Beautiful Soup 中文文档

原文 by Leonard Richardson (leonardr@segfault.org)
翻译 by Richie Yan (richieyan@gmail.com)
###如果有些翻译的不准确或者难以理解，直接看例子吧。###

英文原文点这里

Beautiful Soup 是用Python写的一个HTML/XML的解析器，它可以很好的处理不规范标记并生成剖析树(parse tree)。 它提供简单又常用的导航（navigating），搜索以及修改剖析树的操作。它可以大大节省你的编程时间。 对于Ruby，使用Rubyful Soup。

这个文档说明了Beautiful Soup 3.0主要的功能特性，并附有例子。 从中你可以知道这个库有哪些好处，它是怎样工作的， 怎样让它帮做你想做的事以及你该怎样做当它做的和你期待不一样。

目录

快速开始

从这里获得 Beautiful Soup。 变更日志 描述了3.0 版本与之前版本的不同。

在程序中中导入 Beautiful Soup库:

```
from BeautifulSoup import BeautifulSoup          # For processing HTML
from BeautifulSoup import BeautifulStoneSoup     # For processing XML
import BeautifulSoup                             # To get everything
```

下面的代码是Beautiful Soup基本功能的示范。你可以复制粘贴到你的python文件中，自己运行看看。

```
from BeautifulSoup import BeautifulSoup
import re
doc = ['<html><head><title>Page title</title></head>',
       '<body><p id="firstpara" align="center">This is paragraph <b>one</b>.',
       '<p id="secondpara" align="blah">This is paragraph <b>two</b>.',
       '</html>']
soup = BeautifulSoup(''.join(doc))
print soup.prettify()
# <html>
#  <head>
#   <title>
#    Page title
#   </title>
#  </head>
#  <body>
#   <p id="firstpara" align="center">
#    This is paragraph
#    <b>
#     one
#    </b>
#    .
#   </p>
#   <p id="secondpara" align="blah">
#    This is paragraph
#    <b>
#     two
#    </b>
#    .
#   </p>
#  </body>
# </html>
```

navigate soup的一些方法:

```
soup.contents[0].name
# u'html'
soup.contents[0].contents[0].name
# u'head'
head = soup.contents[0].contents[0]
head.parent.name
# u'html'
head.next
# <title>Page title</title>
head.nextSibling.name
# u'body'
head.nextSibling.contents[0]
# <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>
head.nextSibling.contents[0].nextSibling
# <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>
```

下面是一些方法搜索soup，获得特定标签或有着特定属性的标签：

```
titleTag = soup.html.head.title
titleTag
# <title>Page title</title>
titleTag.string
# u'Page title'
len(soup('p'))
# 2
soup.findAll('p', align="center")
# [<p id="firstpara" align="center">This is paragraph <b>one</b>. </p>]
soup.find('p', align="center")
# <p id="firstpara" align="center">This is paragraph <b>one</b>. </p>
soup('p', align="center")[0]['id']
# u'firstpara'
soup.find('p', align=re.compile('^b.*'))['id']
# u'secondpara'
soup.find('p').b.string
# u'one'
soup('p')[1].b.string
# u'two'
```

修改soup也很简单：

```
titleTag['id'] = 'theTitle'
titleTag.contents[0].replaceWith("New title")
soup.html.head
# <head><title id="theTitle">New title</title></head>
soup.p.extract()
soup.prettify()
# <html>
#  <head>
```

```
#   <title id="theTitle">
#    New title
#   </title>
#  </head>
#  <body>
#   <p id="secondpara" align="blah">
#    This is paragraph
#    <b>
#     two
#    </b>
#    .
#   </p>
#  </body>
# </html>
soup.p.replaceWith(soup.b)
# <html>
#  <head>
#   <title id="theTitle">
#    New title
#   </title>
#  </head>
#  <body>
#   <b>
#    two
#   </b>
#  </body>
# </html>
soup.body.insert(0, "This page used to have ")
soup.body.insert(2, " &lt;p&gt; tags!")
soup.body
# <body>This page used to have <b>two</b> &lt;p&gt; tags!</body>
```

一个实际例子，用于抓取 ICC Commercial Crime Services weekly piracy report页面, 使用Beautiful Soup剖析并获得发生的盗版事件:

```
import urllib2
from BeautifulSoup import BeautifulSoup
page = urllib2.urlopen("http://www.icc-ccs.org/prc/piracyreport.php")
soup = BeautifulSoup(page)
for incident in soup('td', width="90%"):
    where, linebreak, what = incident.contents[:3]
    print where.strip()
    print what.strip()
    print
```

剖析文档

Beautiful Soup使用XML或HTML文档以字符串的方式(或类文件对象)构造。 它剖析文档并在内存中创建通讯的数据结构

如果你的文档格式是非常标准的，解析出来的数据结构正如你的原始文档。但是 如果你的文档有问题，Beautiful Soup会使用heuristics修复可能的结构问题。

剖析 HTML

使用 BeautifulSoup 类剖析HTML文档。 BeautifulSoup会得出以下一些信息：

- 有些标签可以内嵌 (<BLOCKQUOTE>) ，有些不行 (<P>).
- table和list标签有一个自然的内嵌顺序。例如，<TD> 标签内为 <TR> 标签，而不会相反。
- <SCRIPT> 标签的内容不会被剖析为HTML。
- <META> 标签可以知道文档的编码类型。

这是运行例子:

```
from BeautifulSoup import BeautifulSoup
html = "<html><p>Para 1<p>Para 2<blockquote>Quote 1<blockquote>Quote 2"
soup = BeautifulSoup(html)
print soup.prettify()
# <html>
#  <p>
#   Para 1
#  </p>
#  <p>
#   Para 2
#   <blockquote>
#    Quote 1
#    <blockquote>
#     Quote 2
#    </blockquote>
#   </blockquote>
#  </p>
# </html>
```

注意：BeautifulSoup 会智能判断那些需要添加关闭标签的位置，即使原始的文档没有。

也就是说那个文档不是一个有效的HTML，但是它也不是太糟糕。下面是一个比较糟糕的文档。 在一些问题中，它的<FORM>的开始在<TABLE> 外面，结束在<TABLE>里面。 (这种HTML在一些大公司的页面上也屡见不鲜)

```
from BeautifulSoup import BeautifulSoup
html = """
<html>
<form>
 <table>
 <td><input name="input1">Row 1 cell 1
 <tr><td>Row 2 cell 1
 </form>
 <td>Row 2 cell 2<br>This</br> sure is a long cell
</body>
</html>"""
```

Beautiful Soup 也可以处理这个文档：

```
print BeautifulSoup(html).prettify()
# <html>
#  <form>
#   <table>
#    <td>
#     <input name="input1" />
#     Row 1 cell 1
#    </td>
#    <tr>
#     <td>
#      Row 2 cell 1
#     </td>
#    </tr>
#   </table>
#  </form>
#  <td>
#   Row 2 cell 2
#   <br />
#   This
#   sure is a long cell
#  </td>
# </html>
```

table的最后一个单元格已经在标签<TABLE>外了；Beautiful Soup 决定关闭<TABLE>标签当它在<FORM>标签哪里关闭了。 写这个文档家伙原本打算使用<FORM>标签扩展到table的结尾，但是Beautiful Soup 肯定不知道这些。即使遇到这样糟糕的情况，Beautiful Soup 仍可以剖析这个不合格文档，使你开业存取所有数据。

剖析 XML

BeautifulSoup 类似浏览器，是个具有启发性的类，可以尽可能的推测HTML文档作者的意图。 但是XML没有固定的标签集合，因此这些启发式的功能没有作用。因此 BeautifulSoup处理XML不是很好。

使用BeautifulStoneSoup类剖析XML文档。它是一个 概括的类，没有任何特定的XML方言已经简单的标签内嵌规则。 下面是范例：

```
from BeautifulSoup import BeautifulStoneSoup
xml = "<doc><tag1>Contents 1<tag2>Contents 2<tag1>Contents 3"
soup = BeautifulStoneSoup(xml)
print soup.prettify()
# <doc>
#  <tag1>
#   Contents 1
#   <tag2>
#    Contents 2
#   </tag2>
#  </tag1>
#  <tag1>
#   Contents 3
#  </tag1>
# </doc>
```

BeautifulStoneSoup的一个主要缺点就是它不知道如何处理自结束标签 。 HTML 有固定的自结束标签集合，但是XML取决对应的DTD文件。你可以通过传递selfClosingTags 的参数的名字到 BeautifulStoneSoup的构造器中，指定自结束标签:

```
from BeautifulSoup import BeautifulStoneSoup
xml = "<tag>Text 1<selfclosing>Text 2"
print BeautifulStoneSoup(xml).prettify()
# <tag>
#  Text 1
#  <selfclosing>
#   Text 2
#  </selfclosing>
# </tag>
print BeautifulStoneSoup(xml, selfClosingTags=['selfclosing']).prettify()
# <tag>
#  Text 1
#  <selfclosing />
```

```
#  Text 2                    欢迎加入非盈利Python学习交流编程QQ群783462347，群里免费提供500+本Python书籍！
# </tag>
```

如果它不工作

这里有 一些其他的剖析类 使用与上述两个类不同的智能感应。 你也可以子类化以及定制一个剖析器 使用你自己的智能感应方法。

使用Unicode的Beautiful Soup，Dammit

当你的文档被剖析之后，它就自动被转换为unicode。 Beautiful Soup 只存储Unicode字符串。

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("Hello")
soup.contents[0]
# u'Hello'
soup.originalEncoding
# 'ascii'
```

使用UTF-8编码的日文文档例子：

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("\xe3\x81\x93\xe3\x82\x8c\xe3\x81\xaf")
soup.contents[0]
# u'\u3053\u308c\u306f'
soup.originalEncoding
# 'utf-8'
str(soup)
# '\xe3\x81\x93\xe3\x82\x8c\xe3\x81\xaf'
# Note: this bit uses EUC-JP, so it only works if you have cjkcodecs
# installed, or are running Python 2.4.
soup.__str__('euc-jp')
# '\xa4\xb3\xa4\xec\xa4\xcf'
```

Beautiful Soup 使用一个称为UnicodeDammit 的类去来检测文档的编码，并将其转换为Unicode。 如果你需要为其他文档（没有石油Beautiful Soup剖析过得文档）使用这转换，你也可以 直接使用UnicodeDammit。 它是基于Universal Feed Parser开发的。

如果你使用Python2.4之前的版本，请下载和安装cjkcodecs 以及iconvcodec 是python支持更多的编码，特别是CJK编码。要想更好地自动检测， 你也要安装chardet

Beautiful Soup 会按顺序尝试不同的编码将你的文档转换为Unicode：

- 可以通过fromEncoding参数传递编码类型给soup的构造器
- 通过文档本身找到编码类型：例如XML的声明或者HTML文档http-equiv的META标签。 如果Beautiful Soup在文档中发现编码类型，它试着使用找到的类型转换文档。 但是，如果你明显的指定一个编码类型， 并且成功使用了编码：这时它会忽略任何它在文档中发现的编码类型。
- 通过嗅探文件开头的一下数据，判断编码。如果编码类型可以被检测到， 它将是这些中的一个：UTF-*编码，EBCDIC或者ASCII。
- 通过chardet 库,嗅探编码，如果你安装了这个库。
- UTF-8
- Windows-1252

Beautiful Soup总是会猜对它可以猜测的。但是对于那些没有声明以及有着奇怪编码 的文档，它会常常会失败。这时，它会选择Windows-1252编码，这个可能是错误的编码。 下面是EUC-JP的例子，Beautiful Soup猜错了编码。(重申一下：因为它使用了EUC-JP， 这个例子只会在 python 2.4或者你安装了cjkcodecs的情况下才工作。)：

```
from BeautifulSoup import BeautifulSoup
euc_jp = '\xa4\xb3\xa4\xec\xa4\xcf'
soup = BeautifulSoup(euc_jp)
soup.originalEncoding
# 'windows-1252'
str(soup)
# '\xc2\xa4\xc2\xb3\xc2\xa4\xc3\xac\xc2\xa4\xc3\x8f'      # Wrong!
```

但如果你使用fromEncoding参数指定编码， 它可以正确的剖析文档，并可以将文档转换为UTF-8或者转回EUC-JP。

```
soup = BeautifulSoup(euc_jp, fromEncoding="euc-jp")
soup.originalEncoding
# 'windows-1252'
str(soup)
# '\xe3\x81\x93\xe3\x82\x8c\xe3\x81\xaf'                  # Right!
soup.__str__(self, 'euc-jp') == euc_jp
# True
```

如果你指定Beautiful Soup使用 Windows-1252编码（或者类似的编码如ISO-8859-1，ISO-8859-2)，Beautiful Soup会找到并破坏文档的smart quotes以及其他的Windows-specific 字符。 这些字符不会转换为相应的Unicode,而是将它们变为HTML entities (BeautifulSoup) 或者XML entitis(BeautifulStoneSoup)。

但是，你可以指定参数smartQuotesTo=None到BeautifulSoup构造器，这时Smart Quotes会被正确的转换为Unicode。你也可以指定smartQuotesTo为"xml"或"html" 去改变BeautifulSoup和BeautifulStoneSoup的默认操作。

```
from BeautifulSoup import BeautifulSoup, BeautifulStoneSoup
text = "Deploy the \x91SMART QUOTES\x92!"
str(BeautifulSoup(text))
# 'Deploy the &lsquo;SMART QUOTES&rsquo;!'
str(BeautifulStoneSoup(text))
# 'Deploy the &#x2018;SMART QUOTES&#x2019;!'
str(BeautifulSoup(text, smartQuotesTo="xml"))
# 'Deploy the &#x2018;SMART QUOTES&#x2019;!'
BeautifulSoup(text, smartQuotesTo=None).contents[0]
# u'Deploy the \u2018SMART QUOTES\u2019!'
```

输出文档

你可以使用 str函数将Beautiful Soup文档（或者它的子集）转换为字符串， 或者使用它的code>prettify或renderContents。 你也可以使用unicode函数以Unicode字符串的形式获得。

prettify 方法添加了一些换行和空格以便让文档结构看起来更清晰。 它也将那些只包含空白符的，可能影响一个XML文档意义的文档节点(nodes)剔除(strips out)。 str和unicode函数不会剔除这些节点，他们也不会添加任何空白符。

看看这个例子：

```
from BeautifulSoup import BeautifulSoup
doc = "<html><h1>Heading</h1><p>Text"
soup = BeautifulSoup(doc)
str(soup)
# '<html><h1>Heading</h1><p>Text</p></html>'
soup.renderContents()
# '<html><h1>Heading</h1><p>Text</p></html>'
soup.__str__()
# '<html><h1>Heading</h1><p>Text</p></html>'
unicode(soup)
# u'<html><h1>Heading</h1><p>Text</p></html>'
soup.prettify()
# '<html>\n <h1>\n  Heading\n </h1>\n <p>\n  Text\n </p>\n</html>'
print soup.prettify()
# <html>
#  <h1>
#   Heading
#  </h1>
#  <p>
#   Text
#  </p>
# </html>
```

可以看到使用文档中的tag成员时 str和renderContents返回的结果是不同的。

```
heading = soup.h1
str(heading)
# '<h1>Heading</h1>'
heading.renderContents()
# 'Heading'
```

当你调用__str__,prettify或者renderContents时， 你可以指定输出的编码。默认的编码(str使用的)是UTF-8。 下面是处理ISO-8851-1的串并以不同的编码输出同样的串的例子。

```
from BeautifulSoup import BeautifulSoup
doc = "Sacr\xe9 bleu!"
soup = BeautifulSoup(doc)
str(soup)
# 'Sacr\xc3\xa9 bleu!'                          # UTF-8
soup.__str__("ISO-8859-1")
# 'Sacr\xe9 bleu!'
soup.__str__("UTF-16")
# '\xff\xfeS\x00a\x00c\x00r\x00\xe9\x00 \x00b\x00l\x00e\x00u\x00!\x00'
soup.__str__("EUC-JP")
# 'Sacr\x8f\xab\xb1 bleu!'
```

如果原始文档含有编码声明，Beautiful Soup会将原始的编码声明改为新的编码。 也就是说，你载入一个HTML文档到BeautifulSoup后，在输出它，不仅HTML被清理 过了，而且可以明显的看到它已经被转换为UTF-8。

这是HTML的例子：

```
from BeautifulSoup import BeautifulSoup
doc = """<html>
<meta http-equiv="Content-type" content="text/html; charset=ISO-Latin-1" >
Sacr\xe9 bleu!
</html>"""
print BeautifulSoup(doc).prettify()
# <html>
#  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
```

```
#  Sacré bleu!                    欢迎加入非盈利Python学习交流编程QQ群783462347，群里免费提供500+本Python书籍！
# </html>
```

这是XML的例子：

```
from BeautifulSoup import BeautifulStoneSoup
doc = """<?xml version="1.0" encoding="ISO-Latin-1">Sacr\xe9 bleu!"""
print BeautifulStoneSoup(doc).prettify()
# <?xml version='1.0' encoding='utf-8'>
# Sacré bleu!
```

剖析树

到目前为止，我们只是载入文档，然后再输出它。 现在看看更让我们感兴趣的剖析树： Beautiful Soup剖析一个文档后生成的数据结构。

剖析对象 (BeautifulSoup或 BeautifulStoneSoup的实例)是深层嵌套(deeply-nested)，精心构思的(well-connected)的数据结构，可以与XML和HTML结构相互协调。 剖析对象包括2个其他类型的对象，Tag对象， 用于操纵像<TITLE> ，<B>这样的标签；NavigableString对象， 用于操纵字符串，如"Page title"和"This is paragraph"。

NavigableString的一些子类 (CData, Comment, Declaration, and ProcessingInstruction)，也处理特殊XML结构。 它们就像NavigableString一样，除了但他们被输出时， 他们会被添加一些额外的数据。下面是一个包含有注释(comment)的文档：

```
from BeautifulSoup import BeautifulSoup
import re
hello = "Hello! <!--I've got to be nice to get what I want.-->"
commentSoup = BeautifulSoup(hello)
comment = commentSoup.find(text=re.compile("nice"))
comment.__class__
# <class 'BeautifulSoup.Comment'>
comment
# u"I've got to be nice to get what I want."
comment.previousSibling
# u'Hello! '
str(comment)
# "<!--I've got to be nice to get what I want.-->"
print commentSoup
# Hello! <!--I've got to be nice to get what I want.-->
```

现在，我们深入研究一下我们开头使用的那个文档：

```
from BeautifulSoup import BeautifulSoup
doc = ['<html><head><title>Page title</title></head>',
       '<body><p id="firstpara" align="center">This is paragraph <b>one</b>.',
       '<p id="secondpara" align="blah">This is paragraph <b>two</b>.',
       '</html>']
soup = BeautifulSoup(''.join(doc))
print soup.prettify()
# <html>
#  <head>
#   <title>
#    Page title
#   </title>
#  </head>
#  <body>
#   <p id="firstpara" align="center">
#    This is paragraph
#    <b>
#     one
#    </b>
#    .
#   </p>
#   <p id="secondpara" align="blah">
#    This is paragraph
#    <b>
#     two
#    </b>
#    .
#   </p>
#  </body>
# </html>
```

Tag的属性

Tag和NavigableString对象有很多有用的成员，在 Navigating剖析树和 Searching剖析树中我们会更详细的介绍。 现在，我们先看看这里使用的Tag成员：属性

SGML标签有属性：.例如，在上面那个HTML 中每个<P>标签都有"id"属性和"align"属性。 你可以将Tag看成字典来访问标签的属性：

```
firstPTag, secondPTag = soup.findAll('p')
firstPTag['id']
# u'firstPara'
```

```
secondPTag['id']
# u'secondPara'
```


NavigableString对象没有属性;只有Tag 对象有属性。

Navigating剖析树

Tag 对象都有如下含有所有的成员的列表(尽管，某些实际的成员值可能为None). NavigableString对象也有下面这些成员，除了contents和 string成员。

parent

上面那个 例子中，<HEAD> Tag的parent是<HTML> Tag. <HTML> Tag 的parent是BeautifulSoup 剖析对象自己。 剖析对象的parent是None. 利用parent，你可以向前遍历剖析树。

```
soup.head.parent.name
# u'html'
soup.head.parent.parent.__class__.__name__
# 'BeautifulSoup'
soup.parent == None
# True
```

contents

使用parent向前遍历树。使用contents向后遍历树。 contents是Tag的有序列表， NavigableString 对象包含在一个页面元素内。只有最高层的剖析对象和 Tag 对象有contents。 NavigableString 只有strings，不能包含子元素，因此他们也没有contents.

在上面的例子中， contents 的第一个<P> Tag是个列表，包含一个 NavigableString ("This is paragraph "), 一个<B> Tag, 和其它的NavigableString (".")。而contents 的<B> Tag: 包含一个NavigableString ("one")的列表。

```
pTag = soup.p
pTag.contents
# [u'This is paragraph ', <b>one</b>, u'.']
pTag.contents[1].contents
# [u'one']
pTag.contents[0].contents
# AttributeError: 'NavigableString' object has no attribute 'contents'
```

string

为了方便，如果一个标签只有一个子节点且是字符串类型，这个自己可以这样访问 tag.string，等同于tag.contents[0]的形式。 在上面的例子中， soup.b.string是个NavigableString对象，它的值是Unicode字符串"one". 这是剖析树中<B>Tag 的第一个string。

```
soup.b.string
# u'one'
soup.b.contents[0]
# u'one'
```

但是soup.p.string是None， 剖析中的第一个<P> Tag 拥有多个子元素。soup.head.string也为None， 虽然<HEAD> Tag只有一个子节点，但是这个子节点是Tag类型 (<TITLE> Tag)， 不是NavigableString。

```
soup.p.string == None
# True
soup.head.string == None
# True
```

nextSibling 和previousSibling

使用它们你可以跳往在剖析树中同等层次的下一个元素。 在上面的文档中， <HEAD> Tag的nextSibling 是<BODY> Tag， 因为<BODY> Tag是在<html> Tag的下一层。 <BODY>标签的nextSibling为None， 因为<HTML>下一层没有标签是直接的在它之后。

```
soup.head.nextSibling.name
# u'body'
soup.html.nextSibling == None
# True
```

相应的<BODY> Tag的previousSibling是<HEAD>标签， <HEAD> Tag的previousSibling为None:

```
soup.body.previousSibling.name
# u'head'
```

```
soup.head.previousSibling == None欢迎加入非盈利Python学习交流编程QQ群783462347，群里免费提供500+本Python书籍！
# True
```

更多例子：<P> Tag的第一个nextSibling是第二个 <P> Tag。 第二个<P>Tag里的<B>Tag的previousSibling是 NavigableString"This is paragraph"。 这个NavigableString的previousSibling是None， 不会是第一个<P> Tag里面的任何元素。

```
soup.p.nextSibling
# <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>
secondBTag = soup.findAll('b')[1]
secondBTag.previousSibling
# u'This is paragraph'
secondBTag.previousSibling.previousSibling == None
# True
```

next和previous

使用它们可以按照soup处理文档的次序遍历整个文档，而不是它们在剖析树中看到那种次序。 例如<HEAD> Tag的next是<TITLE>Tag， 而不是<BODY> Tag。 这是因为在原始文档中，<TITLE> tag 直接在<HEAD>标签之后。

```
soup.head.next
# u'title'
soup.head.nextSibling.name
# u'body'
soup.head.previous.name
# u'html'
```

Where next and previous are concerned, a Tag's contents come before its nextSibling. 通常不会用到这些成员，但有时使用它们能够非常方便地从剖析树获得不易找到的信息。

遍历一个标签(Iterating over a Tag)

你可以像遍历list一样遍历一个标签(Tag)的contents 。 这非常有用。类似的，一个Tag的有多少child可以直接使用len(tag)而不必使用len(tag.contents)来获得。 以上面那个文档中的为例：

```
for i in soup.body:
    print i
# <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>
# <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>
len(soup.body)
# 2
len(soup.body.contents)
# 2
```

使用标签(tag)名作为成员

像剖析对象或Tag对象的成员一样使用Tag名可以很方便的操作剖析树。 前面一些例子我们已经用到了这种方式。以上述文档为例， soup.head获得文档第一个<HEAD>标签：

```
soup.head
# <head><title>Page title</title></head>
```

通常，调用mytag.foo获得的是mytag的第一个child，同时必须是一个<FOO> 标签。 如果在mytag中没有<FOO> 标签，mytag.foo返回一个None。 你可以使用这中方法快速的读取剖析树：

```
soup.head.title
# <title>Page title</title>
soup.body.p.b.string
# u'one'
```

你也可以使用这种方法快速的跳到剖析树的某个特定位置。例如，如果你担心<TITLE> tags会离奇的在<HEAD> tag之外， 你可以使用soup.title去获得一个HTML文档的标题(title)，而不必使用soup.head.title：

```
soup.title.string
# u'Page title'
```

soup.p跳到文档中的第一个 <P> tag，不论它在哪里。 soup.table.tr.td 跳到文档总第一个table的第一列第一行。

这些成员实际上是下面first 方法的别名，这里更多介绍。 这里提到是因为别名使得一个定位(zoom)一个结构良好剖析树变得异常容易。

获得第一个<FOO> 标签另一种方式是使用.fooTag 而不是 .foo。 例如，soup.table.tr.td可以表示为soup.tableTag.trTag.tdTag，甚至为soup.tableTag.tr.tdTag。 如果你喜欢更明确的知道表示的意义，或者你在剖析一个标签与Beautiful Soup的方法或成员有冲突的XML文档是，使用这种方式非常有用。

```
from BeautifulSoup import BeautifulStoneSoup
xml = '<person name="Bob"><parent rel="mother" name="Alice">'
xmlSoup = BeautifulStoneSoup(xml)
xmlSoup.person.parent                       # A Beautiful Soup member
# <person name="Bob"><parent rel="mother" name="Alice"></parent></person>
xmlSoup.person.parentTag                    # A tag name
# <parent rel="mother" name="Alice"></parent>
```

如果你要找的标签名不是有效的Pyth

搜索剖析树

Beautiful Soup提供了许多方法用于浏览一个剖析树，收集你指定的Tag和NavigableString。

有几种方法去定义用于Beautiful Soup的匹配项。 我们先用深入解释最基本的一种搜索方法findAll。 和前面一样，我们使用下面这个文档说明：

```
from BeautifulSoup import BeautifulSoup
doc = ['<html><head><title>Page title</title></head>',
       '<body><p id="firstpara" align="center">This is paragraph <b>one</b>.',
       '<p id="secondpara" align="blah">This is paragraph <b>two</b>.',
       '</html>']
soup = BeautifulSoup(''.join(doc))
print soup.prettify()
# <html>
#  <head>
#   <title>
#    Page title
#   </title>
#  </head>
#  <body>
#   <p id="firstpara" align="center">
#    This is paragraph
#    <b>
#     one
#    </b>
#    .
#   </p>
#   <p id="secondpara" align="blah">
#    This is paragraph
#    <b>
#     two
#    </b>
#    .
#   </p>
#  </body>
# </html>
```

还有， 这里的两个方法(findAll和 find)仅对Tag对象以及 顶层剖析对象有效，但 NavigableString不可用。 这两个方法在Searching 剖析树内部同样可用。

The basic find method: findAll(name, attrs, recursive, text, limit, **kwargs)

方法findAll 从给定的点开始遍历整个树，并找到满足给定条件所有Tag以及NavigableString。 findall函数原型定义如下：

findAll(name=None, attrs={}, recursive=True, text=None, limit=None, **kwargs)

这些参数会反复的在这个文档中出现。其中最重要的是name参数 和keywords参数(译注：就是**kwargs参数)。

- 参数**name** 匹配tags的名字，获得相应的结果集。 有几种方法去匹配name，在这个文档中会一再的用到。

  1. 最简单用法是仅仅给定一个tag name值。下面的代码寻找文档中所有的 <B> Tag:

     ```
     soup.findAll('b')
     # [<b>one</b>, <b>two</b>]
     ```

  2. 你可以传一个正则表达式。下面的代码寻找所有以b开头的标签:

     ```
     import re
     tagsStartingWithB = soup.findAll(re.compile('^b'))
     [tag.name for tag in tagsStartingWithB]
     # [u'body', u'b', u'b']
     ```

  3. 你可以传一个list或dictionary。下面两个调用是查找所有的<TITLE>和<P>标签。 他们获得结果一样，但是后一种方法更快一些:

     ```
     soup.findAll(['title', 'p'])
     # [<title>Page title</title>,
     #  <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
     #  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
     soup.findAll({'title' : True, 'p' : True})
     # [<title>Page title</title>,
     #  <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
     #  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
     ```

  4. 你可以传一个True值,这样可以匹配每个tag的name：也就是匹配每个tag。

     ```
     allTags = soup.findAll(True)
     [tag.name for tag in allTags]
     [u'html', u'head', u'title', u'body', u'p', u'b', u'p', u'b']
     ```

     这看起来不是很有用，但是当你限定属性(attribute)值时候，使用True就很有用了。

  5. 你可以传callable对象，就是一个使用Tag对象作为它唯一的参数，并返回布尔值的对象。 findAll使用的每个作为参数的Tag对象都会传递给这个callable对象， 并且如果调用返回True，则这个tag便是匹配的。

下面是查找两个并仅两个tag的标签：

```
soup.findAll(lambda tag: len(tag.attrs) == 2)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
```

下面是寻找单个字符为标签名并且没有属性的标签：

```
soup.findAll(lambda tag: len(tag.name) == 1 and not tag.attrs)
# [<b>one</b>, <b>two</b>]
```

- keyword参数用于筛选tag的属性。下面这个例子是查找拥有属性align且值为center的 所有标签：

```
soup.findAll(align="center")
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>]
```

如同name参数，你也可以使用不同的keyword参数对象，从而更加灵活的指定属性值的匹配条件。 你可以向上面那样传递一个字符串，来匹配属性的值。你也可以传递一个正则表达式，一个列表(list)，一个哈希表(hash)， 特殊值True或None，或者一个可调用的以属性值为参数的对象(注意：这个值可能为None)。 一些例子：

```
soup.findAll(id=re.compile("para$"))
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
soup.findAll(align=["center", "blah"])
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
soup.findAll(align=lambda(value): value and len(value) < 5)
# [<p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
```

特殊值True和None更让人感兴趣。 True匹配给定属性为*任意*值的标签，None匹配那些给定的属性值为*空*的标签。 一些例子如下：

```
soup.findAll(align=True)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
[tag.name for tag in soup.findAll(align=None)]
# [u'html', u'head', u'title', u'body', u'b', u'b']
```

如果你需要在标签的属性上添加更加复杂或相互关联的(interlocking)匹配值， 如同上面一样，以callable对象的传递参数来处理Tag对象。

在这里你也许注意到一个问题。 如果你有一个文档，它有一个标签定义了一个name属性,会怎么样？ 你不能使用name为keyword参数，因为Beautiful Soup已经定义了一个name参数使用。 你也不能用一个Python的保留字例如for作为关键字参数。

Beautiful Soup提供了一个特殊的参数attrs，你可以使用它来应付这些情况。 attrs是一个字典，用起来就和keyword参数一样：

```
soup.findAll(id=re.compile("para$"))
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
soup.findAll(attrs={'id' : re.compile("para$")})
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
```

你可以使用attrs去匹配那些名字为Python保留字的属性， 例如class, for, 以及import； 或者那些不是keyword参数但是名字为Beautiful Soup搜索方法使用的参数名的属性， 例如name, recursive, limit, text, 以及attrs本身。

```
from BeautifulSoup import BeautifulStoneSoup
xml = '<person name="Bob"><parent rel="mother" name="Alice">'
xmlSoup = BeautifulStoneSoup(xml)
xmlSoup.findAll(name="Alice")
# []
xmlSoup.findAll(attrs={"name" : "Alice"})
# [parent rel="mother" name="Alice"></parent>]
```

使用CSS类查找

对于CSS类attrs参数更加方便。例如class不仅是一个CSS属性， 也是Python的保留字。

你可以使用soup.find("tagName", { "class" : "cssClass" })搜索CSS class，但是由于有很多这样的操作， 你也可以只传递一个字符串给attrs。 这个字符串默认处理为CSS的class的参数值。

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("""Bob's <b>Bold</b> Barbeque Sauce now available in
                        <b class="hickory">Hickory</b> and <b class="lime">Lime</a>""")
soup.find("b", { "class" : "lime" })
# <b class="lime">Lime</b>
soup.find("b", "hickory")
# <b class="hickory">Hickory</b>
```

- **text** 是一个用于搜索NavigableString对象的参数。 它的值可以是字符串，一个正则表达式， 一个list或dictionary，True或None， 一个以NavigableString为参数的可调用对象：

```
soup.findAll(text="one")
# [u'one']
soup.findAll(text=u'one')
# [u'one']
soup.findAll(text=["one", "two"])
# [u'one', u'two']
soup.findAll(text=re.compile("paragraph"))
# [u'This is paragraph ', u'This is paragraph ']
soup.findAll(text=True)
# [u'Page title', u'This is paragraph ', u'one', u'.', u'This is paragraph ',
#  u'two', u'.']
soup.findAll(text=lambda(x): len(x) < 12)
# [u'Page title', u'one', u'.', u'two', u'.']
```

如果你使用text，任何指定给name 以及keyword参数的值都会被忽略。

- **recursive** 是一个布尔参数(默认为True)，用于指定Beautiful Soup遍历整个剖析树， 还是只查找当前的子标签或者剖析对象。下面是这两种方法的区别：

```
[tag.name for tag in soup.html.findAll()]
# [u'head', u'title', u'body', u'p', u'b', u'p', u'b']
[tag.name for tag in soup.html.findAll(recursive=False)]
# [u'head', u'body']
```

当recursive为false，只有当前的子标签<HTML>会被搜索。如果你需要搜索树， 使用这种方法可以节省一些时间。

- 设置**limit** 参数可以让Beautiful Soup 在找到特定个数的匹配时停止搜索。 文档中如果有上千个表格，但你只需要前四个，传值4到limit可以让你节省很多时间。 默认是没有限制(limit没有指定值).

```
soup.findAll('p', limit=1)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>]
soup.findAll('p', limit=100)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>,
#  <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>]
```

像findall一样调用tag

一个小捷径。如果你像函数一样调用剖析对象或者Tag对象， 这样你调用所用参数都会传递给findall的参数，就和调用findall一样。 就上面那个文档为例：

```
soup(text=lambda(x): len(x) < 12)
# [u'Page title', u'one', u'.', u'two', u'.']
soup.body('p', limit=1)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>]
```

find(name, attrs, recursive, text, **kwargs)

好了，我们现在看看其他的搜索方法。他们都是有和 findAll 几乎一样的参数。

find方法是最接近findAll的函数， 只是它并不会获得所有的匹配对象，它仅仅返回找到第一个可匹配对象。 也就是说，它相当于limit参数为1的结果集。 以上面的 文档为例:

```
soup.findAll('p', limit=1)
# [<p id="firstpara" align="center">This is paragraph <b>one</b>.</p>]
soup.find('p', limit=1)
# <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>
soup.find('nosuchtag', limit=1) == None
# True
```

通常，当你看到一个搜索方法的名字由复数构成 (如findAll和findNextSiblings)时， 这个方法就会存在limit参数，并返回一个list的结果。但你 看到的方法不是复数形式(如find和findNextSibling)时， 你就可以知道这函数没有limit参数且返回值是单一的结果。

first哪里去了?

早期的Beautiful Soup 版本有一些first，fetch以及fetchPrevious方法。 这些方法还在，但是已经被弃用了，也许不久就不在存在了。 因为这些名字有些令人迷惑。新的名字更加有意义： 前面提到了，复数名称的方法名，比如含有All的方法名，它将返回一个 多对象。否则，它只会返回单个对象。

Searching Within the Parse Tree

上面说明的方法findAll及find，都是从剖析树的某一点开始并一直往下。 他们反复的遍历对象的contents直到最低点。

也就是说你不能在 NavigableString对象上使用这些方法， 因为NavigableString没有contents：它们是剖析树的叶子。

[这段翻译的不太准确]但是向下搜索不是唯一的遍历剖析树的方法。在Navigating剖析树 中，我们可以使用这些方法：parent， nextSibling等。 他们都有2个相应的方法：一个类似findAll,一个类似find. 由于NavigableString对象也支持这些方法，你可以像Tag一样 使用这些方法。

为什么这个很有用?因为有些时候，你不能使用findAll或find 从Tag或NavigableString获得你想要的。例如，下面的HTML文档：

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup('''<ul>
 <li>An unrelated list
</ul>
<h1>Heading</h1>
<p>This is <b>the list you want</b>:</p>
<ul><li>The data you want</ul>''')
```

有很多方法去定位到包含特定数据的标签，非常利于学习。

```
soup('li', limit=2)[1]
# <li>The data you want</li>
```

显然，这样获得所需的<LI>标签并不稳定。如果，你只分析一次页面，这没什么影响。 但是如果你需要在一段时间分析很多次这个页面，就需要考虑一下这种方法。 If the irrelevant list grows another <LI> tag, you'll get that tag instead of the one you want, and your script will break or give the wrong data.
因为如果列表发生变化，你可能就得不到你想要的结果。

```
soup('ul', limit=2)[1].li
# <li>The data you want</li>
```

That's is a little better, because it can survive changes to the irrelevant list. But if the document grows another irrelevant list at the top, you'll get the first <LI> tag of that list instead of the one you want. A more reliable way of referring to the ul tag you want would better reflect that tag's place in the structure of the document.
这有一点好处，因为那些不相干的列表的变更生效了。 但是如果文档增长的不相干的列表在顶部，你会获得第一个<LI>标签而不是 你想要的标签。一个更可靠的方式是去引用对应的ul标签， 这样可以更好的处理文档的结构。

在HTML里面，你也许认为你想要的list是<H1>标签下的<UL>标签。 问题是那个标签不是在<H1>下，它只是在它后面。获得<H1>标签很容易，但是获得 <UL>却没法使用first和fetch， 因为这些方法只是搜索<H1>标签的contents。 你需要使用next或nextSibling来获得<UL>标签。

```
s = soup.h1
while getattr(s, 'name', None) != 'ul':
    s = s.nextSibling
s.li
# <li>The data you want</li>
```

或者，你觉得这样也许会比较稳定：

```
s = soup.find(text='Heading')
while getattr(s, 'name', None) != 'ul':
    s = s.next
s.li
# <li>The data you want</li>
```

但是还有很多困难需要你去克服。这里会介绍一下非常有用的方法。 你可以在你需要的使用它们写一些遍历成员的方法。它们以某种方式遍历树，并跟踪那些满足条件的Tag 和NavigableString对象。代替上面那个例子的第一的循环的代码，你可以这样写：

```
soup.h1.findNextSibling('ul').li
# <li>The data you want</li>
```

第二循环，你可以这样写：

```
soup.find(text='Heading').findNext('ul').li
# <li>The data you want</li>
```

这些循环代替调用findNextString和findNext。 本节剩下的内容是这种类型所用方法的参考。同时，对于遍历总是有两种方法： 一个是返回list的findAll，一个是返回单一量的find。

下面，我们再举一个例子来说明：

```
from BeautifulSoup import BeautifulSoup
doc = ['<html><head><title>Page title</title></head>',
       '<body><p id="firstpara" align="center">This is paragraph <b>one</b>.',
       '<p id="secondpara" align="blah">This is paragraph <b>two</b>.',
       '</html>']
soup = BeautifulSoup(''.join(doc))
print soup.prettify()
# <html>
#  <head>
#   <title>
#    Page title
#   </title>
#  </head>
#  <body>
#   <p id="firstpara" align="center">
#    This is paragraph
#    <b>
#     one
#    </b>
#    .
#   </p>
#   <p id="secondpara" align="blah">
#    This is paragraph
#    <b>
#     two
#    </b>
#    .
#   </p>
#  </body>
# </html>
```

findNextSiblings (name, attrs, text, limit, **kwargs) and findNextSibling (name, attrs, text, **kwargs)

这两个方法以nextSibling的成员为依据，获得满足条件的Tag和NavigableText对象。 以上面的文档为例:

```
paraText = soup.find(text='This is paragraph ')
paraText.findNextSiblings('b')
# [<b>one</b>]
paraText.findNextSibling(text = lambda(text): len(text) == 1)
# u'.'
```

findPreviousSiblings(name, attrs, text, limit, **kwargs) and findPreviousSibling(name, attrs, text, **kwargs)

这两个方法以previousSibling成员为依据，获得满足条件的Tag和 NavigableText对象。 以上面的文档为例:

```
paraText = soup.find(text='.')
paraText.findPreviousSiblings('b')
# [<b>one</b>]
paraText.findPreviousSibling(text = True)
# u'This is paragraph '
```

findAllNext(name, attrs, text, limit, **kwargs) and findNext(name, attrs, text, **kwargs)

这两个方法以next的成员为依据， 获得满足条件的Tag和NavigableText对象。 以上面的文档为例:

```
pTag = soup.find('p')
pTag.findAllNext(text=True)
# [u'This is paragraph ', u'one', u'.', u'This is paragraph ', u'two', u'.']
pTag.findNext('p')
# <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>
pTag.findNext('b')
# <b>one</b>
```

findAllPrevious(name, attrs, text, limit, **kwargs) and findPrevious(name, attrs, text, **kwargs)

这两方法以previous的成员依据， 获得满足条件的Tag和NavigableText对象。 以上面的文档为例:

```
lastPTag = soup('p')[-1]
lastPTag.findAllPrevious(text=True)
# [u'.', u'one', u'This is paragraph ', u'Page title']
# Note the reverse order!
lastPTag.findPrevious('p')
# <p id="firstpara" align="center">This is paragraph <b>one</b>.</p>
lastPTag.findPrevious('b')
# <b>one</b>
```

findParents(name, attrs, limit, **kwargs) and findParent(name, attrs, **kwargs)

这两个方法以parent成员为依据， 获得满足条件的Tag和NavigableText对象。 他们没有text参数,因为这里的对象的parent不会有NavigableString。 以上面的文档为例:

```
bTag = soup.find('b')
[tag.name for tag in bTag.findParents()]
# [u'p', u'body', u'html', '[document]']
# NOTE: "u'[document]'" means that that the parser object itself matched.
bTag.findParent('body').name
# u'body'
```

修改剖析树

现在你已经知道如何在剖析树中寻找东西了。但也许你想对它做些修改并输出出来。 你可以仅仅将一个元素从其父母的contents中分离，但是文档的其他部分仍然 拥有对这个元素的引用。Beautiful Soup 提供了几种方法帮助你修改剖析树并保持其内部的一致性。

修改属性值

你可以使用字典赋值来修改Tag对象的属性值。

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("<b id="2">Argh!</b>")
print soup
# <b id="2">Argh!</b>
b = soup.b
b['id'] = 10
print soup
# <b id="10">Argh!</b>
b['id'] = "ten"
print soup
# <b id="ten">Argh!</b>
b['id'] = 'one "million"'
print soup
# <b id='one "million"'>Argh!</b>
```

你也可以删除一个属性值，然后添加一个新的属性:

```
del(b['id'])
print soup
```

```
# <b>Argh!</b>                欢迎加入非盈利Python学习交流编程QQ群783462347，群里免费提供500+本Python书籍！
b['class'] = "extra bold and brassy!"
print soup
# <b class="extra bold and brassy!">Argh!</b>
```

删除元素

要是你引用了一个元素，你可以使用extract将它从树中抽离。 下面是将所有的注释从文档中移除的代码：

```
from BeautifulSoup import BeautifulSoup, Comment
soup = BeautifulSoup("""1<!--The loneliest number-->
                        <a>2<!--Can be as bad as one--><b>3""")
comments = soup.findAll(text=lambda text:isinstance(text, Comment))
[comment.extract() for comment in comments]
print soup
# 1
# <a>2<b>3</b></a>
```

这段代码是从文档中移除一个子树:

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("<a1></a1><a><b>Amazing content<c><d></a><a2></a2>")
soup.a1.nextSibling
# <a><b>Amazing content<c><d></d></c></b></a>
soup.a2.previousSibling
# <a><b>Amazing content<c><d></d></c></b></a>
subtree = soup.a
subtree.extract()
print soup
# <a1></a1><a2></a2>
soup.a1.nextSibling
# <a2></a2>
soup.a2.previousSibling
# <a1></a1>
```

extract方法将一个剖析树分离为两个不连贯的树。naviation的成员也因此变得看起来好像这两个树 从来不是一起的。

```
soup.a1.nextSibling
# <a2></a2>
soup.a2.previousSibling
# <a1></a1>
subtree.previousSibling == None
# True
subtree.parent == None
# True
```

使用一个元素替换另一个元素

replaceWith方法抽出一个页面元素并将其替换为一个不同的元素。 新元素可以为一个Tag（它可能包含一个剖析树）或者NavigableString。 如果你传一个字符串到replaceWith，它会变为NavigableString。 这个Navigation成员会完全融入到这个剖析树中，就像它本来就存在一样。

下面是一个简单的例子：

The new element can be a Tag (possibly with a whole parse tree beneath it) or a NavigableString. If you pass a plain old string into replaceWith, it gets turned into a NavigableString. The navigation members are changed as though the document had been parsed that way in the first place.

Here's a simple example:

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("<b>Argh!</b>")
soup.find(text="Argh!").replaceWith("Hooray!")
print soup
# <b>Hooray!</b>
newText = soup.find(text="Hooray!")
newText.previous
# <b>Hooray!</b>
newText.previous.next
# u'Hooray!'
newText.parent
# <b>Hooray!</b>
soup.b.contents
# [u'Hooray!']
```

这里有一个更复杂点的，相互替换标签(tag)的例子：

```
from BeautifulSoup import BeautifulSoup, Tag
soup = BeautifulSoup("<b>Argh!<a>Foo</a></b><i>Blah!</i>")
tag = Tag(soup, "newTag", [("id", 1)])
tag.insert(0, "Hooray!")
soup.a.replaceWith(tag)
print soup
# <b>Argh!<newTag id="1">Hooray!</newTag></b><i>Blah!</i>
```

You can even rip out an element from one part of the document and stick it in another part:
你也可以将一个元素抽出然后插入到文档的其他地方：

```
from BeautifulSoup import BeautifulSoup
text = "<html>There's <b>no</b> business like <b>show</b> business</html>"
soup = BeautifulSoup(text)
no, show = soup.findAll('b')
show.replaceWith(no)
print soup
# <html>There's  business like <b>no</b> business</html>
```

添加一个全新的元素

The Tag class and the parser classes support a method called insert. It works just like a Python list's insert method: it takes an index to the tag's contents member, and sticks a new element in that slot.
标签类和剖析类有一个insert方法，它就像Python列表的insert方法： 它使用索引来定位标签的contents成员，然后在那个位置插入一个新的元素。

This was demonstrated in the previous section, when we replaced a tag in the document with a brand new tag. You can use insert to build up an entire parse tree from scratch:
在前面那个小结中，我们在文档替换一个新的标签时有用到这个方法。你可以使用insert来重新构建整个剖析树：

```
from BeautifulSoup import BeautifulSoup, Tag, NavigableString
soup = BeautifulSoup()
tag1 = Tag(soup, "mytag")
tag2 = Tag(soup, "myOtherTag")
tag3 = Tag(soup, "myThirdTag")
soup.insert(0, tag1)
tag1.insert(0, tag2)
tag1.insert(1, tag3)
print soup
# <mytag><myOtherTag></myOtherTag><myThirdTag></myThirdTag></mytag>
text = NavigableString("Hello!")
tag3.insert(0, text)
print soup
# <mytag><myOtherTag></myOtherTag><myThirdTag>Hello!</myThirdTag></mytag>
```

An element can occur in only one place in one parse tree. If you give insert an element that's already connected to a soup object, it gets disconnected (with extract) before it gets connected elsewhere. In this example, I try to insert my NavigableString into a second part of the soup, but it doesn't get inserted again. It gets moved:
一个元素可能只在剖析树中出现一次。如果你给insert的元素已经和soup对象所关联， 它会被取消关联(使用extract)在它在被连接别的地方之前。在这个例子中，我试着插入我的NavigableString到 soup对象的第二部分，但是它并没有被再次插入而是被移动了：

```
tag2.insert(0, text)
print soup
# <mytag><myOtherTag>Hello!</myOtherTag><myThirdTag></myThirdTag></mytag>
```

This happens even if the element previously belonged to a completely different soup object. An element can only have one parent, one nextSibling, et cetera, so it can only be in one place at a time.
即使这个元素属于一个完全不同的soup对象，还是会这样。 一个元素只可以有一个parent，一个nextSibling等等，也就是说一个地方只能出现一次。

常见问题(Troubleshooting)

This section covers common problems people have with Beautiful Soup. 这一节是使用BeautifulSoup时会遇到的一些常见问题的解决方法。

为什么Beautiful Soup不能打印我的no-ASCII字符?

If you're getting errors that say: "'ascii' codec can't encode character 'x' in position y: ordinal not in range(128)", the problem is probably with your Python installation rather than with Beautiful Soup. Try printing out the non-ASCII characters without running them through Beautiful Soup and you should have the same problem. For instance, try running code like this:
如果你遇到这样的错误： "'ascii' codec can't encode character 'x' in position y: ordinal not in range(128)", 这个错误可能是Python的问题而不是BeautifulSoup。
（译者注：在已知文档编码类型的情况下，可以先将编码转换为unicode形式，在转换为utf-8编码，然后才传递给BeautifulSoup。 例如HTML的内容htm是GB2312编码:
htm=unicode(htm,'gb2312','ignore').encode('utf-8','ignore')
soup=BeautifulSoup(htm)
如果不知道编码的类型,可以使用chardet先检测一下文档的编码类型。chardet需要自己安装一下，在网上很容下到。）
试着不用Beautiful Soup而直接打印non-ASCII 字符，你也会遇到一样的问题。 例如，试着运行以下代码：

```
latin1word = 'Sacr\xe9 bleu!'
unicodeword = unicode(latin1word, 'latin-1')
print unicodeword
```

If this works but Beautiful Soup doesn't, there's probably a bug in Beautiful Soup. However, if this doesn't work, the problem's with your Python setup. Python is playing it safe and not sending non-ASCII characters to your terminal. There are two ways to override this behavior.
如果它没有问题而Beautiful Soup不行，这可能是BeautifulSoup的一个bug。 但是，如果这个也有问题，就是Python本身的问题。Python为了安全缘故不支持发送non-ASCII 到终端。有两种方法可以解决这个限制。

1. The easy way is to remap standard output to a converter that's not afraid to send ISO-Latin-1 or UTF-8 characters to the terminal.
最简单的方式是将标准输出重新映射到一个转换器，不在意发送到终端的字符类型是ISO-Latin-1还是UTF-8字符串。

   ```
   import codecs
   import sys
   streamWriter = codecs.lookup('utf-8')[-1]
   sys.stdout = streamWriter(sys.stdout)
   ```

   codecs.lookup returns a number of bound methods and other objects related to a codec. The last one is a StreamWriter object capable of wrapping an output stream.
   codecs.lookup返回一些绑定的方法和其它和codec相关的对象。 最后一行是一个封装了输出流的StreamWriter对象。

2. The hard way is to create a sitecustomize.py in your Python installation, and tell it to set the default encoding to ISO-Latin-1 or to UTF-8. Then all your Python programs will use that encoding for standard output, without you having to do something for each program. In my installation, I have a /usr/lib/python/sitecustomize.py which looks like this:
稍微困难点的方法是创建一个sitecustomize.py文件在你的Python安装中， 将默认编码设置为ISO-Latin-1或UTF-8。这样你所有的Python程序都会使用这个编码作为标准输出， 不用在每个程序里再设置一下。在我的安装中，我有一个 /usr/lib/python/sitecustomize.py，内容如下：

```
import sys
sys.setdefaultencoding("utf-8")
```

For more information about Python's Unicode support, look at Unicode for Programmers or End to End Unicode Web Applications in Python. Recipes 1.20 and 1.21 in the Python cookbook are also very helpful.
更多关于Python的Unicode支持的信息，参考 Unicode for Programmers or End to End Unicode Web Applications in Python。Python食谱的给的菜谱1.20和1.21也很有用。

Remember, even if your terminal display is restricted to ASCII, you can still use Beautiful Soup to parse, process, and write documents in UTF-8 and other encodings. You just can't print certain strings with print.
但是即使你的终端显示被限制为ASCII，你也可以使用BeautifulSoup以UTF-8和其它的编码类型来剖析，处理和修改文档。 只是对于某些字符，你不能使用print来输出。

Beautiful Soup 弄丢了我给的数据!为什么?为什么?????

Beautiful Soup can handle poorly-structured SGML, but sometimes it loses data when it gets stuff that's not SGML at all. This is not nearly as common as poorly-structured markup, but if you're building a web crawler or something you'll surely run into it.
Beautiful Soup可以处理结构不太规范的SGML，但是给它的材料非常不规范， 它会丢失数据。如果你是在写一个网络爬虫之类的程序，你肯定会遇到这种，不太常见的结构有问题的文档。

The only solution is to sanitize the data ahead of time with a regular expression. Here are some examples that I and Beautiful Soup users have discovered:
唯一的解决方法是先使用正则表达式来规范数据。 下面是一些我和一些Beautiful Soup的使用者发现的例子：

- Beautiful Soup treats ill-formed XML definitions as data. However, it loses well-formed XML definitions that don't actually exist:
Beautiful Soup 将不规范德XML定义处理为数据(data)。然而，它丢失了那些实际上不存在的良好的XML定义：

```
from BeautifulSoup import BeautifulSoup
BeautifulSoup("< ! FOO @=>")
# < ! FOO @=>
BeautifulSoup("<b><!FOO>!</b>")
# <b>!</b>
```

- If your document starts a declaration and never finishes it, Beautiful Soup assumes the rest of your document is part of the declaration. If the document ends in the middle of the declaration, Beautiful Soup ignores the declaration totally. A couple examples:
如果你的文档开始了声明但却没有关闭，Beautiful Soup假定你的文档的剩余部分都是这个声明的一部分。 如果文档在声明的中间结束了，Beautiful Soup会忽略这个声明。如下面这个例子：

```
from BeautifulSoup import BeautifulSoup
BeautifulSoup("foo<!bar")
# foo
soup = BeautifulSoup("<html>foo<!bar</html>")
print soup.prettify()
# <html>
#  foo<!bar</html>
# </html>
```

  There are a couple ways to fix this; one is detailed here.
  有几种方法来处理这种情况；其中一种在 这里有详细介绍。

  Beautiful Soup also ignores an entity reference that's not finished by the end of the document:
  Beautiful Soup 也会忽略实体引用，如果它没有在文档结束的时候关闭：

```
BeautifulSoup("&lt;foo&gt")
# &lt;foo
```

  I've never seen this in real web pages, but it's probably out there somewhere. 我从来没有在实际的网页中遇到这种情况，但是也许别的地方会出现。

- A malformed comment will make Beautiful Soup ignore the rest of the document. This is covered as the example in Sanitizing Bad Data with Regexps.
一个畸形的注释会是Beautiful Soup回来文档的剩余部分。在使用正则规范数据这里有详细的例子。

The parse tree built by the BeautifulSoup class offends my senses!
BeautifulSoup类构建的剖析树让我感到头痛。

To get your markup parsed differently, check out
尝试一下别的剖析方法，试试 其他内置的剖析器,或者 自定义一个剖析器.

Beautiful Soup 太慢了!

Beautiful Soup will never run as fast as ElementTree or a custom-built SGMLParser subclass. ElementTree is written in C, and SGMLParser lets you write your own mini-Beautiful Soup that only does what you want. The point of Beautiful Soup is to save programmer time, not processor time.
Beautiful Soup 不会像ElementTree或者自定义的SGMLParser子类一样快。 ElementTree是用C写的，并且做那些你想要做的事。 Beautiful Soup是用来节省程序员的时间，而不是处理器的时间。

That said, you can speed up Beautiful Soup quite a lot by only parsing the parts of the document you need, and you can make unneeded objects get garbage-collected by using extract.
但是你可以加快Beautiful Soup通过解析部分的文档,

高级主题

That does it for the basic usage of Beautiful Soup. But HTML and XML are tricky, and in the real world they're even trickier. So Beautiful Soup keeps some extra tricks of its own up its sleeve.
那些是对Beautiful Soup的基本用法。但是现实中的HTML和XML是非常棘手的(tricky)，即使他们不是trickier。 因此Beautiful Soup也有一些额外的技巧。

产生器

The search methods described above are driven by generator methods. You can use these methods yourself: they're called nextGenerator, previousGenerator, nextSiblingGenerator, previousSiblingGenerator, and parentGenerator. Tag and parser objects also have childGenerator and recursiveChildGenerator available.
以上的搜索方法都是由产生器驱动的。你也可以自己使用这些方法： 他们是nextGenerator, previousGenerator, nextSiblingGenerator, previousSiblingGenerator, 和parentGenerator. Tag和剖析对象 可以使用childGenerator和recursiveChildGenerator。

Here's a simple example that strips HTML tags out of a document by iterating over the document and collecting all the strings.
下面是一个简单的例子,将遍历HTML的标签并将它们从文档中剥离，搜集所有的字符串:

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("""<div>You <i>bet</i>
<a href="http://www.crummy.com/software/BeautifulSoup/">BeautifulSoup</a>
rocks!</div>""")
''.join([e for e in soup.recursiveChildGenerator()
         if isinstance(e,unicode)])
# u'You bet\nBeautifulSoup\nrocks!'
```

Here's a more complex example that uses recursiveChildGenerator to iterate over the elements of a document, printing each one as it gets it. 这是一个稍微复杂点的使用recursiveChildGenerator 的例子来遍历文档中所有元素， 并打印它们。

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup("1<a>2<b>3")
g = soup.recursiveChildGenerator()
while True:
    try:
        print g.next()
    except StopIteration:
        break
# 1
# <a>2<b>3</b></a>
# 2
# <b>3</b>
# 3
```

其它内置的剖析器

Beautiful Soup comes with three parser classes besides BeautifulSoup and BeautifulStoneSoup:
除了BeautifulSoup和 BeautifulStoneSoup，还有其它三个Beautiful Soup剖析器:

- MinimalSoup is a subclass of BeautifulSoup. It knows most facts about HTML like which tags are self-closing, the special behavior of the <SCRIPT> tag, the possibility of an encoding mentioned in a <META> tag, etc. But it has no nesting heuristics at all. So it doesn't know that <LI> tags go underneath <UL> tags and not the other way around. It's useful for parsing pathologically bad markup, and for subclassing.
  MinimalSoup是BeautifulSoup的子类。对于HTML的大部分内容都可以处理， 例如自关闭的标签，特殊的标签<SCRIPT>,<META>中写到的可能的编码类型，等等。 但是它没有内置的智能判断能力。例如它不知道<LI>标签应该在<UL>下，而不是其他方式。 对于处理糟糕的标记和用来被继承还是有用的。

- ICantBelieveItsBeautifulSoup is also a subclass of BeautifulSoup. It has HTML heuristics that conform more closely to the HTML standard, but ignore how HTML is used in the real world. For instance, it's valid HTML to nest <B> tags, but in the real world a nested <B> tag almost always means that the author forgot to close the first <B> tag. If you run into someone who actually nests <B> tags, then you can use ICantBelieveItsBeautifulSoup. ICantBelieveItsBeautifulSoup 也是BeautifulSoup的子类。 它具有HTML的智能(heuristics)判断能力，更加符合标准的HTML，但是忽略实际使用的HTML。 例如：一个嵌入<B>标签的HTML是有效的，但是实际上一个嵌入的<B>通常意味着 那个HTML的作者忘记了关闭第一个<B>标签。如果你运行某些人确实使用嵌入的<B>标签的HTML， 这是你可以是使用ICantBelieveItsBeautifulSoup。

- BeautifulSOAP is a subclass of BeautifulStoneSoup. It's useful for parsing documents like SOAP messages, which use a subelement when they could just use an attribute of the parent element. Here's an example:
  BeautifulSOAP是BeautifulStoneSoup的子类。对于处理那些类似SOAP消息的文档， 也就是处理那些可以将标签的子标签变为其属性的文档很方便。下面是一个例子：

  ```
  from BeautifulSoup import BeautifulStoneSoup, BeautifulSOAP
  xml = "<doc><tag>subelement</tag></doc>"
  print BeautifulStoneSoup(xml)
  # <doc><tag>subelement</tag></doc>
  print BeautifulSOAP(xml)
  <doc tag="subelement"><tag>subelement</tag></doc>
  ```

  With BeautifulSOAP you can access the contents of the <TAG> tag without descending into the tag.
  使用BeautifulSOAP,你可以直接存取<TAG>而不需要再往下解析。

定制剖析器(Parser)

When the built-in parser classes won't do the job, you need to customize. This usually means customizing the lists of nestable and self-closing tags. You can customize the list of self-closing tags by passing a selfClosingTags argument into the soup constructor. To customize the lists of nestable tags, though, you'll have to subclass.
当内置的剖析类不能做一些工作时，你需要定制它们。 这通常意味着重新定义可内嵌的标签和自关闭的标签列表。 你可以通过传递参数selfClosingTags 给soup的构造器来定制自关闭的标签。自定义可以内嵌的标签的列表，你需要子类化。

The most useful classes to subclass are MinimalSoup (for HTML) and BeautifulStoneSoup (for XML). I'm going to show you how to override RESET_NESTING_TAGS and NESTABLE_TAGS in a subclass. This is the most complicated part of Beautiful Soup and I'm not going to explain it very well here, but I'll get something written and then I can improve it with feedback.
非常有用的用来子类的类是MinimalSoup类(针对HTML)和BeautifulStoneSoup(针对XML)。 我会说明如何在子类中重写RESET_NESTING_TAGS和NESTABLE_TAGS。这是Beautiful Soup 中 最为复杂的部分，所以我也不会在这里详细的解释，但是我会写些东西并利用反馈来改进它。

When Beautiful Soup is parsing a document, and it finds a start tag, it tosses that tag on top of the stack. But before it does, it might close some of the open tags and remove them from the stack. Which tags it closes depends on the qualities of tag it just found, and the qualities of the tags in the stack.
当Beautiful Soup剖析一个文档的时候，它会保持一个打开的tag的堆栈。任何时候只要它看到一个新的 开始tag，它会将这个tag拖到堆栈的顶端。但在做这步之前，它可能会关闭某些已经打开的标签并将它们从 堆栈中移除。

The best way to explain it is through example. Let's say the stack looks like ['html', 'p', 'b'], and Beautiful Soup encounters a <P> tag. If it just tossed another 'p' onto the stack, this would imply that the second <P> tag is within the first <P> tag, not to mention the open <B> tag. But that's not the way <P> tags work. You can't stick a <P> tag inside another <P> tag. A <P> tag isn't "nestable" at all.
我们最好还是通过例子来解释。我们假定堆栈如同['html','p','b']， 并且Beautiful Soup遇到一个<P>标签。如果它仅仅将另一个'p'拖到堆栈的顶端， 这意味着第二个<P>标签在第一个<P>内，而不会影响到打开的<B>。 但是这不是<P>应该的样子。你不能插入一个<P>到另一个<P>里面去。<P>标签不是可内嵌的。

So when Beautiful Soup encounters a <P> tag, it closes and pops all the tags up to and including the previously encountered tag of the same type. This is the default behavior, and this is how BeautifulStoneSoup treats *every* tag. It's what you get when a tag is not mentioned in either NESTABLE_TAGS or RESET_NESTING_TAGS. It's also what you get when a tag shows up in RESET_NESTING_TAGS but has no entry in NESTABLE_TAGS, the way the <P> tag does.
因此当Beautiful Soup 遇到一个<P>时，它先关闭并弹出所有的标签，包括前面遇到的同类型的标签。 这是默认的操作，这也是Beautiful Soup对待每个标签的方式。当一个标签不在NESTABLE_TAGS 或RESET_NESTING_TAGS中时，你会遇到的处理方式。这也是当一个标签在RESET_NESTING_TAGS 中而不在NESTABLE_TAGS中时的处理方式，就像处理<P>一样。

```
from BeautifulSoup import BeautifulSoup
BeautifulSoup.RESET_NESTING_TAGS['p'] == None
# True
BeautifulSoup.NESTABLE_TAGS.has_key('p')
# False
print BeautifulSoup("<html><p>Para<b>one<p>Para two")
# <html><p>Para<b>one</b></p><p>Para two</p></html>
#                        ^--- --The second <p> tag made those two tags get closed
```

Let's say the stack looks like ['html', 'span', 'b'], and Beautiful Soup encounters a <SPAN> tag. Now, <SPAN> tags can contain other <SPAN> tags without limit, so there's no need to pop up to the previous <SPAN> tag when you encounter one. This is represented by mapping the tag name to an empty list in NESTABLE_TAGS. This kind of tag should not be mentioned in RESET_NESTING_TAGS: there are no circumstances when encountering a <SPAN> tag would cause any tags to be popped.
我们假定堆栈如同['html','span','b']，并且Beautiful Soup 遇到一个<SPAN>标签。 现在，<SPAN>可以无限制包含其他的<SPAN>，因此当再次遇到<SPAN>标签时没有必要弹出前面的<SPAN>标签。 这是通过映射标签名到NESTABLE_TAGS中的一个空列表里。这样的标签也需要在RESET_NESTING_TAGS里 设置：当再次遇到<SPAN>是不会再导致任何标签被弹出并关闭。

```
from BeautifulSoup import BeautifulSoup
BeautifulSoup.NESTABLE_TAGS['span']
# []
BeautifulSoup.RESET_NESTING_TAGS.has_key('span')
# False
print BeautifulSoup("<html><span>Span<b>one<span>Span two")
# <html><span>Span<b>one<span>Span two</span></b></span></html>
```

Third example: suppose the stack looks like ['ol','li','ul']: that is, we've got an ordered list, the first element of which contains an unordered list. Now suppose Beautiful Soup encounters a <LI> tag. It shouldn't pop up to the first <LI> tag, because this new <LI> tag is part of the unordered sublist. It's okay for an <LI> tag to be inside another <LI> tag, so long as there's a <UL> or <OL> tag in the way.
第三个例子:假定堆栈如同['ol','li','ul']: 也就是，我们有一个有序的list，且列表的第一个元素包含一个无序的list。现在假设，Beautiful Soup 遇到一个<LI>标签。它不会弹出第一个<LI>，因为这个新的<LI>是无序的子list一部分。 <LI>中内嵌一个<LI>是可以的，同样的<UL>和<OL>标签也可以这样。

```
from BeautifulSoup import BeautifulSoup
print BeautifulSoup("<ol><li>1<ul><li>A").prettify()
# <ol>
#  <li>
#   1
#   <ul>
#    <li>
#     A
#    </li>
#   </ul>
#  </li>
# </ol>
```

But if there is no intervening <UL> or <OL>, then one <LI> tag can't be underneath another:
如果<UL>和<OL>没有被干扰，这时一个<LI>标签也不能在另一个之下。[bad]

```
print BeautifulSoup("<ol><li>1<li>A").prettify()
# <ol>
#  <li>
#   1
#  </li>
#  <li>
#   A
#  </li>
# </ol>
```

We tell Beautiful Soup to treat <LI> tags this way by putting "li" in RESET_NESTING_TAGS, and by giving "li" a NESTABLE_TAGS entry showing list of tags under which it can nest.
Beautiful Soup这样对待<LI>是通过将"li"放入RESET_NESTING_TAGS，并给在NESTABLE_TAGS中给"li"一个可以内嵌接口。

```
BeautifulSoup.RESET_NESTING_TAGS.has_key('li')
# True
BeautifulSoup.NESTABLE_TAGS['li']
# ['ul', 'ol']
```

This is also how we handle the nesting of table tags:
这也是处理内嵌的table标签的方式：

```
BeautifulSoup.NESTABLE_TAGS['td']
# ['tr']
BeautifulSoup.NESTABLE_TAGS['tr']
# ['table', 'tbody', 'tfoot', 'thead']
BeautifulSoup.NESTABLE_TAGS['tbody']
# ['table']
BeautifulSoup.NESTABLE_TAGS['thead']
# ['table']
BeautifulSoup.NESTABLE_TAGS['tfoot']
# ['table']
BeautifulSoup.NESTABLE_TAGS['table']
# []
```

That is: <TD> tags can be nested within <TR> tags. <TR> tags can be nested within <TABLE>, <TBODY>, <TFOOT>, and <THEAD> tags. <TBODY>, <TFOOT>, and <THEAD> tags can be nested in <TABLE> tags, and <TABLE> tags can be nested in other <TABLE> tags. If you know about HTML tables, these rules should already make sense to you.
也就是<TD>标签可以嵌入到<TR>中。 <TR>可以被嵌入到<TABLE>, <TBODY>, <TFOOT>, 以及 <THEAD> 中。 <TBODY>,<TFOOT>, and <THEAD>标签可以嵌入到<TABLE> 标签中, 而 <TABLE> 嵌入到其它的<TABLE> 标签中. 如果你对HTML有所了解，这些规则对你而言应该很熟悉。

One more example. Say the stack looks like ['html', 'p', 'table'] and Beautiful Soup encounters a <P> tag.
再举一个例子，假设堆栈如同['html','p','table']，并且Beautiful Soup遇到一个<P>标签。

At first glance, this looks just like the example where the stack is ['html', 'p', 'b'] and Beautiful Soup encounters a <P> tag. In that example, we closed the <B> and <P> tags, because you can't have one paragraph inside another. 首先，这看起来像前面的同样是Beautiful Soup遇到了堆栈['html','p','b']。 在那个例子中，我们关闭了<B>和<P>标签，因为你不能在一个段落里内嵌另一个段落。

Except... you *can* have a paragraph that contains a table, and then the table contains a paragraph. So the right thing to do is to not close any of these tags. Beautiful Soup does the right thing: 除非，你的段落里包含了一个table，然后这table包含了一个段落。因此，这种情况下正确的处理是 不关闭任何标签。Beautiful Soup就是这样做的：

```
from BeautifulSoup import BeautifulSoup
print BeautifulSoup("<p>Para 1<b><p>Para 2")
# <p>
#  Para 1
#  <b>
#  </b>
# </p>
# <p>
#  Para 2
# </p>
print BeautifulSoup("<p>Para 1<table><p>Para 2").prettify()
# <p>
#  Para 1
#  <table>
#   <p>
#    Para 2
#   </p>
#  </table>
# </p>
```

What's the difference? The difference is that <TABLE> is in RESET_NESTING_TAGS and <B> is not. A tag that's in RESET_NESTING_TAGS doesn't get popped off the stack as easily as a tag that's not.
有什么不同？不同是<TABLE>标签在RESET_NESTING_TAGS中，而<B>不在。 一个在RESET_NESTING_TAGS中标签不会像不在其里面的标签那样，会是堆栈中标签被弹出。

Okay, hopefully you get the idea. Here's the NESTABLE_TAGS for the BeautifulSoup class. Correlate this with what you know about HTML, and you should be able to create your own NESTABLE_TAGS for bizarre HTML documents that don't follow the normal rules, and for other XML dialects that have different nesting rules.
好了，希望你明白了(我被弄有点晕，有些地方翻译的不清，还请见谅)。 NESTABLE_TAGS用于BeautifulSoup类。 依据你所知道的HTML，你可以创建你自己NESTABLE_TAGS来处理那些不遵循标准规则的HTML文档。 以及那些使用不同嵌入规则XML的方言。

```
from BeautifulSoup import BeautifulSoup
nestKeys = BeautifulSoup.NESTABLE_TAGS.keys()
nestKeys.sort()
for key in nestKeys:
    print "%s: %s" % (key, BeautifulSoup.NESTABLE_TAGS[key])
# bdo: []
# blockquote: []
# center: []
# dd: ['dl']
# del: []
# div: []
# dl: []
# dt: ['dl']
# fieldset: []
# font: []
# ins: []
# li: ['ul', 'ol']
# object: []
# ol: []
# q: []
# span: []
# sub: []
# sup: []
# table: []
# tbody: ['table']
# td: ['tr']
# tfoot: ['table']
# th: ['tr']
# thead: ['table']
# tr: ['table', 'tbody', 'tfoot', 'thead']
# ul: []
```

And here's BeautifulSoup's RESET_NESTING_TAGS. Only the keys are important: RESET_NESTING_TAGS is a list, put into the form of a dictionary for quick random access.
这是BeautifulSoup的RESET_NESTING_TAGS。只有键(keys)是重要的： RESET_NESTING_TAGS实际是一个list，以字典的形式可以快速随机存取。

```
from BeautifulSoup import BeautifulSoup
resetKeys = BeautifulSoup.RESET_NESTING_TAGS.keys()
resetKeys.sort()
resetKeys
# ['address', 'blockquote', 'dd', 'del', 'div', 'dl', 'dt', 'fieldset',
#  'form', 'ins', 'li', 'noscript', 'ol', 'p', 'pre', 'table', 'tbody',
#  'td', 'tfoot', 'th', 'thead', 'tr', 'ul']
```

Since you're subclassing anyway, you might as well override SELF_CLOSING_TAGS while you're at it. It's a dictionary that maps self-closing tag names to any values at all (like RESET_NESTING_TAGS, it's actually a list in the form of a dictionary). Then you won't have to pass that list in to the constructor (as selfClosingTags) every time you instantiate your subclass.
因为无论如何都有使用继承，你最好还是在需要的时候重写SELF_CLOSING_TAGS。 这是一个映射自关闭标签名的字典(如同RESET_NESTING_TAGS,它实际是字典形式的list)。 这样每次实例化你的子类时，你就不用传list给构造器(如selfClosingTags)。

实体转换

When you parse a document, you can convert HTML or XML entity references to the corresponding Unicode characters. This code converts the HTML entity "é" to the Unicode character LATIN SMALL LETTER E WITH ACUTE, and the numeric entity "e" to the Unicode character LATIN SMALL LETTER E.
当你剖析一个文档是，你可以转换HTML或者XML实体引用到可表达Unicode的字符。 这个代码转换HTML实体"é"到Unicode字符 LATIN SMALL LETTER E WITH ACUTE,以及将 数量实体"e"转换到Unicode字符LATIN SMALL LETTER E.

```
from BeautifulSoup import BeautifulStoneSoup
BeautifulStoneSoup("Sacr&eacute; bl&#101;u!",
                   convertEntities=BeautifulStoneSoup.HTML_ENTITIES).contents[0]
# u'Sacr\xe9 bleu!'
```

That's if you use HTML_ENTITIES (which is just the string "html"). If you use XML_ENTITIES (or the string "xml"), then only numeric entities and the five XML entities (""", "'", ">", "<", and "&") get converted. If you use ALL_ENTITIES (or the list ["xml", "html"]), then both kinds of entities will be converted. This last one is neccessary because ' is an XML entity but not an HTML entity.
这是针对使用HTML_ENTITIES(也就是字符串"html")。如果你使用XML_ENTITIES(或字符串"xml"), 这是只有数字实体和五个XML实体((""","'", ">", "<", 和 "&") 会被转换。如果你使用ALL_ENTITIES(或者列表["xml","html"]), 两种实体都会被转换。最后一种方式是必要的，因为'是一个XML的实体而不是HTML的。

```
BeautifulStoneSoup("Sacr&eacute; bl&#101;u!",
                   convertEntities=BeautifulStoneSoup.XML_ENTITIES)
# Sacr&eacute; bleu!
from BeautifulSoup import BeautifulStoneSoup
BeautifulStoneSoup("Il a dit, &lt;&lt;Sacr&eacute; bl&#101;u!&gt;&gt;",
                   convertEntities=BeautifulStoneSoup.XML_ENTITIES)
# Il a dit, <<Sacr&eacute; bleu!>>
```

If you tell Beautiful Soup to convert XML or HTML entities into the corresponding Unicode characters, then Windows-1252 characters (like Microsoft smart quotes) also get transformed into Unicode characters. This happens even if you told Beautiful Soup to convert those characters to entities.
如果你指定Beautiful Soup转换XML或HTML实体到可通信的Unicode字符时，Windows-1252(微软的smart quotes)也会 被转换为Unicode字符。即使你指定Beautiful Soup转换这些字符到实体是，也还是这样。

```
from BeautifulSoup import BeautifulStoneSoup
smartQuotesAndEntities = "Il a dit, \x8BSacr&eacute; bl&#101;u!\x9b"
BeautifulStoneSoup(smartQuotesAndEntities, smartQuotesTo="html").contents[0]
# u'Il a dit, &lsaquo;Sacr&eacute; bl&#101;u!&rsaquo;'
BeautifulStoneSoup(smartQuotesAndEntities, convertEntities="html",
                   smartQuotesTo="html").contents[0]
# u'Il a dit, \u2039Sacr\xe9 bleu!\u203a'
BeautifulStoneSoup(smartQuotesAndEntities, convertEntities="xml",
                   smartQuotesTo="xml").contents[0]
# u'Il a dit, \u2039Sacr&eacute; bleu!\u203a'
```

It doesn't make sense to create new HTML/XML entities while you're busy turning all the existing entities into Unicode characters.
将所有存在的实体转换为Unicode时，不会影响创建新的HTML/XML实体。

使用正则式处理糟糕的数据

Beautiful Soup does pretty well at handling bad markup when "bad markup" means tags in the wrong places. But sometimes the markup is just malformed, and the underlying parser can't handle it. So Beautiful Soup runs regular expressions against an input document before trying to parse it.
对于那些在错误的位置的"坏标签"，Beautiful Soup处理的还不错。但有时有些 非常不正常的标签，底层的剖析器也不能处理。这时Beautiful Soup会在剖析之前运用正则表达式 来处理输入的文档。

By default, Beautiful Soup uses regular expressions and replacement functions to do search-and-replace on input documents. It finds self-closing tags that look like
, and changes them to look like
. It finds declarations that have extraneous whitespace, like <! --Comment-->, and removes the whitespace: <!--Comment-->.
默认情况下，Beautiful Soup使用正则式和替换函数对输入文档进行搜索替换操作。 它可以发现自关闭的标签如
，转换它们如同
（译注：多加了一个空格）。 它可以找到有多余空格的声明，如<! --Comment-->，移除空格:<!--Comment-->.

If you have bad markup that needs fixing in some other way, you can pass your own list of (regular expression, replacement function) tuples into the soup constructor, as the markupMassage argument.
如果你的坏标签需要以其他的方式修复，你也可以传递你自己的以(regular expression, replacement function) 元组的list到soup对象构造器，作为markupMassage参数。

Let's take an example: a page that has a malformed comment. The underlying SGML parser can't cope with this, and ignores the comment and everything afterwards: 我们举个例子：有一个页面的注释很糟糕。底层的SGML不能解析它，并会忽略注释以及它后面的所有内容。

```
from BeautifulSoup import BeautifulSoup
badString = "Foo<!-This comment is malformed.-->Bar<br/>Baz"
BeautifulSoup(badString)
# Foo
```

Let's fix it up with a regular expression and a function:
让我们使用正则式和一个函数来解决这个问题：

```
import re
myMassage = [(re.compile('<!-([^-])'), lambda match: '<!--' + match.group(1))]
BeautifulSoup(badString, markupMassage=myMassage)
# Foo<!--This comment is malformed.-->Bar
```

Oops, we're still missing the
 tag. Our markupMassage overrides the parser's default massage, so the default search-and-replace functions don't get run. The parser makes it past the comment, but it dies at the malformed self-closing tag. Let's add our new massage function to the default list, so we run all the functions.
哦呃呃，我们还是漏掉了
标签。我们的markupMassage 重载了剖析默认的message，因此默认的搜索替换函数不会运行。 剖析器让它来处理注释，但是它在坏的自关闭标签那里停止了。让我加一些新的message函数到默认的list中去， 并让这些函数都运行起来。

```
import copy
myNewMassage = copy.copy(BeautifulSoup.MARKUP_MASSAGE)
myNewMassage.extend(myMassage)
BeautifulSoup(badString, markupMassage=myNewMassage)
# Foo<!--This comment is malformed.-->Bar<br />Baz
```

Now we've got it all.
这样我们就搞定了。

If you know for a fact that your markup doesn't need any regular expressions run on it, you can get a faster startup time by passing in False for markupMassage.
如果你已经知道你的标签不需要任何正则式，你可以通过传递一个False给markupMassage.

玩玩SoupStrainer

Recall that all the search methods take more or less the same arguments. Behind the scenes, your arguments to a search method get transformed into a SoupStrainer object. If you call one of the methods that returns a list (like findAll), the SoupStrainer object is made available as the source property of the resulting list.
回忆起所有的搜索方法都是或多或少使用了一些一样的参数。 在后台，你传递给搜索函数的参数都会传给SoupStrainer对象。 如果你所使用的函数返回一个list(如findAll)，那是SoupStrainer对象使 结果列表的source属性变的可用。

```
from BeautifulSoup import BeautifulStoneSoup
xml = '<person name="Bob"><parent rel="mother" name="Alice">'
xmlSoup = BeautifulStoneSoup(xml)
results = xmlSoup.findAll(rel='mother')
results.source
# <BeautifulSoup.SoupStrainer instance at 0xb7e0158c>
str(results.source)
# "None|{'rel': 'mother'}"
```

The SoupStrainer constructor takes most of the same arguments as find: name, attrs, text, and **kwargs. You can pass in a SoupStrainer as the name argument to any search method:
SoupStrainer的构造器几乎使用和find一样的参数： name, attrs, text, 和**kwargs. 你可以在一个SoupStrainer中传递和其他搜索方法一样的name参数：

```
xmlSoup.findAll(results.source) == results
# True
customStrainer = BeautifulSoup.SoupStrainer(rel='mother')
xmlSoup.findAll(customStrainer) == results
# True
```

Yeah, who cares, right? You can carry around a method call's arguments in many other ways. But another thing you can do with SoupStrainer is pass it into the soup constructor to restrict the parts of the document that actually get parsed. That brings us to the next section:
耶,谁会在意，对不对？你可以把一个方法的参数用在很多其他地方。 还有一件你可以用SoupStrainer做的事是，将它传递给soup的构建器，来部分的解析文档。下一节，我们就谈这个。

通过剖析部分文档来提升效率

Beautiful Soup turns every element of a document into a Python object and connects it to a bunch of other Python objects. If you only need a subset of the document, this is really slow. But you can pass in a SoupStrainer as the parseOnlyThese argument to the soup constructor. Beautiful Soup checks each element against the SoupStrainer, and only if it matches is the element turned into a Tag or NavigableText, and added to the tree.
Beautiful Soup 将一个文档的每个元素都转换为Python对象并将文档转换为一些Python对象的集合。 如果你只需要这个文档的子集，全部转换确实非常慢。 但是你可以传递SoupStrainer作为parseOnlyThese参数的值给 soup的构造器。Beautiful Soup检查每一个元素是否满足SoupStrainer条件， 只有那些满足条件的元素会转换为Tag标签或NavigableText，并被添加到剖析树中。

If an element is added to to the tree, then so are its children—even if they wouldn't have matched the SoupStrainer on their own. This lets you parse only the chunks of a document that contain the data you want.
如果一个元素被加到剖析树中，那么的子元素即使不满足SoupStrainer也会被加入到树中。 这可以让你只剖析文档中那些你想要的数据块。

Here's a pretty varied document:
看看下面这个有意思的例子：

```
doc = '''Bob reports <a href="http://www.bob.com/">success</a>
with his plasma breeding <a
href="http://www.bob.com/plasma">experiments</a>. <i>Don't get any on
us, Bob!</i>
<br><br>Ever hear of annular fusion? The folks at <a
```

```
href="http://www.boogabooga.net/">BoogaBooga</a>, <a href="http://www.bob.com/plasma">experiments</a>
with it. Secret project, or <b>WEB MADNESS?</b> You decide!'''
```

Here are several different ways of parsing the document into soup, depending on which parts you want. All of these are faster and use less memory than parsing the whole document and then using the same SoupStrainer to pick out the parts you want.
有几种不同的方法可以根据你的需求来剖析部分文档.比起剖析全部文档，他们都更快并占用更少的内存，他们都是使用相同的 SoupStrainer来挑选文档中你想要的部分。

```
from BeautifulSoup import BeautifulSoup, SoupStrainer
import re
links = SoupStrainer('a')
[tag for tag in BeautifulSoup(doc, parseOnlyThese=links)]
# [<a href="http://www.bob.com/">success</a>,
#  <a href="http://www.bob.com/plasma">experiments</a>,
#  <a href="http://www.boogabooga.net/">BoogaBooga</a>]
linksToBob = SoupStrainer('a', href=re.compile('bob.com/'))
[tag for tag in BeautifulSoup(doc, parseOnlyThese=linksToBob)]
# [<a href="http://www.bob.com/">success</a>,
#  <a href="http://www.bob.com/plasma">experiments</a>]
mentionsOfBob = SoupStrainer(text=re.compile("Bob"))
[text for text in BeautifulSoup(doc, parseOnlyThese=mentionsOfBob)]
# [u'Bob reports ', u"Don't get any on\nus, Bob!"]
allCaps = SoupStrainer(text=lambda(t):t.upper()==t)
[text for text in BeautifulSoup(doc, parseOnlyThese=allCaps)]
# [u'. ', u'\n', u'WEB MADNESS?']
```

There is one major difference between the SoupStrainer you pass into a search method and the one you pass into a soup constructor. Recall that the name argument can take a function whose argument is a Tag object. You can't do this for a SoupStrainer's name, because the SoupStrainer is used to decide whether or not a Tag object should be created in the first place. You can pass in a function for a SoupStrainer's name, but it can't take a Tag object: it can only take the tag name and a map of arguments.
把SoupStrainer传递给搜索方法和soup构造器有一个很大的不同。 回忆一下，name参数可以使用以Tag对象为参数的函数。 但是你不能对SoupStrainer的name使用这招，因为SoupStrainer被用于决定 一个Tag对象是否可以在第一个地方被创建。 你可以传递一个函数给SoupStrainer的name，但是不能是使用Tag对象的函数： 只能使用tag的名字和一个参数映射。

```
shortWithNoAttrs = SoupStrainer(lambda name, attrs: \
                                len(name) == 1 and not attrs)
[tag for tag in BeautifulSoup(doc, parseOnlyThese=shortWithNoAttrs)]
# [<i>Don't get any on us, Bob!</i>,
#  <b>WEB MADNESS?</b>]
```

使用extract改进内存使用

When Beautiful Soup parses a document, it loads into memory a large, densely connected data structure. If you just need a string from that data structure, you might think that you can grab the string and leave the rest of it to be garbage collected. Not so. That string is a NavigableString object. It's got a parent member that points to a Tag object, which points to other Tag objects, and so on. So long as you hold on to any part of the tree, you're keeping the whole thing in memory.
但Beautiful Soup剖析一个文档的时候，它会将整个文档以一个很大很密集的数据结构中载入内存。 如果你仅仅需要从这个数据结构中获得一个字符串， 你可能觉得为了这个字符串而弄了那么一堆要被当垃圾收集的数据会很不划算。 而且，那个字符串还是NavigableString对象。 也就是要获得一个指向Tag对象的parent的成员，而这个Tag又会指向其他的Tag对象，等等。 因此，你不得不保持一颗剖析树所有部分，也就是把整个东西放在内存里。

The extract method breaks those connections. If you call extract on the string you need, it gets disconnected from the rest of the parse tree. The rest of the tree can then go out of scope and be garbage collected, while you use the string for something else. If you just need a small part of the tree, you can call extract on its top-level Tag and let the rest of the tree get garbage collected.
extrace方法可以破坏这些链接。如果你调用extract来获得你需要字符串， 它将会从树的其他部分中链接中断开。 当你使用这个字符串做什么时,树的剩下部分可以离开作用域而被垃圾收集器捕获。 如果你即使需要一个树的一部分，你也可以讲extract使用在顶层的Tag上， 让其它部分被垃圾收集器收集。

This works the other way, too. If there's a big chunk of the document you *don't* need, you can call extract to rip it out of the tree, then abandon it to be garbage collected while retaining control of the (smaller) tree. 也可以使用extract实现些别的功能。如果文档中有一大块不是你需要，你也可以使用extract来将它弄出剖析树， 再把它丢给垃圾收集器同时对(较小的那个)剖析树的控制。

If you find yourself destroying big chunks of the tree, you might have been able to save time by not parsing that part of the tree in the first place.
如果你觉得你正在破坏树的大块头，你应该看看 通过剖析部分文档来提升效率来省省时间。

其它

Applications that use Beautiful Soup

Lots of real-world applications use Beautiful Soup. Here are the publicly visible applications that I know about:
很多实际的应用程序已经使用Beautiful Soup。 这里是一些我了解的公布的应用程序：

- Scrape 'N' Feed is designed to work with Beautiful Soup to build RSS feeds for sites that don't have them.
- htmlatex uses Beautiful Soup to find LaTeX equations and render them as graphics.
- chmtopdf converts CHM files to PDF format. Who am I to argue with that?
- Duncan Gough's Fotopic backup uses Beautiful Soup to scrape the Fotopic website.
- Iñigo Serna's googlenews.py uses Beautiful Soup to scrape Google News (it's in the parse_entry and parse_category functions)
- The Weather Office Screen Scraper uses Beautiful Soup to scrape the Canadian government's weather office site.
- News Clues uses Beautiful Soup to parse RSS feeds.
- BlinkFlash uses Beautiful Soup to automate form submission for an online service.
- The linky link checker uses Beautiful Soup to find a page's links and images that need checking.
- Matt Croydon got Beautiful Soup 1.x to work on his Nokia Series 60 smartphone. C.R. Sandeep wrote a real-time currency converter for the Series 60 using Beautiful Soup, but he won't show us how he did it.
- Here's a short script from jacobian.org to fix the metadata on music files downloaded from allofmp3.com.
- The Python Community Server uses Beautiful Soup in its spam detector.

类似的库

I've found several other parsers which use Beautiful Soup, or are otherwise more useful than your average parser.
我已经找了几个其他的用于不同语言的可以处理烂标记的剖析器。简单介绍一下，也许对你有所帮助。

- I've ported Beautiful Soup to Ruby. The result is Rubyful Soup.
- Hpricot is giving Rubyful Soup a run for its money.
- ElementTree is a fast Python XML parser with a bad attitude. I love it.
- Tag Soup is an XML/HTML parser written in Java which rewrites bad HTML into parseable HTML.
- HtmlPrag is a Scheme library for parsing bad HTML.
- xmltramp is a nice take on a 'standard' XML/XHTML parser. Like most parsers, it makes you traverse the tree yourself, but it's easy to use.
- pullparser includes a tree-traversal method.
- Mike Foord didn't like the way Beautiful Soup can change HTML if you write the tree back out, so he wrote HTML Scraper. It's basically a version of HTMLParser that can handle bad HTML. It might be obsolete with the release of Beautiful Soup 3.0, though; I'm not sure.
- Ka-Ping Yee's scrape.py combines page scraping with URL opening.

小结

That's it! Have fun! I wrote Beautiful Soup to save everybody time. Once you get used to it, you should be able to wrangle data out of poorly-designed websites in just a few minutes. Send me email if you have any comments, run into problems, or want me to know about your project that uses Beautiful Soup.
就这样了！玩的开心！我写的Beautiful Soup是为了帮助每个人节省时间。 一旦你习惯上用它之后，只要几分钟就能整好那些有些糟糕的站点。可以发邮件给我，如果你有什么建议，或者 遇到什么问题，或者让我知道你的项目再用Beautiful Soup。

--Leonard

Document tree:

http://www.crummy.com/

software/

BeautifulSoup/

documentation.zh.html

Site Search: