
>>> set(L1).difference(L2)
set([ ])
>>> set(L2).difference(L1)
set([4])
```

希望这样的结果可以解释得更清楚：

```
>>> containsAll(L1, L2)
False
>>> containsAll(L2, L1)
True
```

另一方面，不要把 difference 和 set 的其他方法搞混了，比如 symmetric_difference，它返回的集合包含了所有属于其中一个集合且不属于另一个集合的元素。

如果需要处理 seq 和 aset 中的字符串（非 Unicode），可能不需要本节中这些通用的函数，而可以尝试更加特殊的方式，如第 1.10 节中的方法，基于字符串的方法 translate 和 Python 标准库中的 string.maketrans 函数：

```
import string
notrans = string.maketrans('', '')          # identity "translation"
def containsAny(astr, strset):
     return len(strset) != len(strset.translate(notrans, astr))
def containsAll(astr, strset):
     return not strset.translate(notrans, astr)
```

这看起来有点诡异的方法主要依赖于这个事实：strset.translate(notrans, astr)是 strset 的子序列，而且是由所有不属于 astr 的字符组成的。如果这个子序列和 strset 有同样的长度，说明 strset.translate 没有删除任何字符，因此 strset 中任何一个字符都不属于 astr。相反，如果子序列是空，说明所有的字符都被移除了，所以所有属于 strset 的字符也属于 astr。当程序员们把字符串当做字符集合的时候，很自然地就会想到使用 translate 办法，因为这个方法速度不错，而且灵活易用，更多细节参看第 1.10 节。

不过本节中的两种方式的通用性完全不同。早先提出的方式通用性非常好：并不局限于字符串处理，它对你要处理的对象类型的要求也更少。而基于 translate 方法的方式则相反，它要求 astr 和 strset 都是普通字符串，或者在行为和功能上要与普通字符串非常相似。甚至连 Unicode 字符串都不行，因为 Unicode 字符串的 translate 方法的签名不同于普通字符串对应的 translate 版本——Unicode 版本只需要一个参数（该参数是一个把码值映射到 Unicode 字符串或者 None 的 dict 对象），普通字符串版本则需要两个（必须都是普通字符串）。

更多资料

1.10 节；*Library Reference* 和 *Python in a Nutshell* 中关于普通字符串和 Unicode 字符串的 translate 方法的文档，string 模块的 maketrans 函数的内容；同上阅读材料中的内建 set 对象，sets 和 itertools 模块，以及特殊的_ _contains_ _方法相关内容。

1.9 简化字符串的 translate 方法的使用

感谢：Chris Perkins、Raymond Hettinger

任务

用字符串的 translate 方法来进行快速编码，但却发现很难记住这个方法和 string.maketrans 函数的应用细节，所以需要对它们做个简单的封装，以简化其使用流程。

解决方案

字符串的 translate 方法非常强大而灵活，具体细节可参考第 1.10 节。正因为它的威力和灵活性，将它“包装”起来以简化应用就成了个好主意。一个返回闭包的工厂函数可以很好地完成这种任务：

```
import string
def translator(frm='', to='', delete='', keep=None):
     if len(to) == 1:
          to = to * len(frm)
     trans = string.maketrans(frm, to)
     if keep is not None:
          allchars = string.maketrans('', '')
          delete = allchars.translate(allchars, keep.translate(allchars,
delete))
     def translate(s):
          return s.translate(trans, delete)
     return translate
```

讨论

我经常发现我有使用字符串的 translate 方法的需求，但每次我都得停下来回想它的用法细节（见第 1.10 节提供的更多细节信息）。所以，我干脆给自己写了个类（后来改写成了本节中展示的工厂闭包的形式），把各种可能性封闭在一个简单易用的接口后面。现在，如果我需要一个函数来选出属于指定集合的字符，我就可以简单地创建并使用它：

```
>>> digits_only = translator(keep=string.digits)
>>> digits_only('Chris Perkins : 224-7992')
'2247992'
```

移除属于某字符集合的元素也同样简单：

```
>>> no_digits = translator(delete=string.digits)
>>> no_digits('Chris Perkins : 224-7992')
'Chris Perkins : -'
```

甚至，我可以用某个字符替换属于某指定集合的字符：

```
>>> digits_to_hash = translator(from=string.digits, to='#')
>>> digits_to_hash('Chris Perkins : 224-7992')
'Chris Perkins : ###-####'
```

虽然后面那个应用显得有点特殊，但我仍然不时地碰到有这种需求的任务。

当然，我的设计有点武断，当 delete 参数和 keep 参数有重叠部分的时候，我让 delete 参数优先：

```
>>> trans = translator(delete='abcd', keep='cdef')
>>> trans('abcdefg')
'ef'
```

对于你的程序，如果 keep 被指定了，可能忽略掉 delete 会更好一些，再或者，如果两者都被指定了，抛出个异常也不错，因为在一个对 translator 的调用中同时指定两者可能没什么意义。另外，和第 1.8 节和第 1.10 节相似，本节代码只适用于普通字符串，对 Unicode 字符串并不适用。参看第 1.10 节，可以了解到怎样为 Unicode 字符串

编写类似功能的代码，并可看到 Unicode 的 translate 方法与普通（单字节）字符串的 translate 的区别。

> 闭包
>
> 闭包（closure）不是什么复杂得不得了的东西：它只不过是个“内层”的函数，由一个名字（变量）来指代，而这个名字（变量）对于“外层”包含它的函数而言，是本地变量。我们用一个教科书般的例子来说明：
>
> ```
> def make_adder(addend):
>     def adder(augend): return augend+addend
>     return adder
> ```
>
> 执行 p = make_addr(23)将产生内层函数 adder 的一个闭包，这个闭包在内部引用了名字 addend，而 addend 又绑定到数值 23。q = make_adder(42)又产生另一个闭包，这次名字 addend 则绑定到了值 42。q 和 p 相互之间并无关联，因此它们可以相互独立地和谐共存。现在我们就可以执行它们了，比如，print p(100), q(100)将打印出 123 142。
>
> 实际上，我们一般认为 make_adder 指向一个闭包，而不是说什么迂腐拗口的“一个返回闭包的函数”——幸运的是，根据上下文环境，通常这样也不至于造成误解。称 make_adder 为一个工厂（或者工厂函数）也是简洁明确的；还可以称它为一个闭包工厂来强调它创建并返回闭包，而不是返回类或者类的实例。

更多资料

参看第 1.10 节中关于本节 translator(keep=…)的一个等价实现，以及该节对 translate 方法的更多描述，还有 Unicode 字符串的对应方案；*Library Reference* 和 *Python in a Nutshell* 中的字符串的 translate 方法的文档，string 模块的 maketrans 函数的相关内容。

1.10 过滤字符串中不属于指定集合的字符

感谢：Jürgen Hermann、Nick Perkins、Peter Cogolo

任务

给定一个需要保留的字符的集合，构建一个过滤函数，并可将其应用于任何字符串 s，函数返回一个 s 的拷贝，该拷贝只包含指定字符集合中的元素。

解决方案

对于此类问题，string 对象的 translate 方法是又快又好用的工具。不过，为了有效地使用 translate 来解决问题，事先我们必须做一些准备工作。传递给 translate 的第一个参数是一个翻译表：在本节中，我们其实不需要什么翻译，所以我们必须准备一个特制的

参数来指明“无须翻译”。第二个参数指出了我们需要删除的字符：这个任务要求我们保留（正好反过来，不是删除）属于某字符集合的字符，所以我们必须为该字符集合准备一个补集，作为第二个参数——这样就可以删除所有我们不想保留的字符。闭包是一次性完成所有准备工作的最好方法，它能够返回一个满足需求的快速过滤函数：

```
import string
# 生成所有字符的可复用的字符串，它还可以作为
# 一个翻译表，指明“无须翻译”
allchars = string.maketrans('', '')
def makefilter(keep):
    """ 返回一个函数，此返回函数接受一个字符串为参数
        并返回字符串的一个部分拷贝，此拷贝只包含在
        keep 中的字符，注意 keep 必须是一个普通字符串
    """
    # 生成一个由所有不在 keep 中的字符组成的字符串：keep 的
    # 补集，即所有我们需要删除的字符
    delchars = allchars.translate(allchars, keep)
    # 生成并返回需要的过滤函数（作为闭包）
    def thefilter(s):
        return s.translate(allchars, delchars)
    return thefilter
if __name__ == '__main__':
    just_vowels = makefilter('aeiouy')
    print just_vowels('four score and seven years ago')
# 输出：ouoeaeeyeaao
    print just_vowels('tiger, tiger burning bright')
# 输出：ieieuii
```

讨论

理解本节技巧的关键在于对 Python 标准库的 string 模块的 maketrans 函数以及字符串对象的 translate 方法的理解。translate 应用于一个字符串并返回该字符串的一个拷贝，这个拷贝中的所有字符都将按照传入的第一个参数（翻译表）指定的替换方式来替换，而且，第二个参数指定的所有字符都将被删除。maketrans 是创建翻译表的一个工具函数。（翻译表是一个正好有 256 个字符的字符串 t：当你把 t 作为第一个参数传递给 translate 方法时，原字符串中的每一个字符 c，在处理之后都被翻译成了字符 t[ord(c)]。）

在本节的技巧中，我们将整个过滤任务分解为准备阶段和执行阶段，使效能达到了最大化。由于包含所有字符的字符串需要重复使用，我们只创建它一次，并在模块导入后将它设置为全局变量。采用这种方式的原因是我们确定每个过滤函数都使用同样的翻译表，因此它非常节省内存。而我们要传递给 translate 的第二个参数——需要删除的字符，则依赖于需要保留的字符集合，因为它完全是后者的“补集”：我们需要通知 translate 删除我们不想保留的字符。所以，我们用 makefilter 工厂函数来创建需要删除

的字符集合（字符串），即通过使用 translate 方法来删除“需要保留的字符”，这一步很快得以完成。和本节的其他函数一样，无论是创建还是执行，translate 的速度都非常快。程序中的执行部分给出的测试代码，则展示了如何通过调用 makefilter 构建一个过滤函数，并给这个过滤函数绑定一个名字（只需简单地给 makefilter 的返回结果指定个名字即可），然后对一些测试字符串调用该函数并打印出结果。

顺带一提，用 allchars 作为参数调用过滤函数会把所有需要保留的字符处理成一种非常规整的字符串——严格按照字母表排序而且没有重复的字符。可以根据这种思路编写一个很简单的函数，将以字符串形式给出的字符集合处理成规整的形式：

```
def canonicform(s):
    """ 给定字符串 s，将 s 的字符以一种规整的字符串形式返回:
        按照字母顺序排列且没有重复 """
    return makefilter(s)(allchars)
```

在“解决方案”小节中给出的代码，使用了 def 语句来建立一个嵌套的函数（闭包），这是因为 def 是最常见、最通用，也是最清晰的创建函数的语句。不过如果你乐意，也可以用 lambda 来代替原来的语句，只需修改 makefilter 函数中的 def 和 return 语句，然后写成只需一行的 return lambda 语句：

```
return lambda s: s.translate(allchars, delchars)
```

大多数 Python 玩家认为，相对于 lambda，def 更清晰且更具可读性。

既然本节处理的一些字符串可以被看作字符集合，也可以用 sets.Set 类型（或 Python 2.4 中的内建 set 类型）来完成相同的任务。但得益于 translate 方法的威力和速度，在类似的这种直接处理字符串的任务中，使用 translate 总是比通过 set 来实现要快一些。不过，正如第 1.8 节提到的，本节中给出的技巧只适用于普通字符串，对 Unicode 字符串则不适用。

为了能够解决 Unicode 字符串的问题，我们需要做完全不同的准备工作。Unicode 字符串的 translate 方法只需要一个参数：一个序列或者映射，并且根据字符串中的每个字符的码值进行索引。码值不是映射的键（或者序列的索引值）的字符会被直接复制，不做改变。与每个字符码对应的值必须是一个 Unicode 字符串（该字符的替换物）或者 None（这意味着该字符需要被删除）。这种用法看上去既优雅又强大，可惜对于普通字符串却不适用，所以我们还得重写代码。

通常，我们使用 dict 或 list 作为 Unicode 字符串的 translate 方法的参数，来翻译或者删除某些字符。但由于本节任务有些特殊（保留一些字符，删掉所有其余字符），我们可能会需要一个非常庞大的 dict 或 string——但仅仅是把所有的其他字符映射到 None。更好的办法是，编写一个简单的大致实现了 __getitem__（进行索引操作时会调用的特殊方法）方法的类。一旦我们花点功夫完成了这个小类，我们可以让这个类的实例可被调用，还可以直接给这个类起个 makefilter 的别名：

```
import sets
class Keeper(object):
      def __init__(self, keep):
            self.keep = sets.Set(map(ord, keep))
      def __getitem__(self, n):
            if n not in self.keep:
                  return None
            return unichr(n)
      def __call__(self, s):
            return unicode(s).translate(self)
makefilter = Keeper
if __name__== '__main__':
      just_vowels = makefilter('aeiouy')
      print just_vowels(u'four score and seven years ago')
# 输出: ouoeaeeyeaao
      print just_vowels(u'tiger, tiger burning bright')
# 输出: ieieuii
```

我们也可以直接就把这个类命名为 makefilter，但是，基于传统，一个类的名字通常应该首字母大写；遵循传统一般来说没有什么害处，所以，代码就成了这个样子。

更多资料

第 1.8 节；*Library Reference* 和 *Python in a Nutshell* 中普通字符串和 Unicode 字符串的 translate 方法的有关内容，以及 string 模块的 maketrans 函数的介绍。

1.11 检查一个字符串是文本还是二进制

感谢：Andrew Dalke

任务

在 Python 中，普通字符串既可以容纳文本，也可以容纳任意的字节，现在需要探知（当然，完全是启发式的试探：对于这个问题并没有什么精准的算法）一个字符串中的数据究竟是文本还是二进制。

解决方案

我们采取 Perl 的判定方法，如果字符串中包含了空值或者其中有超过 30%的字符的高位被置 1（意味着该字符的码值大于 126）或是奇怪的控制码，我们就认为这段数据是二进制数据。我们得自己编写代码，其优点是对于特殊的程序需求，我们随时可以调整这种启发式的探知方式：

```
from __future__ import division          # 确保/不会截断
import string
text_characters = "".join(map(chr, range(32, 127))) + "\n\r\t\b"
```

```
_null_trans = string.maketrans("", "")
def istext(s, text_characters=text_characters, threshold=0.30):
    # 若 s 包含了空值，它不是文本
    if "\0" in s:
        return False
    # 一个“空”字符串是“文本”（这是一个主观但又很合理的选择）
    if not s:
        return True
    # 获得 s 的由非文本字符构成的子串
    t = s.translate(_null_trans, text_characters)
    # 如果不超过 30%的字符是非文本字符，s 是字符串
    return len(t)/len(s) <= threshold
```

讨论

可以轻易地修改函数 istext 的启发式探知部分，只需传递一个指定的阀值作为判断某字符串所含数据是“文本”（即正常的 ASCII 字符加上 4 个“正常”的控制码，在文本中这几个控制码都是有意义的）的基准，默认的阀值是 0.30（30%）。举个例子，如果期望它是采用了 iso-8859-1 的意大利文本，可以给 text_characters 参数添加意大利语中的一些重音字母，“àèéìòù”。

很多时候，需要检查的对象是文件，而不是字符串，也就是说要判断文件中的内容是文本还是二进制数据。同样地，我们仍可采用 Perl 的启发式方法，用前面提供的 istext 函数来检查文件的第一个数据块：

```
def istextfile(filename, blocksize=512, **kwds):
    return istext(open(filename).read(blocksize), **kwds)
```

注意，默认情况下，istext 函数中的 len(t)/len(s)将被截断成 0，因为这是一个整数之间的除法结果。以后的版本（估计是 Python 3.0，几年后发布），Python 中的/操作符的意义会被改变，这样我们在做除法运算的时候就不会发生截断——如果你确实需要截断，可以用截断除法操作符//。

不过，现在 Python 还没有改变除法的语义，这是为了保证一定的向后兼容性。为了让成千上万行的现有的 Python 程序和模块平滑地工作于所有的 Python 2.x 版本，这非常重要。不过，对于语言版本的主版本号的改变，Python 允许进行不考虑向后兼容性的改变。

因此，对于本节的解决方案中的模块，按照未来版本中计划的行为模式来改变除法的行为是非常方便的，我们用这种方式来引入模块：

```
from __future__ import division
```

这条语句并不影响程序的其余部分，只影响紧随此声明的模块；通过这个模块，/表现得像“真实的除法”（没有截断）。对于 Python 2.3 和 2.4，dvision 可能是唯一需要从_ _future_ _导入的模块。其他的一些未来版本中计划的特性，nested_scope 和生成

器，现在已经是语言的一部分了，因而无法被关闭——当然明确导入它们没有什么坏处，但只有在你的程序需要能够运行在老版本的 Python 环境下时，这种做法才有意义。

更多资料

1.10 节中关于 maketrans 函数以及字符串方法 translate 的诸多细节；*Language Reference* 中关于真实除法和截断除法的内容。

1.12 控制大小写

感谢：Luther Blissett

任务

将一个字符串由大写转成小写，或者反其道而行之。

解决方案

这正是字符串对象提供 upper 和 lower 方法的原因。每个方法都不需要参数，直接返回一个字符串的拷贝，其中的每个字母都被改变成大写形式——或小写形式：

```
big = little.upper( )
little = big.lower( )
```

非字母的字符按照原样被复制。

s.capitalize 和 s[:1].upper()+s[1:].lower()相似：第一个字符被改成大写，其余字符被转成小写。s.title 也很相似，不过它将每个单词的第一个字母大写（这里的单词可以是字母的序列），其余部分则转成小写：

```
>>> print 'one tWo thrEe'.capitalize( )
One two three
>>> print 'one tWo thrEe'.title( )
One Two Three
```

讨论

操作字符串大小写是很常见的需求，有很多方法可以让你创建需要的字符串。另外，还可以检查一个字符串是否已经是满足需求的形式，比如 isupper、islower 和 istitle 方法，如果给定的字符串不是空的，至少含有一个字母，而且分别满足全部大写、全部小写、每个单词开头大写的条件，这三种方法都会返回一个 True，但是却没有类似的 iscapitalized 方法。不过如果我们需要一个行为方式类似于“is…”的方法，自己编写代码也很简单。如果给定的字符串是空的，那些方法都会返回 False。如果给定的字符串非空，但是却不包含任何字母字符，也将全部返回 False。

最清楚简单的 iscapitalized，仅需简洁的一行：

```
def iscapitalized(s):
     return s == s.capitalize( )
```

不过，这偏离了“is…”方法们的行为模式，对于空字符串和不含字母的字符串，它也返回 True。我们再给出一个严格点的版本：

```
import string
notrans = string.maketrans('', '') #identity''translation''
def containsAny(str, strset):
    return len(strset) != len(strset.translate(notrans, str))
def iscapitalized(s):
    return s == s.capitalize( ) and containsAny(s, string.letters)
```

这里，我们用了第 1.8 节中的函数来确保，当遇到了空字符串或不含字母的字符串，返回值是 False。不过也正如第 1.8 节的提示一样，那意味着这个特别的 iscapitalized 只适用于普通字符串，对 Unicode 字符串不适用。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于字符串方法的介绍；*Perl Cookbook* 1.9；1.8 节。

1.13 访问子字符串

感谢：Alex Martelli

任务

获取字符串的某个部分。比如，你读取了一条定长的记录，但只想获取这条记录中的某些字段的数据。

解决方案

切片是个好方法，但是它一次只能取得一个字段：

```
afield = theline[3:8]
```

如果还需考虑字段的长度，struct.unpack 可能更适合。比如：

```
import struct
# 得到一个 5 字节的字符串，跳过 3 字节，得到两个 8 字节字符串，以及其余部分:
baseformat = "5s 3x 8s 8s"
# theline 超出的长度也由这个 base-format 确定
#（在本例中是 24 字节，但 struct.calcsize 是很通用的）
numremain = len(theline) - struct.calcsize(baseformat)
# 用合适的 s 或 x 字段完成格式，然后 unpack
format = "%s %ds" % (baseformat, numremain)
l, s1, s2, t = struct.unpack(format, theline)
```

如果想跳过“其余部分”，只需要给出正确的长度，拆解出 theline 的开头部分的数据即可：

```
l, s1, s2 = struct.unpack(baseformat, theline[:struct.calcsize(baseformat)])
```

如果需要获取 5 字节一组的数据，可以利用带列表推导（LC）的切片方法，代码很简单：

```
fivers = [theline[k:k+5] for k in xrange(0, len(theline), 5)]
```

将字符切成一个个单独的字符更加容易：

```
chars = list(theline)
```

如果想把数据切成指定长度的列，用带 LC 的切片方法通常是最简单的：

```
cuts = [8, 14, 20, 26, 30]
pieces = [ theline[i:j] for i, j in zip([0]+cuts, cuts+[None]) ]
```

在 LC 中调用 zip，返回的是一个列表，其中每项都是形如(cuts[k], cuts[k+1])这样的数对，除了第一项和最后一项，这两项分别是(0, cuts[0])和(cuts[len(cuts)-1], None)。换句话说，每一个数对都给出了用于切割的正确的(i, j)，仅有第一项和最后一项例外，前者给出的是切割之前的切片方式，后者给出的是切割完成之后到字符串末尾的剩余部分。LC 利用这些数对就可以正确地将 theline 切分开来。

讨论

本节受到了 *Perl Cookbook* 1.1 的启发。Python 的切片方法，取代了 Perl 的 substr。Perl 的内建的 unpack 和 Python 的 struct.unpack 也非常相似。不过 Perl 的手段更丰富一点，它可以用*来指定最后一个字段长度，并指代剩余部分。在 Python 中，无论是为了获取或者跳过某些数据，我们都得计算和插入正确的长度。不过这不是什么大问题，因为此类抽取字段数据的任务往往可以被封装成小函数。如果该函数需要反复被调用的话，memoizing，通常也被称为自动缓存机制，能够极大地提高性能，因为它避免了为 struct.unpack 反复做一些格式准备工作。参见第 18.5 节中关于 memoizing 的更多细节。

在纯 Python 的环境中，struct.unpack 作为字符串切片的一种替代方案，非常好用（当然不能和 Perl 的 substr 比，虽然它不接受用*指定的区域长度，但仍是值得推荐的好东西）。

这些代码片段，最好被封装成函数。封装的一个优点是，我们不需要每次使用时都计算最后一个区域的长度。下面的函数基本上等价于“解决方案”小节给出的直接使用 struct.unpack 的代码片段：

```
def fields(baseformat, theline, lastfield=False):
    # theline 超出的长度也由这个 base-format 确定
    #（通过 struct.calcsize 计算确切的长度）
    numremain = len(theline)-struct.calcsize(baseformat)
```

```
    # 用合适的 s 或 x 字段完成格式，然后 unpack
    format = "%s %d%s" % (baseformat, numremain, lastfield and "s" or "x")
    return struct.unpack(format, theline)
```

一个值得注意（或者说值得批评）的设计是该函数提供了 lastfield=False 这样一个可选参数。这基于一个经验，虽然我们常常需要跳过最后的长度未知的部分，有时候我们还是需要获取那段数据。采用 lastfield and s or x（等同于 C 语言中的三元运算符，lastfield?"s":"c"）这样的表达式，我们省去了一个 if/else，不过是否需要为这点紧凑牺牲可读性还有值得商榷之处。参看第 18.9 节中有关在 Python 中模拟三元运算符的内容。

若 fields 函数在一个循环内部被调用，使用元组(baseformat, len(theline), lastfield)作为 key 来充分利用 memoizing 机制将极大地提高性能。这里给出一个使用 memoizing 机制的 fields 版本：

```
def fields(baseformat, theline, lastfield=False, _cache={ }):
    # 生成键并尝试获得缓存的格式字符串
    key = baseformat, len(theline), lastfield
    format = _cache.get(key)
    if format is None:
        # 没有缓存的格式字符串，创建并缓存之
        numremain = len(theline)-struct.calcsize(baseformat)
        _cache[key] = format = "%s %d%s" % (
        baseformat, numremain, lastfield and "s" or "x")
    return struct.unpack(format, theline)
```

这种利用缓存的方法，目的是将比较耗时的格式准备工作一次完成，并存储在_cache 字典中。当然，正像所有的优化措施一样，这种采用了缓存机制的优化也需要通过测试来确定究竟能在多大程度上提高性能。对这个例子，我的测试结果是，通过缓存优化的版本要比优化之前快约 30%到 40%，换句话说，如果这个函数不是你的程序的性能瓶颈部分，其实没有什么必要多此一举。

“解决方案中”给出的另一个关于 LC 的代码片段，也可以封装成函数：

```
def split_by(theline, n, lastfield=False):
    # 切割所有需要的片段
    pieces = [theline[k:k+n] for k in xrange(0, len(theline), n)]
    # 若最后一段太短或不需要，丢弃之
    if not lastfield and len(pieces[-1]) < n:
        pieces.pop( )
    return pieces
```

对最后一个代码片段的封装：

```
def split_at(theline, cuts, lastfield=False):
    #切割所有需要的片段
    pieces = [ theline[i:j] for i j in zip([0]+cuts, cuts+[None]) ]
```

```
        # 若不需要最后一段，丢弃之
        if not lastfield:
             pieces.pop( )
        return pieces
```

在上面这些例子中，利用列表推导来切片要比用 struct.unpack 略好一些。

用生成器可以实现一个完全不同的方式，像这样：

```
def split_at(the_line, cuts, lastfield=False):
     last = 0
     for cut in cuts:
          yield the_line[last:cut]
          last = cut
     if lastfield:
          yield the_line[last:]
def split_by(the_line, n, lastfield=False):
     return split_at(the_line, xrange(n, len(the_line), n), lastfield)
```

当需要循环遍历获取的结果序列时，无论是显式调用，还是借助一些可调用的“累加器”，比如”.join 来进行隐式调用，基于生成器的方式都会更加合适。如果需要的是各字段数据的列表，你手上得到的结果却是一个生成器，可以调用内建的 list 来完成转化，像这样：

```
list_of_fields = list(split_by(the_line, 5))
```

更多资料

第 18.9 节和第 18.5 节；*Perl Cookbook* 1.1。

1.14 改变多行文本字符串的缩进

感谢：Tom Good

任务

有个包含多行文本的字符串，需要创建该字符串的一个拷贝，并在每行行首添加或者删除一些空格，以保证每行的缩进都是指定数目的空格数。

解决方案

字符串对象已经提供了趁手的工具，我们只需写个简单的函数即可满足需求：

```
def reindent(s, numSpaces):
     leading_space = numSpaces * ' '
     lines = [ leading_space + line.strip( )
                  for line in s.splitlines( ) ]
     return '\n'.join(lines)
```

讨论

处理文本的时候，我们常常需要改变一块文本的缩进。“解决方案”给出的代码，在多行文本的每行行首增减了空格，这样每行开头都有相同的空格数。比如：

```
>>> x = """  line one
...     line two
...  and line three
... """
>>> print x
  line one
    line two
 and line three
>>> print reindent(x, 4)
    line one
    line two
    and line three
```

即使每行的缩进都截然不同，该函数仍能够使它们的缩进变得完全一致，这有时正是我们所需要的，但有时却不是。一个常见的需求是，调整每行行首的空格数，并确保整块文本的行之间的相对缩进不发生变化。无论是正向还是反向调整，这都不是难事。不过，反向调整需要检查一下每行行首的空格，以确保不会把非空格字符截去。因此，我们需要将这个任务分解，用两个函数来完成转化，再加上一个计算每行行首空格并返回一个列表的函数：

```
def addSpaces(s, numAdd):
     white = " "*numAdd
     return white + white.join(s.splitlines(True))
def numSpaces(s):
     return [len(line)-len(line.lstrip( )) for line in s.splitlines( )]
def delSpaces(s, numDel):
    if numDel > min(numSpaces(s)):
          raise ValueError, "removing more spaces than there are!"
    return '\n'.join([ line[numDel:] for line in s.splitlines( ) ])
```

所有这些函数都依赖字符串的方法 splitlines，它和根据’\n’来切分的 split 很相似。不过 splitlines 还有额外的好处，它保留了每行末尾的换行符（当你传入的参数是 True 的时候）。有时这非常方便：如果 splitlines 这个字符串方法没有提供这个能力，addSpaces 不可能这么短小精悍。

然后，我们用这些函数组合成另一个函数来删除行首空格。该函数可以在保持各行之间的相对缩进不变的情况下，只删除它能够删除的空格，让缩进最小的行与左端边界平齐。

```
def unIndentBlock(s):
     return delSpaces(s, min(numSpaces(s)))
```

更多资料

Library Reference 和 *Python in a Nutshell* 中关于序列类型的部分。

1.15 扩展和压缩制表符

感谢：Alex Martelli、David Ascher

任务

将字符串中的制表符转化成一定数目的空格，或者反其道而行之。

解决方案

将制表符转换为一定数目的空格是一种很常见的需求，用 Python 的字符串提供的 expandtabs 方法可以轻松解决问题。由于字符串不能被改变，这个方法返回的是一个新的字符串对象，是原字符串的一个修改过的拷贝。不过，仍可以将修改过的拷贝绑定到原字符串的名字：

```
mystring = mystring.expandtabs( )
```

这样并不会改变 mystring 原先指向的字符串对象，只不过将名字 mystring 绑定到了一个新创建的修改过的字符串拷贝上了，该字符串拷贝中的制表符也已经被扩展为一些空格了。对 expandtabs 来说，默认情况下，制表符的宽度为 8；可以给 expandtabs 传递一个整数参数来指定新的制表符宽度。

将空格转成制表符则比较少见和怪异。如果真的想要压缩制表符，最好还是用别的办法来解决，因为 Python 没有提供一个内建的方法来“反扩展”空格，将其转化为制表符。当然，我们可以自己写个函数来完成任务。字符串的切分、处理以及重新拼接，往往比对整个字符串反复转换要快得多：

```
def unexpand(astring, tablen=8):
     import re
     # 切分成空格和非空格的序列
     pieces = re.split(r'( +)', astring.expandtabs(tablen))
     # 记录目前的字符串总长度
     lensofar = 0
     for i, piece in enumerate(pieces):
          thislen = len(piece)
          lensofar += thislen
          if piece.isspace( ):
               # 将各个空格序列改成 tabs+spaces
               numblanks = lensofar % tablen
               numtabs = (thislen-numblanks+tablen-1)/tablen
               pieces[i] = '\t'*numtabs + ' '*numblanks
     return ''.join(pieces)
```

例子中的 unexpand 函数，只适用于单行字符串；要处理多行字符串，用’’. Join ([unexpand(s) for s in astring.splitlines(True)])即可。

讨论

虽然在 Python 的字符串操作中，正则表达式从来不是必不可少的部分，但有时它真的很方便。正如代码所示，unexpand 函数利用了 re.split 相对于字符串的 split 额外提供的特性：当正则表达式包含了一个括弧组时，re.split 返回了一个 list 列表，列表中的每两个相邻的分隔片段之间都被插入了一个用于分隔的“分隔器”。这样，我们就得到了一个 pieces 列表，所有的连续空白字符串和非空白字符串都成为了它的子项；接着我们在 for 循环中持续跟踪已处理字符串的长度，并将所有的空白字符片段尽可能地转换成相应的制表符，最后加上必要的剩余空格以保持总体的长度。

对于很多编程任务来说，扩展制表符并不是简简单单地调用 expandtabs 方法。比如，需要整理一些 Python 源代码，这些代码写得不是很规范，不只用空格来控制缩进（只用空格是最好的方式），而是采取了制表符和空格的混合方式（这是非常糟糕的做法）。这实际上加剧了复杂性，比如，需要猜测制表符的宽度（采用标准的 4 个空格的缩进方式是值得强烈推荐的）。另外，你可能还需要保留一些在字符串内部的并非用于控制缩进的制表符（有人可能错误地使用了实际的制表符，而不是“\t”，来指代字符串中间的制表字符），甚至你还可能需要对具有不同意义的文本做不同的处理。在某些情况下，问题还不算棘手，比如，假设只需要处理每行文本行首的空白符，而无须理会其他的制表符。一个像下面这样的运用正则表达式的小函数就足够了：

```
def expand_at_linestart(P, tablen=8):
     import re
     def exp(mo):
          return mo.group( ).expand(tablen)
     return ''.join([ re.sub(r'^\s+', exp, s) for s in P.splitlines(True) ])
```

expand_at_linestart 函数充分利用了 re.sub 函数，re.sub 在一个字符串中搜寻符合其正则表达式描述的片段，每当它找到一个匹配，便将匹配的字符串作为一个参数传递给一个函数，并调用该函数返回一个替换匹配字符串的字符串。为了方便，expand_at_linestart 被设计为可以处理多行文本字符串 P，并对调用 splitlines 的结果使用了列表推导处理，最后’\n’.join 将完成后的各行字符串拼接起来。当然，这样的设计完全也可用于单行文本字符串。

实际的制表符扩展的任务可能会很特殊，比如需要考虑制表符是在一个字符串的中间还是外部，是处于哪种类型的文本中（比如，源代码中的注释文本和代码文本），但不管怎么样，至少都需要做一个断词的工作。另外，也可能需要对待处理的源代码进行一个完全的解析，而不是简单地依赖一些字符串和正则表达式操作。如果你想完成的任务具有这种要求，其工作量会相当可观。可以仔细阅读第 16 章，那一章的内容对初学者有很大帮助。

当你汗流浃背地最终完成了转化任务，一定能深深体会到，不管是写代码还是编辑代码，一定要遵循那种常用的、被推荐的 Python 编码风格：只使用空格、4 个空格缩进一级、不用制表符、在字符串中间的制表符应该用“\t”，而不是实际的制表符。你喜爱的 Python 编辑器也应该被强化支持这些使用习惯，这样在保存 Python 源代码文件时你能得到一个遵守约定的格式；比如，IDLE（Python 所带的免费的集成开发环境）附带的编辑器就完全支持这些约定。最好能够在问题出现之前就配置好你的编辑器，而不是在问题出现之后采取弥补措施。

更多文档

Library Reference 的“序列类型”一节下的字符串 expandtabs 方法；*Perl Cookbook* 1.7；*Library Reference* 和 *Python in a Nutshell* 文档中的 re 模块。

1.16 替换字符串中的子串

感谢：Scott David Daniels

任务

需要一个简单的方法来完成这样一个任务：给定一个字符串，通过查询一个替换字典，将字符串中被标记的子字符串替换掉。

解决方案

下面给出的解决办法既适用于 Python 2.3，也适用于 2.4：

```
def expand(format, d, marker='"', safe=False):
     if safe:
          def lookup(w): return d.get(w, w.join(marker*2))
     else:
          def lookup(w): return d[w]
     parts = format.split(marker)
     parts[1::2] = map(lookup, parts[1::2])
     return ''.join(parts)
if __name__ == '__main__':
     print expand('just "a" test', {'a': 'one'})
# 输出: just one test
```

如果参数 safe 是 False，则默认条件下，字符串中所有被标记的子字符串必须能够在字典 d 中找到，否则，expand 会抛出一个 KeyError 异常并终止执行。当参数 safe 被明确指定为 True 时，如果被标记的子字符串在字典中找不到，则被标记的部分也不会被改变。

讨论

expand 函数代码的主体部分有个很有趣的地方：根据操作是否被要求为安全，它使用

两个不同的嵌套函数（两者有着同样的名字 lookup）中的一个。安全意味着被标记的子字符串应该能够在字典查到，如果查不到，不抛出 KeyError 异常。如果这个函数并不是必须安全的（默认情况下不安全），lookup 根据索引访问字典 d，并在该索引（子字符串）不存在的时候抛出个错误。但如果 lookup 被要求为安全的，它将使用 d 的方法 get，get 返回根据索引能够查到的值，若找不到就返回在两边加上了标记的被查询的子字符串。给 safe 传入 True，表明你宁可看到输出中有标记符也不愿看到异常信息。marker+w+marker 是可以替换 w.join(marker*2)的另一种方式，但我采用后者的原因是，它展示了一种不太简明却很有意思的构造带引号字符串的方法。

不管用哪个版本的 lookup，expand 都会执行切分、修改、拼接——这些在 Python 的字符串处理中最重要的操作。在 expand 中进行修改的部分，使用了指定了步长的列表切片方法。确切地说，expand 访问并重新绑定了 parts 的奇数索引的项，因为这些项正好是原字符串中位于两个标记符之间的部分。因此，它们就是被标记的子字符串，也就是需要在字典中查找的字符串。

本节解决方案给出的 expand 函数接受非常灵活的字符串语法形式，比基于$的 string.Template 更灵活。你如果想让输出字符串包括双引号，也可以指定其他的标记符。当然，函数也没有限制被标记的子串不能是标识符，可以轻松地插入 Python 表达式（d 的_ _getitem_ _方法会执行 eval 操作）或者任意其他占位符。而且，还可以轻易地搞出点有些不同的更有趣的效果，比如：

```
print expand('just "a" ""little"" test', {'a' : 'one', '' : '""'})
```

输出结果是 just one "little" test。高级用户可以定制 Python 2.4 的 string.Template 类，通过继承来实现上述的所有功能，甚至更多其他高级功能。但本节解决方案中的小巧的 expand 函数却更加简洁易用。

更多文档

Library Reference 关于 string.Template（Python 2.4）、序列类型（关于字符串的 split、join 方法以及切片操作）以及字典（关于索引和 get 方法）的文档资料。更多的关于 Python 2.4 中的 string.Template 类的信息，参看第 1.17 节。

1.17 替换字符串中的子串——Python 2.4

感谢：John Nielsen、Lawrence Oluyede、Nick Coghlan

任务

在 Python 2.4 的环境下，你想完成这样的任务：给定一个字符串，通过查询一个字符串替换字典，将字符串中被标记的子字符串替换掉。

解决方案

Python 2.4 提供了一个新的 string.Template 类，可以应用于这个任务。下面给出一段代码以展示怎样使用这个类：

```
import string
# 从字符串生成模板，其中标识符被$标记
new_style = string.Template('this is $thing')
# 给模板的 substitute 方法传入一个字典参数并调用之
print new_style.substitute({'thing':5})      # 输出: this is 5
print new_style.substitute({'thing':'test'}) # 输出: this is test
# 另外，也可以给 substitute 方法传递关键字参数
print new_style.substitute(thing=5)          # 输出: this is 5
print new_style.substitute(thing='test')     # 输出: this is test
```

讨论

Python 2.3 中，用于标记——替换的字符串格式被写为更加繁琐的形式：

```
old_style = 'this is %(thing)s'
```

标识符被放在一对括弧中，括弧前面一个%，后面一个 s。然后，还需要使用%操作符，使用的格式是将需要处理的字符串放在%操作符左边并在右边放上字典：

```
print old_style % {'thing':5}      # emits: this is 5
print old_style % {'thing':'test'} # emits: this is test
```

当然，这样的代码在 Python 2.4 中也可以正常工作。不过，新的 string.Template 提供了一个更简单的替代方法。

当你创建 string.Template 实例时，在字符串格式中，可以用两个美元符（$）来代表$，还可以让那些需要被替换的标识后面直接跟上用于替换的文本或者数字，并用一对花括号（{ }）将它们括起来。下面是一个例子：

```
form_letter = '''Dear $customer,
I hope you are having a great time.
If you do not find Room $room to your satisfaction,
let us know. Please accept this $$5 coupon.
               Sincerely,
               $manager
               ${name}Inn'''
letter_template = string.Template(form_letter)
print letter_template.substitute({'name':'Sleepy', 'customer':'Fred Smith',
                                  'manager':'Barney Mills','room':307,
                                  })
```

上面的代码片段给出下列输出：

```
Dear Fred Smith,
I hope you are having a great time.
If you do not find Room 307 to your satisfaction,
```

```
let us know. Please accept this $5 coupon.
               Sincerely,
               Barney Mills
               SleepyInn
```

有时，为了给 substitute 准备一个字典做参数，最简单的方法是设定一些本地变量，然后将所有这些变量交给 locals()（此函数将创建一个字典，字典的 key 就是本地变量，本地变量的值可通过 key 来访问）：

```
msg = string.Template('the square of $number is $square')
for number in range(10):
     square = number * number
     print msg.substitute(locals( ))
```

另一个简单的办法是使用关键字参数语法而非字典，直接将值传递给 substitute：

```
msg = string.Template('the square of $number is $square')
for i in range(10):
      print msg.substitute(number=i, square=i*i)
```

甚至可以同时传递字典和关键字参数：

```
msg = string.Template('the square of $number is $square')
for number in range(10):
     print msg.substitute(locals( ), square=number*number)
```

为了防止字典的条目和关键字参数显式传递的值发生冲突，关键字参数优先。比如：

```
msg = string.Template('an $adj $msg')
adj = 'interesting'
print msg.substitute(locals( ), msg='message')
# emits an interesting message
```

更多资料

Library Reference 文档中关于 string.Template（2.4）部分，以及内建的 locals 函数的相关信息。

1.18 一次完成多个替换

感谢：Xavier Defrang、Alex Martelli

任务

你想对字符串的某些子串进行替换。

解决方案

正则表达式虽然不易读懂，但有时它的确是最快的方法。re 对象（标准库中的 re 模

块）提供的强大 sub 方法，非常利于进行高效的正则表达式匹配替换。下面给出一个函数，该函数返回一个输入字符串的拷贝，该拷贝中的所有能够在指定字典中找到的子串都被替换为字典中的对应值：

```
import re
def multiple_replace(text, adict):
    rx = re.compile('|'.join(map(re.escape, adict)))
    def one_xlat(match):
        return adict[match.group(0)]
    return rx.sub(one_xlat, text)
```

讨论

本节展示了怎样使用 Python 的标准模块 re 来一次完成多个子串的替换。假设你有个基于字典的字符串的映射关系。字典的 key 就是你想要替换的子串，而字典中 key 的对应值则正是被用来做替代物的字符串。也可以针对字典的键值对应关系，调用字符串方法 replace 来完成替换，它将多次处理和创建原文本的复制，但逻辑却很清晰，速度也不错。不过 re.sub 的回调函数机制可以让处理方式变得更加简单。

首先，我们根据想要匹配的 key 创建一个正则表达式。这个正则表达式形式为 a1|a2|...|aN，由 N 个需要被替换的字符串组成，并被竖线隔开，创建的方法也很简单，如代码所示，一行代码完成。然后，我们不直接给 re.sub 传递用于替换的字符串，而是传入一个回调函数参数。这样，每当遇到一次匹配，re.sub 就会调用该回调函数，并将 re.MatchObject 的实例作为唯一参数传递给该回调函数，并期望着该回调函数返回作为替换物的字符串。在本例中，回调函数在字典中查找匹配的文本，并返回了对应值。

本节展示的函数 multiple_replace，每次被调用时都会重新计算正则表达式并重新定义 one_xlat 辅助函数。但你经常只需要使用同一个固定不变的翻译表来完成很多文本的替换，这种情况下也许会希望只做一次准备工作。出于这种需求，也许会使用下面的基于闭包的方式：

```
import re
def make_xlat(*args, **kwds):
    adict = dict(*args, **kwds)
    rx = re.compile('|'.join(map(re.escape, adict)))
    def one_xlat(match):
        return adict[match.group(0)]
    def xlat(text):
        return rx.sub(one_xlat, text)
    return xlat
```

可以给 make_xlat 函数传递一个字典参数，或者其他的可以传递给内建的 dict 用于创建一个字典的参数组合；make_xlat 返回一个 xlat 闭包，它只需要一个字符串参数 text，并返回 text 的一个拷贝，该拷贝是根据字典给出的翻译表完成了替换之后的结果。

下面给出应用此函数的例子。通常我们可以把这个片段中的示例代码作为本节给出的代码源文件的一部分，这段代码受到前面的 Python 语句的保护不会被执行，除非这个模块被作为主脚本被执行：

```
if __name__ == "__main__":
      text = "Larry Wall is the creator of Perl"
      adict = {
         "Larry Wall" : "Guido van Rossum",
         "creator" : "Benevolent Dictator for Life",
         "Perl" : "Python",
      }
      print multiple_replace(text, adict)
      translate = make_xlat(adict)
      print translate(text)
```

本节中的替换任务常常是基于单词的替换任务，而不是基于任意一个子字符串。通过特殊的 r'\b'序列，正则表达式可以很好地找出单词的开始和结束位置。我们可以修改 multiple_replace 和 make_xlat 中创建和分配正则表达式 rx 的部分，从而完成一些自定义任务：

```
rx = re.compile(r'\b%s\b' % r'\b|\b'.join(map(re.escape, adict)))
```

其余的代码和本节前面给出的一样。但是，这种代码相似性可不是好事：那意味着我们需要很多相似的版本，每个创建正则表达式的部分都有点不同，我们可能会需要做大量的复制粘贴工作，这是代码复用中最糟糕的情况，另外在未来的维护上也增加了麻烦。

编写好代码的一个关键规则是："一次，只做一次！"当我们注意到代码重复的时候，应该能够很快嗅到"不妙"的气味，并对原来的代码进行重构以提高复用性。因此，为了便于定制，我们更需要的是一个类，而不是函数或者闭包。下面给出个例子，我们实现了一个类，功能近似于 make_xlat，但却能够通过子类化和重载进行定制：

```
class make_xlat:
     def __init__ (self, *args, **kwds):
          self.adict = dict(*args, **kwds)
          self.rx = self.make_rx( )
     def make_rx(self):
          return re.compile('|'.join(map(re.escape, self.adict)))
     def one_xlat(self, match):
          return self.adict[match.group(0)]
     def __call__ (self, text):
          return self.rx.sub(self.one_xlat, text)
```

这是对 make_xlt 函数的一个完全的替代：另一方面，我们在这之前展示的代码，由于有 if __name__ == '__main__'来保护，即使 make_xlat 由以前的函数变成了类，也不会有什么问题。函数更加简单快速，但是类的优势是可以通过面向对象的方法——子类化或

重载某些函数，轻易地实现重新定制。为了对单词进行翻译替换，代码可以这样写：

```
    class make_xlat_by_whole_words(make_xlat):
         def make_rx(self):
             return re.compile(r'\b%s\b' % r'\b|\b'.join(map(re.escape,
self.adict)))
```

通过简单的子类化和重载来实现定制化，我们避免了对代码的大量的复制和粘贴，这也是有时我们宁可舍弃更加简单的函数或者闭包不用，而使用面向对象结构的原因。仅仅把相关的功能打包成一个类并不能自动实现需要的定制。要实现高度的可定制性，在把功能划分成独立的类方法时必须具有一定的前瞻性。幸好，你不用逼自己第一次就把代码写得很完美；如果代码中没有能够符合任务需求的内部结构时（在这个例子中，我们通过子类化和选择性重载来复用代码），可以而且也应该对代码进行重构，构建出符合需求的结构。当然需要进行一些合适的测试来确保没有把原有的逻辑破坏掉，与此同时，你也完成了对自己的思想内容的重构。访问 http://www.refactoring.com 可以获得更多关于重构的艺术和实践的信息。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于 re 模块的文档；Refactoring 主页（http://www.refactoring.com）。

1.19 检查字符串中的结束标记

感谢：Michele Simionato

任务

给定一个字符串 s，你想检查 s 中是否含有多个结束标记中的一个。需要一种快捷、优雅的方式，来替换掉 s.endswith(end1)、s.endswith(end2)或 s.endswith(end3)之类的笨重用法。

解决方案

对于类似于本节的问题，itertools.imap 给出了一种快速方便的解决办法：

```
    import itertools
    def anyTrue(predicate, sequence):
          return True in itertools.imap(predicate, sequence)
    def endsWith(s, *endings):
          return anyTrue(s.endswith, endings)
```

讨论

一个典型的 endsWith 应用是打印出当前目录中所有的图片文件：

```
import os
for filename in os.listdir('.'):
      if endsWith(filename, '.jpg', '.jpeg', '.gif'):
          print filename
```

本节解决方案中给出的思想可以很容易地应用到其他类似的检查任务中去。辅助函数 anyTrue 是一个通用而快速的函数，可以给它传入其他的被绑定方法(bound method)作为第一个参数，比如 s.startswith 或 s._ _contains_ _。事实上，不使用辅助函数而直接编码也许更好：

```
if anyTrue(filename.endswith, (".jpg", ".gif", ".png")):
```

我认为它的可读性也没什么问题。

被绑定方法（Bound Method）

如果一个 Python 对象提供一个方法，可以直接获得一个已经绑定到该对象的方法，从而直接使用此方法。（比如，可以将其赋值给别的对象、将它作为一个参数传递、或者在一个函数中直接返回它，等等。）举个例子：

```
L = ['fee', 'fie', 'foo']
x = L.append
```

现在 x 指向了列表对象 L 的一个被绑定方法。调用 x，比如 x('fum')，和调用 L.append('fum')是完全等价的：结果都是对象 L 变成了['fee', 'fie', 'foo', 'fum']。

如果访问的是一个类型或者一个类的方法，而不是一个类型或者类的实例的方法，你得到的是一个非绑定方法，该方法并未“依附”于此类型或者类的任何一个实例：当调用它时，需要提供该类型或类的一个实例作为第一个参数。比如，如果设定 y = list.append，你不能直接调用 y('I')，因为 Python 猜不出你想给哪个列表对象添加一个 I。可以调用 y(L,'I')，这和调用 L.append('I')效果完全一样（只要 isinstance(L,list)成立）。

本节的解决方案和想法来源于 news:comp.lang.python 的一个讨论，并综合和概括了很多人的观点，包括了 Raymond Hettinger、Chris Perkins、Bengt Richter 等。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于 itertools 和字符串方法的内容。

1.20 使用 Unicode 来处理国际化文本

感谢：Holger Krekel

任务

需要处理包含了非 ASCII 字符的文本字符串。

解决方案

可以在一些使用普通的字节串 str 类型的场合，使用 Python 提供的内置的 unicode 类型。用法很简单，只要接受了在字节串和 unicode 字符串之间的显式转换的方式：

```
>>> german_ae = unicode('\xc3\xa4', 'utf8')
```

这里 german_ae 是一个 unicode 字符串，代表了小写的德语元音变音（umlaut，或其他分音符）字符“æ”。根据指定的 UTF-8 编码方式，通过解析单字节字符串'\xc3\xa4'，这段代码创建了一个 unicode 字符串。还有很多其他的编码方式，不过 UTF-8 最常用，因为它是最通用的（UTF-8 可以编码任何 unicode 字符串），而且也和 7 位的 ASCII 字符集兼容（任何 ASCII 单字节字符串，也是正确的 UTF-8 编码字符串）。

一旦跨过这一屏障，生活就变得更美好了！可以像处理普通的 str 字符串那样操纵 unicode 字符串：

```
>>> sentence = "This is a " + german_ae
>>> sentence2 = "Easy!"
>>> para = ". ".join([sentence, sentence2])
```

注意，para 是一个 unicode 字符串，这是因为一个 unicode 字符串和一个字节串之间的操作总会产生一个 unicode 字符串——除非这个操作发生错误并抛出异常：

```
>>> bytestring = '\xc3\xa4'     #某个非 ASCII 字节串
>>> german_ae += bytestring
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in
position 0: ordinal not in range(128)
```

字符’0xc3’不是 7 位 ASCII 编码中的有效字符，Python 也拒绝猜测其编码。所以，在 Python 中使用 unicode 的关键点是，你要随时明确编码是什么。

讨论

如果你遵守一些规范，并且学会处理一些常见的问题，则 Python 中的 unicode 处理是非常简单的事情。这不是说完成一个高效的 Unicode 实现是个简单的任务。不过，正如其他的一些难题一样，无须担心太多：只管使用 Python 的高效的 Unicode 实现就行了。

最重要的一点是，首先要完全接受字节串和 unicode 字符串的差异。正如解决方案小节所示，你经常需要通过一个字节串和一个编码方式显式地创建一个 unicode 字符串。不指定编码方式，字节串基本没有什么意义，除非你很有运气而且碰巧那个字节串是 ASCII 文本。

在 Python 中使用 unicode 字符串的最常见的问题是，你正在处理的文本一部分是 unicode 对象，另一部分则是字节串。Python 会简单地尝试把你的字节串隐式地转换成 unicode。它通常假设那些是 ASCII 编码，如果其中碰巧含有了非 ASCII 字符，它会给你一个 UnicodeDecodeError 的异常。UnicodeDecodeError 异常通知你，你把 Unicode 和字节串

混在了一起，而且 Python 无法（它根本也不会去尝试）猜测你的字节串代表何种文本。

各个 Python 大项目的开发人员们总结出了一些简单的规则，来避免这种运行时的 UnicodeDecodeError 异常，该规则可以被总结为一句话：总是在 IO 动作的关口做转换。下面更深入地解释一下。

- 无论何时，当你的程序接收到了来自“外部”的文本数据（来自网络、文件、或者用户输入等）时，应当立刻创建一个 unicode 对象，找出最适合的编码，如查看 HTTP 头，或者寻找一个合适的转化方法来确定所用的编码方式。
- 无论何时，当你的程序需要向“外部”发送文本数据（发到网络、写入文件、或者输出给用户等）时，应当探察正确的编码，并用那种编码将你的文本转化成字节串。（否则，Python 会尝试把 Unicode 转成 ASCII 字节串，这很有可能发生 UnicodeEncodeError 异常，正好是前面例子中给出 UnicodeDecodeError 的相反情况）。

遵循这两个规则，可以解决绝大多数的 Unicode 问题。如果你仍然遇到了那两种 UnicodeError 之一，应当赶快检查是否忘记了在什么地方创建一个 unicode 对象，或者忘记了把它转化为编码过的字节串，再或者使用了完全不正确的编码方式。（编码错误也有可能来自于用户，或者其他与你的程序进行交互的程序，因为它们没有遵循编码规则或惯例。）

为了将一个 Unicode 字符串转回到编码过的字节串，你通常可以这么做：

```
>>> bytestring = german_ae.decode('latin1')
>>> bytestring
'\xe4'
```

现在，bytestring 是德语中的用’latin1’进行编码的 æ 字符。注意，’\xe4’（Latin1）以及前面展示的’\xc3\xa4’（UTF-8）代表了同样的德语字符，但使用了不同的编码。

至此为止，应该能够了解为什么 Python 拒绝在几百种可能的编码中进行猜测了吧。这是一种很重要的设计选择，基于了 *Zen of Python* 原则中的一条：“在模糊含混面前拒绝猜测。”在任何一个 Python 的交互式 shell 提示符下，输入 import this 语句，你就可以阅读 *Zen of Python* 中的重要原则。

更多资料

Unicode 是一个很宽泛的主题，值得推荐的书有：Unicode: A Primer，Tony Graham 著 (Hungry Minds, Inc.)，更多细节见 http://www.menteith.com/unicode/primer/；以及 Joel Spolsky 写的一篇短小但透彻的文章，“The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses)!”具体可见 http://www.joelonsoftware.com/articles/Unicode.html。另外参阅 *Library Reference* 和 *Python in a Nutshell* 中关于内建 str 和 unicode 类型、unidata 模块和 codecs 模块。

1.21 在 Unicode 和普通字符串之间转换

感谢：David Ascher、Paul Prescod

任务

需要处理一些可能不符合 ASCII 字符集的文本数据。

解决方案

普通字符串可以用多种方式编码成 Unicode 字符串，具体要看你究竟选择了哪种编码：

```
unicodestring = u"Hello world"
# 将 Unicode 转化为普通 Python 字符串: "encode"
utf8string = unicodestring.encode("utf-8")
asciistring = unicodestring.encode("ascii")
isostring = unicodestring.encode("ISO-8859-1")
utf16string = unicodestring.encode("utf-16")
# 将普通 Python 字符串转化为 Unicode: "decode"
plainstring1 = unicode(utf8string, "utf-8")
plainstring2 = unicode(asciistring, "ascii")
plainstring3 = unicode(isostring, "ISO-8859-1")
plainstring4 = unicode(utf16string, "utf-16")
assert plainstring1 == plainstring2 == plainstring3 == plainstring4
```

讨论

如果想处理含有非 ASCII 字符的文本数据，首先要懂一些 Unicode——什么是 unicode、unicode 怎么工作、Python 如何处理 unicode。前一节提供了少量却很重要的指导，本节将在这个话题下继续深入讨论。

不用在完全了解 Unicode 的一切之后，才去处理现实世界中的有关 unicode 的问题，但是一些基本知识却是不可或缺的。首先，需要理解字节和字符之间的区别。在过去的以 ASCII 字符为主体的语言以及环境中，字节和字符被认为是同一种东西。一个字节可以有 256 个不同的值，因此环境被限制为只能处理不超过 256 个不同的字符。而另一方面，Unicode 则支持成千上万的字符，那也意味着每个 unicode 字符占用超过 1 个字节的宽度。因此，首先需要搞清楚字符和字节之间的区别。

标准的 Python 字符串实际上是字节串，这种字符串中的每个字符，长度为 1，实际上就是一个字节。我们还可以用 Python 的标准字符串类型的其他一些术语来称其为 8 位字符串或者普通字符串。在本节中，我们称这种类型的字符串为字节串，以此来提醒你它们的单字节的特点。

Python 的 Unicode 字符是一个抽象的对象，它足够大，能够容纳任何字符，可以同 Python

的长整数进行类比。完全无须担心它的内部表示；只有当你试图将它们传递给一些基于字节处理的函数时——比如文件的 write 方法和网络 socket 的 send 方法，unicode 字符的表示才会成为一个问题。基于这个原因，必须选择以何种方式将这些字符表示成字节。将 unicode 字符串转化成字节串被称为对该字符串编码。同样的，如果从一个文件、socket 或者其他基于字节的对象中载入一个 unicode 字符串，必须对其解码，将其从字节转成字符。

从 unicode 对象转化成字节串有很多方法，这些方法都被称为某种编码。由于一系列历史的、政治的、以及技术上的原因，没有哪种编码是“正确”的。每种编码都有个大小写不敏感的名字，这个名字可以被传递给 encode 和 decode 函数作为参数。下面给出一些应该知道的编码。

- UTF-8 编码可以应用于任何 Unicode 字符。它也向后兼容 ASCII，所以一个纯粹的 ASCII 文件也可以被认为是一个 UTF-8 文件，而只使用 ASCII 字符的 UTF-8 文件，也完全等同于使用这些字符的 ASCII 文件。这个属性使得 UTF-8 具有极好的向后兼容能力，特别是对一些老的 UNIX 工具来说。UTF-8 是迄今为止 UNIX 上最具主导性的编码，同时也是 XML 文档的默认编码。UTF-8 的主要弱点是，对于一些东方的语言文本，它的效率比较低。
- UTF-16 是微软操作系统和 Java 环境喜爱的编码。它对于西方语言效率略低，但对于东方语言却更有效率。UTF-16 有一个变种，被称为 UCS-2。
- ISO-8859 系列的编码是 ASCII 的超集，每种编码都能处理 256 个不同的字符。这些编码不能支持所有的 Unicode 字符；它们只支持一些特定的语系或语言。ISO-8859-1，也以“Latin-1”的名字为人所知，覆盖了大多西欧和非洲的语言，但不包括阿拉伯语。ISO-8859-2，也被称为“Latin-2”，覆盖了很多东欧的语言，比如匈牙利和波兰。ISO-8859-15，现在在欧洲非常流行，基本上它和 ISO-8859-1 一样，但增加了对欧洲货币符号的支持。

如果想对所有的 Unicode 字符编码，你可能需要用 UTF-8。当需要处理别的程序或者输入设备用其他编码创建的数据时，你才可能需要用到其他编码，或者相反，在需要输出数据给你的下游程序或者输出设备，而它们采用的是另一种特定的编码时。第 1.22 节展示了一个例子，这个例子中，可以看到怎样通过程序的标准输出来驱动其他下游程序或设备。

更多资料

Unicode 是一个宽泛的主题，值得推荐的书有：Unicode: A Primer，Tony Graham 著（Hungry Minds, Inc.），更多细节见 http://www.menteith.com/unicode/primer/；以及 Joel Spolsky 写的一篇短小但透彻的文章，“The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses)!”具体可

见 http://www.joelonsoftware.com/articles/Unicode.html。另外参阅 *Library Reference* 和 *Python in a Nutshell* 中关于内建 str 和 unicode 类型、unidata 模块和 codecs 模块；以及第 1.20 节和第 1.22 节。

1.22 在标准输出中打印 Unicode 字符

感谢：David Ascher

任务

你想将 Unicode 字符串打印到标准输出中（比如为了调试），但是这些字符串并不符合默认的编码。

解决方案

通过 Python 标准库中的 codecs 模块，将 sys.stdout 流用转换器包装起来。比如，如果你知道输出会被打印到一个终端，而且该终端以 ISO-8859-1 的编码显式字符，可以这样编写代码：

```
import codecs, sys
sys.stdout = codecs.lookup('iso8859-1')[-1](sys.stdout)
```

讨论

Unicode 涵盖极广，全世界的语言字符都在 Unicode 的表示范围之内，另外，Unicode 字符串的内部表示也与 Unicode 使用者没有关系。一个用于处理字节的文件流，比如 sys.stdout，都有自己的编码。可以通过修改 site 模块改变其默认的编码，该文件流将对新文件使用新编码。不过，这样也需要完全改变你的 Python 安装，而且其他一些程序则可能会被搞乱，它们依然会按照你原先的编码设置工作（一般是典型的 Python 标准编码，ASCII）。因此，这种修改并不值得推荐。

本节的方法则用了一个技巧：将 sys.stdout 绑定到一个使用 Unicode 输入和 ISO-8859-1（也就是 Latin-1）输出的流。这种方法并不改变之前 sys.stdout 上的任何编码，如下面代码所示。首先，我们用一个变量指向原来的基于 ASCII 的 sys.stdout：

```
>>> old = sys.stdout
```

然后，我们可以创建一个 Unicode 字符串，这个字符串通常情况下是不能通过 sys.stdout 输出的：

```
>>> char = u"\N{LATIN SMALL LETTER A WITH DIAERESIS}"
>>> print char
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
```

如果这个操作没有出现错误，那是因为 Python 认为它知道你的“终端”用了什么编码（特别是，如果你的“终端”是 IDLE——Python 所附的免费的开发环境，Python 极有可能能够确认正确的编码）。如果出现了错误，或者没有提示错误，但是输出的字符却不是你期望的，那是因为你的“终端”使用了 UTF-8 编码，而 Python 却不知道。如果属于后者的情况，可以用 codecs 流对 sys.stdout 进行包装以解决 UTF-8 编码问题，将 sys.stdout 绑定到被封装过的流，然后重新试一次：

```
>>> sys.stdout = codecs.lookup('utf-8')[-1](sys.stdout)
>>> print char
ä
```

这个方法只在你的“终端”、终端模拟器或者其他类型的交互式 Python 解释窗口支持 UTF-8 编码时才有效，而且具有极强的字符表现力，能够显示出任何需要的字符。如果没有这样的程序或设备，可以在因特网上找一个适用于你的平台的免费的程序。

Python 会尝试确认你的“终端”的编码，并把编码的名字存在 sys.stdout.encoding 中作为一个属性。有时（但不是总是），它能够判断出正确的编码。IDLE 已经对 sys.stdout 进行了包装，正如本节解决方案的方法一样，所以，在 Python 的交互式环境之下，可以直接打印出 Unicode 字符串。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于 codecs 和 site 模块、以及 sys 模块中的 setdefaultencoding；第 1.20 节和第 1.21 节。

1.23 对 Unicode 数据编码并用于 XML 和 HTML

感谢：David Goodger、Peter Cogolo

任务

你想对 Unicode 文本进行编码，使用一种有限制，但很流行的编码，如 ASCII 或 Latin-1，并将处理后的结果用于 HTML 输出或者某些 XML 应用。

解决方案

Python 提供了一种编码错误处理工具，叫做 xmlcharrefreplace，它会将所有不属于所选编码的字符用 XML 的数字字符引用来替代：

```
def encode_for_xml(unicode_data, encoding='ascii'):
    return unicode_data.encode(encoding, 'xmlcharrefreplace')
```

也可以将此法用于 HTML 输出，不过你可能会更喜欢 HTML 的符号实体引用。出于这个目的，需要定义并注册一个自定义的编码错误处理函数。要实现这样一个处理函

数非常简单，因为 Python 标准库已经提供了一个叫做 htmlentitydefs 的模块，包含了所有的 HTML 实体定义：

```
import codecs
from htmlentitydefs import codepoint2name
def html_replace(exc):
     if isinstance(exc, (UnicodeEncodeError, UnicodeTranslateError)):
          s = [ u'&%s;' % codepoint2name[ord(c)]
                  for c in exc.object[exc.start:exc.end] ]
          return ''.join(s), exc.end
     else:
          raise TypeError("can't handle %s" % exc.__name__)
codecs.register_error('html_replace', html_replace)
```

注册完错误处理函数之后，可以再写个包装函数，以简化使用：

```
def encode_for_html(unicode_data, encoding='ascii'):
     return unicode_data.encode(encoding, 'html_replace')
```

讨论

如同其他的一些 Python 模块一样，这个模块也将提供一个测试的示例，由 if __name__ == '__main__'这一行语句进行保护：

```
if __name__ == '__main__':
     # demo
     data = u'''\
<html>
<head>
<title>Encoding Test</title>
</head>
<body>
<p>accented characters:
<ul>
<li>\xe0 (a + grave)
<li>\xe7 (c + cedilla)
<li>\xe9 (e + acute)
</ul>
<p>symbols:
<ul>
<li>\xa3 (British pound)
<li>\u20ac (Euro)
<li>\u221e (infinity)
</ul>
</body></html>
'''
     print encode_for_xml(data)
     print encode_for_html(data)
```

如果将此模块作为主脚本来运行，你会看到如下的输出（来自于 encode_for_xml）：

```
<li>&#224; (a + grave)
<li>&#231; (c + cedilla)
<li>&#233; (e + acute)
    ...
<li>&#163; (British pound)
<li>&#8364; (Euro)
<li>&#8734; (infinity)
```

还有这些（来自于 encode_for_html）：

```
<li>&agrave; (a + grave)
<li>&ccedil; (c + cedilla)
<li>&eacute; (e + acute)
    ...
<li>&pound; (British pound)
<li>&euro; (Euro)
<li>&infin; (infinity)
```

这两段输出都很清晰，不过 encode_for_xml 更加具有通用性（它可以用于任何 XML 应用，不仅仅是 HTML），但 encode_for_html 却能够生成更易读的结果——如果希望直接读取或者编辑那些结果。如果给浏览器提供这两种形式的数据，能够看到渲染输出实际上是一样的。为了看到这两种方式在浏览器中的展示，可以将上述代码作为主脚本运行，将输出导向到一个磁盘文件，并用文本编辑器将这两种输出分隔开来，然后用浏览器分别查看。（再或者，运行脚本两次，一次将调用 encode_for_xml 的输出注释掉，一次将 encode_for_html 的输出注释掉）。

请记住，Unicode 数据在被打印或者写到文件之前一定要先编码。由于 UTF-8 能够处理任何 Unicode 字符，所以它是理想的编码。但对很多用户和应用而言，ASCII 或 Latin-1 比 UTF-8 更受欢迎。当 Unicode 数据包含了指定编码之外的字符时（比如，一些重音字符和很多符号大多不在 ASCII 或 Latin-1 之中，比如，Latin-1 无法表现“无穷”符号），单靠这些编码本身，根本无法处理这些数据。Python 提供一个内建的编码错误处理函数，叫做 xmlcharrefreplace，将所有的不能被编码的字符替换为 XML 的数字字符引用，比如“∞”，表示“无穷”符号。本节还展示了怎样编写和注册另一个类似的错误处理函数 html_replace，针对 HTML 输出进行处理。html_replace 将不能编码的字符替换为更具可读性的 HTML 符号实体引用，比如用“∞”来表示“无穷”符号。html_replace 相比于 xmlcharrefreplace，它的通用性没那么好，因为它并不支持所有的 Unicode 字符，同时也不能用于非 HTML 的应用；但如果你希望 HTML 输出的源码文件能够具有更好的可读性，html_replace 是非常有用的。

如果输出的不是 HTML 或者任何形式的 XML，这些错误处理函数就没什么意义了。比如，TeX 和其他的标记语言并不认识 XML 的数字字符引用，但如果你知道怎样创建一个该标记语言中的字符引用，可以修改本节示例中的错误处理函数 html_replace，

并注册一个新的符合需求的处理函数。

当需要将 Unicode 数据写入文件时，可以使用 Python 标准库的 codecs 模块提供的另一个（非常高效）可以指定编码和错误处理函数的方法：

```
outfile = codecs.open('out.html', mode='w', encoding='ascii',
                              errors='html_replace')
```

现在，可以将 outfile.write(unicode_data)应用于任何 Unicode 字符串 unicode_data，所有的编码和错误处理都会透明地自动进行。当然，在输出完成之后，还得调用 outfile.close()。

更多文档

Library Reference 和 *Python in a Nutshell* 中关于 codecs 模块和 htmlentitydefs。

1.24 让某些字符串大小写不敏感

感谢：Dale Strickland-Clark、Peter Cogolo、Mark McMahon

任务

你想让某些字符串在比较和查询的时候是大小写不敏感的，但在其他操作中却保持原状。

解决方案

最好的解决方式是，将这种字符串封装在 str 的一个合适的子类中：

```
class iStr(str):
     """
     大小写不敏感的字符串类
     行为方式类似于 str，只是所有的比较和查询
     都是大小写不敏感的
     """
     def __init__ (self, *args):
          self._lowered = str.lower(self)
     def __repr__ (self):
          return '%s(%s)' % (type(self). __name__, str. __repr__(self))
     def __hash__(self):
          return hash(self._lowered)
   def lower(self):
          return self._lowered
def _make_case_insensitive(name):
   ''' 将 str 的方法封装成 iStr 的方法，大小写不敏感 '''
   str_meth = getattr(str, name)
   def x(self, other, *args):
```

```
            ''' 先尝试将 other 小写化，通常这应该是一个字符串，
                但必须要做好准备应对这个过程中出现的错误，
                因为字符串是可以和非字符串正确地比较的
            '''
            try: other = other.lower( )
            except (TypeError, AttributeError, ValueError): pass
            return str_meth(self._lowered, other, *args)
        # 仅 Python 2.4，增加一条语句：x.func_name = name
        setattr(iStr, name, x)
    # 将_make_case_insensitive 函数应用于指定的方法
    for name in 'eq lt le gt gt ne cmp contains'.split( ):
        _make_case_insensitive('__%s__' % name)
    for name in 'count endswith find index rfind rindex startswith'.split( ):
        _make_case_insensitive(name)
    # 注意，我们并不修改 replace、split、strip 等方法
    # 当然，如果有需要，也可以对它们进行修改
    del _make_case_insensitive    # 删除帮助函数，已经不再需要了
```

讨论

iStr 类的一些实现上的选择很值得讨论。首先，我们在__init__中一次性生成了小写版本，这是因为我们认识到在 iStr 的典型应用中，这个小写版本将会被反复地使用。我们在一个私有的变量中保存这个小写版本，将其作为一个属性，当然，也别保护得太过分了（它以一个下划线开头，而不是两个下划线），因为如果从 iStr 再派生子类（比如，进一步对其扩展，支持大小写不敏感的切分和替换等，正如“解决方案”注释中所说的），iStr 的子类很有可能会需要访问其父类 iStr 的一些关键的“实现细节”。

这里我们没有提供其他一些方法的大小写不敏感的版本，如 replace，因为这个例子已经清晰地展示了一种通用的建立输入和输出之间联系的方式。根据应用进行特别定制的子类将提供最能够满足需求的功能。比如，replace 方法并没有被封装，则我们对一个 iStr 的实例调用 replace，返回的是 str 的实例，而不是 iStr。如果这会给你的应用带来问题，可以将所有的返回字符串的 iStr 方法封装起来，这就可以确保所有返回的结果是 iStr 的实例。基于这个目的，需要另一个单独的助手函数，相似但不完全等同于解决方案中给出的_make_case_insensitive：

```
def _make_return_iStr(name):
    str_meth = getattr(str, name)
    def x(*args):
        return iStr(str_meth(*args))
    setattr(iStr, name, x)
```

需要对所有返回字符串的方法的名字应用这个助手函数，_make_return_iStr：

```
for name in 'center ljust rjust strip lstrip rstrip'.split( ):
    _make_return_iStr(name)
```

字符串有约 20 种方法（包括一些特殊方法，比如__add__和__mul__），需要考虑哪

些方法应该被封装起来。也可以把一些额外的方法，比如 split 和 join（它们可能需要一些特别的处理）封装起来，或者其他的方法，如 encode 和 decode，对于此类方法，除非定义了一个大小写不敏感的 unicode 子类型，否则无法处理它们。而实际上，针对一个特定的应用，可能不是所有的未封装的方法都会引起问题。正如你所见的那样，由于 Python 字符串的方法和功能很丰富，要想用一种通用的不依赖于特定应用的方式，完全彻底地定制出一个子类型，还是要花点功夫的。

iStr 的实现很谨慎，主要是为了避免一些重复性的例行公事般的代码（通常是冗长且容易滋生 bug 的代码），如果我们用普通的方式重载 str 每一个需要的方法，在类的实现中写上一堆 def 语句，很有可能就会陷入这种尴尬的境地。使用可自定义的元类或者其他的高级技术对这个例子而言也不会有什么特别的优势，但使用一个辅助函数来生成和安装封装层闭包，就可以轻易地避开问题。然后我们在两个循环中使用该辅助函数，一个循环处理常用的方法，另一个则处理特殊的方法。这两个循环都必须被放置在 class 语句之后，正如我们在解决方案中给出的代码所示，这是因为这两个循环需要修改 iStr 类对象，但除非用 class 语句完成对 iStr 类的声明，否则那个类对象根本就不存在（因此当然也无法修改）。

在 Python 2.4 中，可以重新指定函数对象的 func_name 属性，在本例中，当对 iStr 实例应用内省机制时，可以用这种方法让代码变得更加清晰和易读。但在 Python 2.3 中，函数对象的 func_name 属性是只读的。因此，在本节的讨论中，我们仅仅是指出了另一种可能性，我们不想因为这个小问题失去对 Python 2.3 的兼容性。

大小写不敏感（但仍保留了大小写信息）的字符串有很多用途，包括提高对用户输入进行解析的宽松度，在文件系统（比如 Windows 和 Macintosh 的文件系统）中查找名字包含指定字符的文件，等等。你可能会发现，有很多地方需要“大小写不敏感”的容器类型，比如字典、列表、集合等——它们都需要在某些场合，忽略掉 key 或者子项的大小写的信息。很明显，一个好的方法是一次性构建出“大小写不敏感”的比较和查询功能；现在你的工具箱中已经增加了本节提供的解决方案，可以对字符串进行任何需要的封装和定制，你甚至还能定制其他一些你希望具备“大小写不敏感”能力的容器类型。

比如，一个所有子项都是字符串的列表，你希望能够进行一些大小写无关的处理（如用 count 和 index 进行排序），完全可以基于 iStr 的实现，轻易地构建一个 iList：

```
class iList(list):
      def __init__(self, *args):
            list.__init__(self, *args)
            # 依赖__setitem__将各项封装为 iStr
            self[:] = self
      wrap_each_item = iStr
      def __setitem__(self, i, v):
            if isinstance(i, slice): v = map(self.wrap_each_item, v)
```

```
            else: v = self.wrap_each_item(v)
            list.__setitem__(self, i, v)
    def append(self, item):
            list.append(self, self.wrap_each_item(item))
    def extend(self, seq):
            list.extend(self, map(self.wrap_each_item, seq))
```

本质上，我们做的事情是把 iList 实例中每个子项都通过调用 iStr 来封装，其余部分则保持原状。

另外提一句，iList 的实现方式使得可以根据应用提供特定的 iStr，轻易地完成对子类的定制：只需在 iList 的子类中重载成员变量 wrap_each_item 即可。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于 str 的章节，主要是字符串方法，以及一些特殊的用于比较和哈希的方法。

1.25 将HTML文档转化为文本显示到UNIX终端上

感谢：Brent Burley、Mark Moraes

任务

需要将 HTML 文档中的文本展示在 UNIX 终端上，同时还要支持粗体和下划线的显示。

解决方案

最简单的方法是写一个过滤的脚本，从标准输入接收 HTML，将输出文本和终端控制序列打印到标准的输出上。由于本节的问题只针对 UNIX，我们可以借助 Python 标准库的 os 模块提供的 popen 函数，通过 UNIX 的命令 tput 获取所需的终端控制序列：

```
#!/usr/bin/env python
import sys, os, htmllib, formatter
# 使用 UNIX 的 tput 来获得粗体、下划线和重设的转义序列
set_bold = os.popen('tput bold').read( )
set_underline = os.popen('tput smul').read( )
perform_reset = os.popen('tput sgr0').read( )
class TtyFormatter(formatter.AbstractFormatter):
    ''' 一个保留粗体和斜体状态的格式化对象，并输出
        相应的终端控制序列
    '''
    def __init__(self, writer):
        # 首先，像往常一样，初始化超类
        formatter.AbstractFormatter.__init__(self, writer)
```

```
            # 一开始既没有粗体也没有斜体状态，未保存任何信息
            self.fontState = False, False
            self.fontStack = [  ]
        def push_font(self, font):
            # font 元组有 4 项，我们只看与粗体和斜体的状态
            # 有关的两个标志
            size, is_italic, is_bold, is_tt = font
            self.fontStack.append((is_italic, is_bold))
            self._updateFontState( )
        def pop_font(self, *args):
            # 回到前一个 font 状态
            try:
                self.fontStack.pop( )
            except IndexError:
                pass
            self._updateFontState( )
        def updateFontState(self):
            # 输出正确的终端控制序列，如果粗体和/或斜体（==underline）
            # 的状态被刚刚改变的话
            try:
                newState = self.fontStack[-1]
            except IndexError:
                newState = False, False
            if self.fontState != newState:
                # 相关的状态改变：重置终端
                print perform_reset,
                # 如果需要的话，设置下划线与/或粗体状态
                if newState[0]:
                    print set_underline,
                if newState[1]:
                    print set_bold,
                # 记住当前的两个状态
                self.fontState = newState
    # 生成写入、格式化、解析对象，根据需要将它们连接起来
    myWriter = formatter.DumbWriter( )
    if sys.stdout.isatty( ):
        myFormatter = TtyFormatter(myWriter)
    else:
        myFormatter = formatter.AbstractFormatter(myWriter)
    myParser = htmllib.HTMLParser(myFormatter)
    # 将标准输入和终端操作提供给解析器
    myParser.feed(sys.stdin.read( ))
    myParser.close( )
```

讨论

Python 标准库提供的 formatter.AbstractFormatter 类，可以在任何场合工作。另一方面，它的子类 TtyFormatter 提供的一些改良，则主要是为了操纵和使用类 UNIX（UNIX-like）

终端，具体地说，也就是通过 UNIX 命令 tput 获取控制粗体和下划线的转义序列，并将终端重置为基本状态。

很多系统并没有通过 UNIX 认证，比如 Linux 和 Mac OS X，但它们也提供了一个可用的 tput 命令，因此本节的 TtyFormatter 子类在这样的系统中仍然可以正常工作。或者这么说，可以用更宽泛的眼光来看待本节提及的“UNIX”，就好像我们在一些其他讨论中所用的“UNIX”概念：如果你愿意，可以认为它指的是“*ix”。

如果你的“终端”模拟器支持其他的一些控制输出表现的转义序列，也可以根据情况修改 TtyFormatter 类。比如，据说在 Windows 中，cmd.exe 命令能够支持所有标准的 ANSI 转义序列，所以如果你只想在 Windows 上运行你的脚本，可以用硬编码的方式在类中写入那些序列。

很多时候，你可能会更喜欢用 UNIX 已经提供的命令，比如 lynx -dump -，相比于本节方案中提供的方法，这些命令能够提供更丰富更具表现力的输出。但有时你会发现 Python 安装在一个不提供这种有用的命令（如 lynx）的系统上，那么本节给出的方法就显得很方便和简洁了。

更多资料

Library Reference 和 *Python in a Nutshell* 文档中关于 formatter 和 htmllib 模块的内容；在 UNIX 或者类 UNIX 系统中用 man 命令查看 tput 命令的有关信息。