Beautiful Soup Documentation

by Leonard Richardson (leonardr@segfault.org)

这份文档也有中文版了 (This document is also available in Chinese translation)

Beautiful Soup is an HTML/XML parser for Python that can turn even invalid markup into a parse tree. It provides simple, idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work. There's also a Ruby port called Rubyful Soup.

This document illustrates all major features of Beautiful Soup version 3.0, with examples. It shows you what the library is good for, how it works, how to use it, how to make it do what you want, and what to do when it violates your expectations.

Table of Contents

Quick Start

Get Beautiful Soup here. The changelog describes differences between 3.0 and earlier versions.

Include Beautiful Soup in your application with a line like one of the following:

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

Here's some code demonstrating the basic features of Beautiful Soup. You can copy and paste this code into a Python session to run it yourself.

```
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>
```

Here are some ways to navigate the soup:

```
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>
```

Here are a couple of ways to search the soup for certain tags, or tags with certain properties:

```
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'
```

It's easy to modify the soup:

```
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>
```

Here's a real-world example. It fetches the ICC Commercial Crime Services weekly piracy report, parses it with Beautiful Soup, and pulls out the piracy incidents:

```
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
```

Parsing a Document

A Beautiful Soup constructor takes an XML or HTML document in the form of a string (or an open file-like object). It parses the document and creates a corresponding data structure in memory.

If you give Beautiful Soup a perfectly-formed document, the parsed data structure looks just like the original document. But if there's something wrong with the document, Beautiful Soup uses heuristics to figure out a reasonable structure for the data structure.

Parsing HTML

Use the `BeautifulSoup` class to parse an HTML document. Here are some of the things that `BeautifulSoup` knows:

- Some tags can be nested (<BLOCKQUOTE>) and some can't (<P>).
- Table and list tags have a natural nesting order. For instance, <TD> tags go inside <TR> tags, not the other way around.
- The contents of a <SCRIPT> tag should not be parsed as HTML.
- A <META> tag may specify an encoding for the document.

Here it is in action:

```
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>
```

Note that `BeautifulSoup` figured out sensible places to put the closing tags, even though the original document lacked them.

That document isn't valid HTML, but it's not too bad either. Here's a really horrible document. Among other problems, it's got a <FORM> tag that starts outside of a <TABLE> tag and ends inside the <TABLE> tag. (HTML like this was found on a website run by a major web company.)

```
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 handles this document as well:

```
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>
```

The last cell of the table is outside the <TABLE> tag; Beautiful Soup decided to close the <TABLE> tag when it closed the <FORM> tag. The author of the original document probably intended the <FORM> tag to extend to the end of the table, but Beautiful Soup has no way of knowing that. Even in a bizarre case like this, Beautiful Soup parses the invalid document and gives you access to all the data.

Parsing XML

The `BeautifulSoup` class is full of web-browser-like heuristics for divining the intent of HTML authors. But XML doesn't have a fixed tag set, so those heuristics don't apply. So `BeautifulSoup` doesn't do XML very well.

Use the `BeautifulStoneSoup` class to parse XML documents. It's a general class with no special knowledge of any XML dialect and very simple rules about tag nesting: Here it is in action:

```
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>
```

The most common shortcoming of `BeautifulStoneSoup` is that it doesn't know about self-closing tags. HTML has a fixed set of self-closing tags, but with XML it depends on what the DTD says. You can tell `BeautifulStoneSoup` that certain tags are self-closing by passing in their names as the `selfClosingTags` argument to the constructor:

```
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
# </tag>
```

If That Doesn't Work

There are several other parser classes with different heuristics from these two. You can also subclass and customize a parser and give it your own heuristics.

Beautiful Soup Gives You Unicode, Dammit

By the time your document is parsed, it has been transformed into Unicode. Beautiful Soup stores only Unicode strings in its data structures.

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

Here's an example with a Japanese document encoded in UTF-8:

```
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 uses a class called `UnicodeDammit` to detect the encodings of documents you give it and convert them to Unicode, no matter what. If you need to do this for other documents (without using Beautiful Soup to parse them), you can use `UnicodeDammit` by itself. It's heavily based on code from the Universal Feed Parser.

If you're running an older version of Python than 2.4, be sure to download and install `cjkcodecs` and `iconvcodec`, which make Python capable of supporting more codecs, especially CJK codecs. Also install the `chardet` library, for better autodetection.

Beautiful Soup tries the following encodings, in order of priority, to turn your document into Unicode:

- An encoding you pass in as the `fromEncoding` argument to the soup constructor.
- An encoding discovered in the document itself: for instance, in an XML declaration or (for HTML documents) an `http-equiv` META tag. If Beautiful Soup finds this kind of encoding within the document, it parses the document again from the beginning and gives the new encoding a try. The only exception is if you explicitly specified an encoding, and that encoding actually worked: then it will ignore any encoding it finds in the document.
- An encoding sniffed by looking at the first few bytes of the file. If an encoding is detected at this stage, it will be one of the UTF-* encodings, EBCDIC, or ASCII.
- An encoding sniffed by the `chardet` library, if you have it installed.
- UTF-8
- Windows-1252

Beautiful Soup will almost always guess right if it can make a guess at all. But for documents with no declarations and in strange encodings, it will often not be able to guess. It will fall back to Windows-1252, which will probably be wrong. Here's an EUC-JP example where Beautiful Soup guesses the encoding wrong. (Again, because it uses EUC-JP, this example will only work if you are running Python 2.4 or have `cjkcodecs` installed):

```
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!
```

But if you specify the encoding with `fromEncoding`, it parses the document correctly, and can convert it to UTF-8 or back to 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
```

If you give Beautiful Soup a document in the Windows-1252 encoding (or a similar encoding like ISO-8859-1 or ISO-8859-2), Beautiful Soup finds and destroys the document's smart quotes and other Windows-specific characters. Rather than transforming those characters into their Unicode equivalents, Beautiful Soup transforms them into HTML entities (`BeautifulSoup`) or XML entities (`BeautifulStoneSoup`).

To prevent this, you can pass `smartQuotesTo=None` into the soup constructor: then smart quotes will be converted to Unicode like any other native-encoding characters. You can also pass in "xml" or "html" for `smartQuotesTo`, to change the default behavior of `BeautifulSoup` and `BeautifulStoneSoup`.

```
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!'
```

Printing a Document

You can turn a Beautiful Soup document (or any subset of it) into a string with the `str` function, or the `prettify` or `renderContents` methods. You can also use the `unicode` function to get the whole document as a Unicode string.

The `prettify` method adds strategic newlines and spacing to make the structure of the document obvious. It also strips out text nodes that contain only whitespace, which might change the meaning of an XML document. The `str` and `unicode` functions don't strip out text nodes that contain only whitespace, and they don't add any whitespace between nodes either.

Here's an example.

```
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>
```

Note that `str` and `renderContents` give different results when used on a tag within the document. `str` prints a tag and its contents, and `renderContents` only prints the contents.

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

When you call `__str__`, `prettify`, or `renderContents`, you can specify an output encoding. The default encoding (the one used by `str`) is UTF-8. Here's an example that parses an ISO-8851-1 string and then outputs the same string in different encodings:

```
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!'
```

If the original document contained an encoding declaration, then Beautiful Soup rewrites the declaration to mention the new encoding when it converts the document back to a string. This means that if you load an HTML document into `BeautifulSoup` and print it back out, not only should the HTML be cleaned up, but it should be transparently converted to UTF-8.

Here's an HTML example:

```
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!
# </html>
```

Here's an XML example:

```
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!
```

The Parse Tree

So far we've focused on loading documents and writing them back out. Most of the time, though, you're interested in the parse tree: the data structure Beautiful Soup builds as it parses the document.

A parser object (an instance of `BeautifulSoup` or `BeautifulStoneSoup`) is a deeply-nested, well-connected data structure that corresponds to the structure of an XML or HTML document. The parser object contains two other types of objects: `Tag` objects, which correspond to tags like the <TITLE> tag and the <B> tags; and `NavigableString` objects, which correspond to strings like "Page title" and "This is paragraph".

There are also some subclasses of `NavigableString` (`CData`, `Comment`, `Declaration`, and `ProcessingInstruction`), which correspond to special XML constructs. They act like `NavigableString`s, except that when it's time to print them out they have some extra data attached to them. Here's a document that includes a comment:

```
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.-->
```

Now, let's take a closer look at the document used at the beginning of the documentation:

```
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>
```

The attributes of `Tags`

`Tag` and `NavigableString` objects have lots of useful members, most of which are covered in Navigating the Parse Tree and Searching the Parse Tree. However, there's one aspect of `Tag` objects we'll cover here: the attributes.

SGML tags have attributes:. for instance, each of the <P> tags in the example HTML above has an "id" attribute and an "align" attribute. You can access a tag's attributes by treating the `Tag` object as though it were a dictionary:

```
firstPTag, secondPTag = soup.findAll('p')

firstPTag['id']
# u'firstPara'

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

`NavigableString` objects don't have attributes; only `Tag` objects have them.

Navigating the Parse Tree

All `Tag` objects have all of the members listed below (though the actual value of the member may be `None`). `NavigableString` objects have all of them except for `contents` and `string`.

`parent`

In the example above, the parent of the <HEAD> `Tag` is the <HTML> `Tag`. The parent of the <HTML> `Tag` is the `BeautifulSoup` parser object itself. The parent of the parser object is `None`. By following `parent`, you can move up the parse tree:

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

`contents`

With `parent` you move up the parse tree. With `contents` you move down the tree. `contents` is an ordered list of the `Tag` and `NavigableString` objects contained within a page element. Only the top-level parser object and `Tag` objects have `contents`. `NavigableString` objects are just strings and can't contain sub-elements, so they don't have `contents`.

In the example above, the `contents` of the first <P> `Tag` is a list containing a `NavigableString` ("This is paragraph "), a <B> `Tag`, and another `NavigableString` ("."). The `contents` of the <B> `Tag`: a list containing a `NavigableString` ("one").

```
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`

For your convenience, if a tag has only one child node, and that child node is a string, the child node is made available as `tag.string`, as well as `tag.contents[0]`. In the example above, `soup.b.string` is a `NavigableString` representing the Unicode string "one". That's the string contained in the first <B> `Tag` in the parse tree.

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

But `soup.p.string` is `None`, because the first <P> `Tag` in the parse tree has more than one child. `soup.head.string` is also `None`, even though the <HEAD> Tag has only one child, because that child is a `Tag` (the <TITLE> `Tag`), not a `NavigableString`.

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

`nextSibling` and `previousSibling`

These members let you skip to the next or previous thing on the same level of the parse tree. In the document above, the `nextSibling` of the <HEAD> `Tag` is the <BODY> `Tag`, because the <BODY> `Tag` is the next thing directly beneath the <html> `Tag`. The `nextSibling` of the <BODY> tag is `None`, because there's nothing else directly beneath the <HTML> `Tag`.

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

Conversely, the `previousSibling` of the <BODY> `Tag` is the <HEAD> tag, and the `previousSibling` of the <HEAD> `Tag` is `None`:

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

Some more examples: the `nextSibling` of the first <P> `Tag` is the second <P> `Tag`. The `previousSibling` of the <B> `Tag` inside the second <P> `Tag` is the `NavigableString` "This is paragraph". The `previousSibling` of that `NavigableString` is `None`, not anything inside the first <P> `Tag`.

```
soup.p.nextSibling
# <p id="secondpara" align="blah">This is paragraph <b>two</b>.</p>

secondBTag = soup.findAlll('b')[1]
secondBTag.previousSibling
# u'This is paragraph'
secondBTag.previousSibling.previousSibling == None
# True
```

`next` and `previous`

These members let you move through the document elements in the order they were processed by the parser, rather than in the order they appear in the tree. For instance, the `next` of the <HEAD> `Tag` is the <TITLE> `Tag`, not the <BODY> `Tag`. This is because, in the original document, the <TITLE> tag comes immediately after the <HEAD> tag.

```
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`. You usually won't have to use these members, but sometimes it's the easiest way to get to something buried inside the parse tree.

Iterating over a `Tag`

You can iterate over the `contents` of a `Tag` by treating it as a list. This is a useful shortcut. Similarly, to see how many child nodes a `Tag` has, you can call `len(tag)` instead of `len(tag.contents)`. In terms of the document above:

```
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
```

Using tag names as members

It's easy to navigate the parse tree by acting as though the name of the tag you want is a member of a parser or `Tag` object. We've been doing it throughout these examples. In terms of the document above, `soup.head` gives us the first (and, as it happens, only) <HEAD> `Tag` in the document:

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

In general, calling `mytag.foo` returns the first child of `mytag` that happens to be a <FOO> `Tag`. If there aren't any <FOO> `Tag`s beneath `mytag`, then `mytag.foo` returns `None`. You can use this to traverse the parse tree very quickly:

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

soup.body.p.b.string
# u'one'
```

You can also use this to quickly jump to a certain part of a parse tree. For instance, if you're not worried about <TITLE> tags in weird places outside of the <HEAD> tag, you can just use `soup.title` to get an HTML document's title. You don't have to use `soup.head.title`:

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

`soup.p` jumps to the first <P> tag inside a document, wherever it is. `soup.table.tr.td` jumps to the first column of the first row of the first table in the document.

These members actually alias to the `first` method, covered below. I mention it here because the alias makes it very easy to zoom in on an interesting part of a well-known parse tree.

An alternate form of this idiom lets you access the first <FOO> tag as `.fooTag` instead of `.foo`. For instance, `soup.table.tr.td` could also be expressed as `soup.tableTag.trTag.tdTag`, or even `soup.tableTag.tr.tdTag`. This is useful if you like to be more explicit about what you're doing, or if you're parsing XML whose tag names conflict with the names of Beautiful Soup methods and members.

```
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>
```

If you're looking for tag names that aren't valid Python identifiers (like `hyphenated-name`), you need to use `first`.

Searching the Parse Tree

Beautiful Soup provides many methods that traverse the parse tree, gathering `Tag`s and `NavigableString`s that match criteria you specify.

There are several ways to define criteria for matching Beautiful Soup objects. Let's demonstrate by examining in depth the most basic of all Beautiful Soup search methods, `findAll`. As before, we'll demonstrate on the following document:

```
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>
```

Incidentally, the two methods described in this section (`findAll` and `find`) are available only to `Tag` objects and the top-level parser objects, not to `NavigableString` objects. The methods defined in Searching Within the Parse Tree are also available to `NavigableString` objects.

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

The `findAll` method traverses the tree, starting at the given point, and finds all the `Tag` and `NavigableString` objects that match the criteria you give. The signature for the `findall` method is this:

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

These arguments show up over and over again throughout the Beautiful Soup API. The most important arguments are `name` and the keyword arguments.

- The **`name`** argument restricts the set of tags by name. There are several ways to restrict the name, and these too show up over and over again throughout the Beautiful Soup API.

  1. The simplest usage is to just pass in a tag name. This code finds all the <B> `Tag`s in the document:

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

  2. You can also pass in a regular expression. This code finds all the tags whose names *start* with B:

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

  3. You can pass in a list or a dictionary. These two calls find all the <TITLE> and all the <P> tags. They work the same way, but the second call runs faster:

     ```
     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. You can pass in the special value `True`, which matches every tag with a name: that is, it matches every 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']
     ```

     This doesn't look useful, but `True` is very useful when restricting attribute values.

  5. You can pass in a callable object which takes a `Tag` object as its only argument, and returns a boolean. Every `Tag` object that `findAll` encounters will be passed into this object, and if the call returns `True` then the tag is considered to match.

     This code finds the tags that have two, and only two, attributes:

     ```
     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>]
     ```

     This code finds the tags that have one-character names and no attributes:

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

- The keyword arguments impose restrictions on the attributes of a tag. This simple example finds all the tags which have a value of "center" for their "align" attribute:

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

As with the `name` argument, you can pass a keyword argument different kinds of object to impose different restrictions on the corresponding attribute. You can pass a string, as seen above, to restrict an attribute to a single value. You can also pass a regular expression, a list, a hash, the special values `True` or `None`, or a callable that takes the attribute value as its argument (note that the value may be `None`). Some examples:

```
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>]
```

The special values `True` and `None` are of special interest. `True` matches a tag that has *any* value for the given attribute, and `None` matches a tag that has *no* value for the given attribute. Some examples:

```
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']
```

If you need to impose complex or interlocking restrictions on a tag's attributes, pass in a callable object for `name`, as seen above, and deal with the `Tag` object.

You might have noticed a problem here. What if you have a document with a tag that defines an attribute called `name`? You can't use a keyword argument called `name` because the Beautiful Soup search methods already define a `name` argument. You also can't use a Python reserved word like `for` as a keyword argument.

Beautiful Soup provides a special argument called `attrs` which you can use in these situations. `attrs` is a dictionary that acts just like the keyword arguments:

```
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>]
```

You can use `attrs` if you need to put restrictions on attributes whose names are Python reserved words, like `class`, `for`, or `import`; or attributes whose names are non-keyword arguments to the Beautiful Soup search methods: `name`, `recursive`, `limit`, `text`, or `attrs` itself.

```
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>]
```

Searching by CSS class

The `attrs` argument would be a pretty obscure feature were it not for one thing: CSS. It's very useful to search for a tag that has a certain CSS class, but the name of the CSS attribute, `class`, is also a Python reserved word.

You could search by CSS class with `soup.find("tagName", { "class" : "cssClass" })`, but that's a lot of code for such a common operation. Instead, you can pass a string for `attrs` instead of a dictionary. The string will be used to restrict the CSS class.

```
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`** is an argument that lets you search for `NavigableString` objects instead of `Tag`s. Its value can be a string, a regular expression, a list or dictionary, `True` or `None`, or a callable that takes a `NavigableString` object as its argument:

```
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'.']
```

If you use `text`, then any values you give for `name` and the keyword arguments are ignored.

- **`recursive`** is a boolean argument (defaulting to `True`) which tells Beautiful Soup whether to go all the way down the parse tree, or whether to only look at the immediate children of the `Tag` or the parser object. Here's the difference:

  ```
  [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']
  ```

  When `recursive` is false, only the immediate children of the <HTML> tag are searched. If you know that's all you need to search, you can save some time this way.

- Setting **`limit`** argument lets you stop the search once Beautiful Soup finds a certain number of matches. If there are a thousand tables in your document, but you only need the fourth one, pass in 4 to `limit` and you'll save time. By default, there is no limit.

  ```
  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>]
  ```

Calling a tag is like calling `findall`

A little shortcut for you. If you call the parser object or a `Tag` like a function, then you can pass in all of `findall`'s arguments and it's the same as calling `findall`. In terms of the document above:

```
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)`

Okay, now let's look at the other search methods. They all take pretty much the same arguments as `findAll`.

The `find` method is almost exactly like `findAll`, except that instead of finding all the matching objects, it only finds the first one. It's like imposing a `limit` of 1 on the result set, and then extracting the single result from the array. In terms of the document above:

```
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
```

In general, when you see a search method with a plural name (like `findAll` or `findNextSiblings`), that method takes a `limit` argument and returns a list of results. When you see a search method that doesn't have a plural name (like `find` or `findNextSibling`), you know that the method doesn't take a `limit` and returns a single result.

What happened to `first`?

Previous versions of Beautiful Soup had methods like `first`, `fetch`, and `fetchPrevious`. These methods are sitll there, but they're deprecated, and may go away soon. The total effect of all those names was very confusing. The new names are named consistently: as mentioned above, if the method name is plural or refers to `All`, it returns multiple objects. Otherwise, it returns one object.

Searching Within the Parse Tree

The methods described above, `findAll` and `find`, start at a certain point in the parse tree and go down. They recursively iterate through an object's `contents` until they bottom out.

This means that you can't call these methods on `NavigableString` objects, because they have no `contents`: they're always the leaves of the parse tree.

But downwards isn't the only way you can iterate through a document. Back in Navigating the Parse Tree I showed you many other ways: `parent`, `nextSibling`, and so on. Each of these iteration techniques has two corresponding methods: one that works like `findAll`, and one that works like `find`. And since `NavigableString` objects *do* support these operations, you can call these methods on them as well as on `Tag` objects and the main parser object.

Why is this useful? Well, sometimes you just can't use `findAll` or `find` to get to the `Tag` or `NavigableString` you want. For instance, consider some HTML like this:

```
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>''')
```

There are a number of ways to navigate to the <LI> tag that contains the data you want. The most obvious is this:

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

It should be equally obvious that that's not a very stable way to get that <LI> tag. If you're only scraping this page once it doesn't matter, but if you're going to scrape it many times over a long period, such considerations become important. 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.

```
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.

When you look at that HTML, you might think of the list you want as 'the <UL> tag beneath the <H1> tag'. The problem is that the tag isn't contained inside the <H1> tag; it just happens to comes after it. It's easy enough to get the <H1> tag, but there's no way to get from there to the <UL> tag using `first` and `fetch`, because those methods only search the `contents` of the <H1> tag. You need to navigate to the <UL> tag with the `next` or `nextSibling` members:

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

Or, if you think this might be more stable:

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

But that's more trouble than you should need to go through. The methods in this section provide a useful shorthand. They can be used whenever you find yourself wanting to write a while loop over one of the navigation members. Given a starting point somewhere in the tree, they navigate the tree in some way and keep track of `Tag` or `NavigableString` objects that match the criteria you specify. Instead of the first loop in the example code above, you can just write this:

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

Instead of the second loop, you can write this:

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

The loops are replaced with calls to `findNextSibling` and `findNext`. The rest of this section is a reference to all the methods of this kind. Again, there are two methods for every navigation member: one that returns a list the way `findAll` does, and one that returns a scalar the way `find` does.

One last time, let's load up the familiar soup document for example's sake:

```
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)

These methods repeatedly follow an object's `nextSibling` member, gathering `Tag` or `NavigableText` objects that match the criteria you specify. In terms of the document above:

```
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)

These methods repeatedly follow an object's `previousSibling` member, gathering `Tag` or `NavigableText` objects that match the criteria you specify. In terms of the document above:

```
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)

These methods repeatedly follow an object's `next` member, gathering `Tag` or `NavigableText` objects that match the criteria you specify. In terms of the document above:

```
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)`

These methods repeatedly follow an object's `previous` member, gathering `Tag` or `NavigableText` objects that match the criteria you specify. In terms of the document above:

```
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)`

These methods repeatedly follow an object's `parent` member, gathering `Tag` or `NavigableText` objects that match the criteria you specify. They don't take a `text` argument, because there's no way any object can have a `NavigableString` for a parent. In terms of the document above:

```
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'
```

Modifying the Parse Tree

Now you know how to find things in the parse tree. But maybe you want to modify it and print it back out. You can just rip an element out of its parent's `contents`, but the rest of the document will still have references to the thing you ripped out. Beautiful Soup offers several methods that let you modify the parse tree while maintaining its internal consistency.

Changing attribute values

You can use dictionary assignment to modify the attribute values of `Tag` objects.

```
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>
```

You can also delete attribute values, and add new ones:

```
del(b['id'])
print soup
# <b>Argh!</b>

b['class'] = "extra bold and brassy!"
print soup
# <b class="extra bold and brassy!">Argh!</b>
```

Removing elements

Once you have a reference to an element, you can rip it out of the tree with the `extract` method. This code removes all the comments from a document:

```
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>
```

This code removes a whole subtree from a document:

```
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>
```

The `extract` method turns one parse tree into two disjoint trees. The navigation members are changed so that it looks like the trees had never been together:

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

Replacing one Element with Another

The `replaceWith` method extracts one page element and replaces it with a different one. 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!']
```

Here's a more complex example that replaces one tag with another:

```
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:

```
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>
```

Adding a Brand New Element

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.

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:

```
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:

```
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.

Troubleshooting

This section covers common problems people have with Beautiful Soup.

Why can't Beautiful Soup print out the non-ASCII characters I gave it?

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:

```
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.

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.

```
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.

2. The hard way is to create a `sitecustomize.py` file in your Python installation which sets 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:

```
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.

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`.

Beautiful Soup loses the data I fed it! Why? WHY?????

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.

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:

- Beautiful Soup treats ill-formed XML definitions as data. However, it loses well-formed XML definitions that don't actually exist:

```
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:

```
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.

Beautiful Soup also ignores an entity reference that's not finished by the end of the document:

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

I've never seen this in real web pages, but it's probably out there somewhere.

- 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.

The parse tree built by the `BeautifulSoup` class offends my senses!

To get your markup parsed differently, check out Other Built-In Parsers, or else build a custom parser.

Beautiful Soup is too slow!

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.

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` or `decompose`.

Advanced Topics

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.

Generators

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.

Here's a simple example that strips HTML tags out of a document by iterating over the document and collecting all the strings.

```
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!'
```

Of course, you don't really need a generator to find only the text beneath a tag. That code does the same thing as `.findAll(text=True)`.

```
''.join(soup.findAll(text=True))
# 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.

```
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
```

Other Built-In Parsers

Beautiful Soup comes with three parser classes besides `BeautifulSoup` and `BeautifulStoneSoup`:

- `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.

- `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`.

- `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:

  ```
  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.

Customizing the 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.

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.

When Beautiful Soup is parsing a document, it keeps a stack of open tags. Whenever it sees a new 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.

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.

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.

```
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.

```
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.

```
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:

```
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.

```
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:

```
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.

One more example. Say the stack looks like `['html', 'p', 'table']` and Beautiful Soup encounters a <P> tag.

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.

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:

```
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.

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.

```
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 actually a list, put into the form of a dictionary for quick random access.

```
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.

Entity Conversion

When you parse a document, you can convert HTML or XML entity references to the corresponding Unicode characters. This code converts the HTML entity "&eacute;" to the Unicode character LATIN SMALL LETTER E WITH ACUTE, and the numeric entity "&#101;" to the Unicode character 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 ("&quot;", "&apos;", "&gt;", "&lt;", and "&amp;") 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 &apos; is an XML entity but not an HTML entity.

```
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.

```
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.

Sanitizing Bad Data with Regexps

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.

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 <BR/>, and changes them to look like <BR />. It finds declarations that have extraneous whitespace, like <! --Comment-->, and removes the whitespace: <!--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.

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:

```
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:

```
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 <BR> 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.

```
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.

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`.

Fun With `SoupStrainer`s

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.

```
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:

```
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:

Improving Performance by Parsing Only Part of the Document

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.

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.

Here's a pretty varied document:

```
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> sure seem obsessed
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.

```
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.

```
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>]
```

Improving Memory Usage with `extract`

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.

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.

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.

If `extract` doesn't work for you, you can try `Tag.decompose`. It's slower than `extract` but more thorough. It recursively disassembles a `Tag` and its contents, disconnecting every part of a tree from every other part.

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.

See Also

Applications that use Beautiful Soup

Lots of real-world applications use Beautiful Soup. Here are the publicly visible applications that I know about:

- 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.

Similar libraries

I've found several other parsers for various languages that can handle bad markup, do tree traversal for you, or are otherwise more useful than your average parser.

- 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.

Conclusion

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.

--Leonard

Document tree:

http://www.crummy.com/software/

Time.

BeautifulSoup/documentation.html

Site Search: