Beautiful Soup Documentation

Release 4.4.0

Leonard Richardson

Sep 04, 2017

Contents

Beautiful Soup is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work.

These instructions illustrate all major features of Beautiful Soup 4, with examples. I show 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.

The examples in this documentation should work the same way in Python 2.7 and Python 3.2.

You might be looking for the documentation for Beautiful Soup 3. If so, you should know that Beautiful Soup 3 is no longer being developed, and that Beautiful Soup 4 is recommended for all new projects. If you want to learn about the differences between Beautiful Soup 3 and Beautiful Soup 4, see *Porting code to BS4*.

This documentation has been translated into other languages by Beautiful Soup users:

- .
- ()
- . ( )

CHAPTER 1

Getting help

If you have questions about Beautiful Soup, or run into problems, send mail to the discussion group. If your problem involves parsing an HTML document, be sure to mention *what the diagnose() function says* about that document.

CHAPTER 2

Quick Start

Here's an HTML document I'll be using as an example throughout this document. It's part of a story from *Alice in Wonderland*:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names␣
↪were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""
```

Running the "three sisters" document through Beautiful Soup gives us a `BeautifulSoup` object, which represents the document as a nested data structure:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')

print(soup.prettify())
# <html>
#  <head>
#   <title>
#    The Dormouse's story
#   </title>
#  </head>
#  <body>
#   <p class="title">
#    <b>
#     The Dormouse's story
```

```
#     </b>
#    </p>
#    <p class="story">
#     Once upon a time there were three little sisters; and their names were
#     <a class="sister" href="http://example.com/elsie" id="link1">
#      Elsie
#     </a>
#     ,
#     <a class="sister" href="http://example.com/lacie" id="link2">
#      Lacie
#     </a>
#     and
#     <a class="sister" href="http://example.com/tillie" id="link2">
#      Tillie
#     </a>
#     ; and they lived at the bottom of a well.
#    </p>
#    <p class="story">
#     ...
#    </p>
#   </body>
# </html>
```

Here are some simple ways to navigate that data structure:

```
soup.title
# <title>The Dormouse's story</title>

soup.title.name
# u'title'

soup.title.string
# u'The Dormouse's story'

soup.title.parent.name
# u'head'

soup.p
# <p class="title"><b>The Dormouse's story</b></p>

soup.p['class']
# u'title'

soup.a
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

soup.find_all('a')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.find(id="link3")
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
```

One common task is extracting all the URLs found within a page's <a> tags:

```
for link in soup.find_all('a'):
    print(link.get('href'))
```

```
# http://example.com/elsie
# http://example.com/lacie
# http://example.com/tillie
```

Another common task is extracting all the text from a page:

```
print(soup.get_text())
# The Dormouse's story
#
# The Dormouse's story
#
# Once upon a time there were three little sisters; and their names were
# Elsie,
# Lacie and
# Tillie;
# and they lived at the bottom of a well.
#
# ...
```

Does this look like what you need? If so, read on.

CHAPTER 3

Installing Beautiful Soup

If you're using a recent version of Debian or Ubuntu Linux, you can install Beautiful Soup with the system package manager:

`$ apt-get install python-bs4` (for Python 2)

`$ apt-get install python3-bs4` (for Python 3)

Beautiful Soup 4 is published through PyPi, so if you can't install it with the system packager, you can install it with `easy_install` or `pip`. The package name is `beautifulsoup4`, and the same package works on Python 2 and Python 3. Make sure you use the right version of `pip` or `easy_install` for your Python version (these may be named `pip3` and `easy_install3` respectively if you're using Python 3).

```
$ easy_install beautifulsoup4
```

```
$ pip install beautifulsoup4
```

(The `BeautifulSoup` package is probably *not* what you want. That's the previous major release, Beautiful Soup 3. Lots of software uses BS3, so it's still available, but if you're writing new code you should install `beautifulsoup4`.)

If you don't have `easy_install` or `pip` installed, you can download the Beautiful Soup 4 source tarball and install it with `setup.py`.

```
$ python setup.py install
```

If all else fails, the license for Beautiful Soup allows you to package the entire library with your application. You can download the tarball, copy its `bs4` directory into your application's codebase, and use Beautiful Soup without installing it at all.

I use Python 2.7 and Python 3.2 to develop Beautiful Soup, but it should work with other recent versions.

Problems after installation

Beautiful Soup is packaged as Python 2 code. When you install it for use with Python 3, it's automatically converted to Python 3 code. If you don't install the package, the code won't be converted. There have also been reports on Windows machines of the wrong version being installed.

If you get the `ImportError` "No module named HTMLParser", your problem is that you're running the Python 2 version of the code under Python 3.

If you get the `ImportError` "No module named html.parser", your problem is that you're running the Python 3 version of the code under Python 2.

In both cases, your best bet is to completely remove the Beautiful Soup installation from your system (including any directory created when you unzipped the tarball) and try the installation again.

If you get the `SyntaxError` "Invalid syntax" on the line `ROOT_TAG_NAME = u'[document]'`, you need to convert the Python 2 code to Python 3. You can do this either by installing the package:

```
$ python3 setup.py install
```

or by manually running Python's `2to3` conversion script on the `bs4` directory:

```
$ 2to3-3.2 -w bs4
```

Installing a parser

Beautiful Soup supports the HTML parser included in Python's standard library, but it also supports a number of third-party Python parsers. One is the lxml parser. Depending on your setup, you might install lxml with one of these commands:

```
$ apt-get install python-lxml
```

```
$ easy_install lxml
```

```
$ pip install lxml
```

Another alternative is the pure-Python html5lib parser, which parses HTML the way a web browser does. Depending on your setup, you might install html5lib with one of these commands:

```
$ apt-get install python-html5lib
```

```
$ easy_install html5lib
```

```
$ pip install html5lib
```

This table summarizes the advantages and disadvantages of each parser library:

Parser	Typical usage	Advantages	Disadvantages
Python's html.parser	`BeautifulSoup(markup, "html.parser")`	• Batteries included • Decent speed • Lenient (as of Python 2.7.3 and 3.2.)	• Not very lenient (before Python 2.7.3 or 3.2.2)
lxml's HTML parser	`BeautifulSoup(markup, "lxml")`	• Very fast • Lenient	• External C dependency
lxml's XML parser	`BeautifulSoup(markup, "lxml-xml")` `BeautifulSoup(markup, "xml")`	• Very fast • The only currently supported XML parser	• External C dependency
html5lib	`BeautifulSoup(markup, "html5lib")`	• Extremely lenient • Parses pages the same way a web browser does • Creates valid HTML5	• Very slow • External Python dependency

If you can, I recommend you install and use lxml for speed. If you're using a version of Python 2 earlier than 2.7.3, or a version of Python 3 earlier than 3.2.2, it's *essential* that you install lxml or html5lib–Python's built-in HTML parser is just not very good in older versions.

Note that if a document is invalid, different parsers will generate different Beautiful Soup trees for it. See *Differences between parsers* for details.

CHAPTER 4

Making the soup

To parse a document, pass it into the `BeautifulSoup` constructor. You can pass in a string or an open filehandle:

```
from bs4 import BeautifulSoup

with open("index.html") as fp:
    soup = BeautifulSoup(fp)

soup = BeautifulSoup("<html>data</html>")
```

First, the document is converted to Unicode, and HTML entities are converted to Unicode characters:

```
BeautifulSoup("Sacr&eacute; bleu!")
<html><head></head><body>Sacré bleu!</body></html>
```

Beautiful Soup then parses the document using the best available parser. It will use an HTML parser unless you specifically tell it to use an XML parser. (See *Parsing XML*.)

CHAPTER 5

Kinds of objects

Beautiful Soup transforms a complex HTML document into a complex tree of Python objects. But you'll only ever have to deal with about four *kinds* of objects: `Tag`, `NavigableString`, `BeautifulSoup`, and `Comment`.

Tag

A `Tag` object corresponds to an XML or HTML tag in the original document:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')
tag = soup.b
type(tag)
# <class 'bs4.element.Tag'>
```

Tags have a lot of attributes and methods, and I'll cover most of them in *Navigating the tree* and *Searching the tree*. For now, the most important features of a tag are its name and attributes.

Name

Every tag has a name, accessible as `.name`:

```
tag.name
# u'b'
```

If you change a tag's name, the change will be reflected in any HTML markup generated by Beautiful Soup:

```
tag.name = "blockquote"
tag
# <blockquote class="boldest">Extremely bold</blockquote>
```

Attributes

A tag may have any number of attributes. The tag `<b id="boldest">` has an attribute "id" whose value is "boldest". You can access a tag's attributes by treating the tag like a dictionary:

```
tag['id']
# u'boldest'
```

You can access that dictionary directly as `.attrs`:

```
tag.attrs
# {u'id': 'boldest'}
```

You can add, remove, and modify a tag's attributes. Again, this is done by treating the tag as a dictionary:

```
tag['id'] = 'verybold'
tag['another-attribute'] = 1
tag
# <b another-attribute="1" id="verybold"></b>

del tag['id']
del tag['another-attribute']
tag
# <b></b>

tag['id']
# KeyError: 'id'
print(tag.get('id'))
# None
```

Multi-valued attributes

HTML 4 defines a few attributes that can have multiple values. HTML 5 removes a couple of them, but defines a few more. The most common multi-valued attribute is `class` (that is, a tag can have more than one CSS class). Others include `rel`, `rev`, `accept-charset`, `headers`, and `accesskey`. Beautiful Soup presents the value(s) of a multi-valued attribute as a list:

```
css_soup = BeautifulSoup('<p class="body"></p>')
css_soup.p['class']
# ["body"]

css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.p['class']
# ["body", "strikeout"]
```

If an attribute *looks* like it has more than one value, but it's not a multi-valued attribute as defined by any version of the HTML standard, Beautiful Soup will leave the attribute alone:

```
id_soup = BeautifulSoup('<p id="my id"></p>')
id_soup.p['id']
# 'my id'
```

When you turn a tag back into a string, multiple attribute values are consolidated:

```
rel_soup = BeautifulSoup('<p>Back to the <a rel="index">homepage</a></p>')
rel_soup.a['rel']
```

```
# ['index']
rel_soup.a['rel'] = ['index', 'contents']
print(rel_soup.p)
# <p>Back to the <a rel="index contents">homepage</a></p>
```

You can use `get_attribute_list to get a value that's always a list, string, whether or not it's a multi-valued atribute

id_soup.p.get_attribute_list('id') # ["my id"]

If you parse a document as XML, there are no multi-valued attributes:

```
xml_soup = BeautifulSoup('<p class="body strikeout"></p>', 'xml')
xml_soup.p['class']
# u'body strikeout'
```

NavigableString

A string corresponds to a bit of text within a tag. Beautiful Soup uses the `NavigableString` class to contain these bits of text:

```
tag.string
# u'Extremely bold'
type(tag.string)
# <class 'bs4.element.NavigableString'>
```

A `NavigableString` is just like a Python Unicode string, except that it also supports some of the features described in *Navigating the tree* and *Searching the tree*. You can convert a `NavigableString` to a Unicode string with `unicode()`:

```
unicode_string = unicode(tag.string)
unicode_string
# u'Extremely bold'
type(unicode_string)
# <type 'unicode'>
```

You can't edit a string in place, but you can replace one string with another, using *replace_with()*:

```
tag.string.replace_with("No longer bold")
tag
# <blockquote>No longer bold</blockquote>
```

`NavigableString` supports most of the features described in *Navigating the tree* and *Searching the tree*, but not all of them. In particular, since a string can't contain anything (the way a tag may contain a string or another tag), strings don't support the `.contents` or `.string` attributes, or the `find()` method.

If you want to use a `NavigableString` outside of Beautiful Soup, you should call `unicode()` on it to turn it into a normal Python Unicode string. If you don't, your string will carry around a reference to the entire Beautiful Soup parse tree, even when you're done using Beautiful Soup. This is a big waste of memory.

BeautifulSoup

The `BeautifulSoup` object itself represents the document as a whole. For most purposes, you can treat it as a *Tag* object. This means it supports most of the methods described in *Navigating the tree* and *Searching the tree*.

Since the `BeautifulSoup` object doesn't correspond to an actual HTML or XML tag, it has no name and no attributes. But sometimes it's useful to look at its `.name`, so it's been given the special `.name` "[document]":

```
soup.name
# u'[document]'
```

Comments and other special strings

`Tag`, `NavigableString`, and `BeautifulSoup` cover almost everything you'll see in an HTML or XML file, but there are a few leftover bits. The only one you'll probably ever need to worry about is the comment:

```
markup = "<b><!--Hey, buddy. Want to buy a used parser?--></b>"
soup = BeautifulSoup(markup)
comment = soup.b.string
type(comment)
# <class 'bs4.element.Comment'>
```

The `Comment` object is just a special type of `NavigableString`:

```
comment
# u'Hey, buddy. Want to buy a used parser'
```

But when it appears as part of an HTML document, a `Comment` is displayed with special formatting:

```
print(soup.b.prettify())
# <b>
#  <!--Hey, buddy. Want to buy a used parser?-->
# </b>
```

Beautiful Soup defines classes for anything else that might show up in an XML document: `CData`, `ProcessingInstruction`, `Declaration`, and `Doctype`. Just like `Comment`, these classes are subclasses of `NavigableString` that add something extra to the string. Here's an example that replaces the comment with a CDATA block:

```
from bs4 import CData
cdata = CData("A CDATA block")
comment.replace_with(cdata)

print(soup.b.prettify())
# <b>
#  <![CDATA[A CDATA block]]>
# </b>
```

CHAPTER 6

Navigating the tree

Here's the "Three sisters" HTML document again:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names␣
↪were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
```

I'll use this as an example to show you how to move from one part of a document to another.

Going down

Tags may contain strings and other tags. These elements are the tag's *children*. Beautiful Soup provides a lot of different attributes for navigating and iterating over a tag's children.

Note that Beautiful Soup strings don't support any of these attributes, because a string can't have children.

Navigating using tag names

The simplest way to navigate the parse tree is to say the name of the tag you want. If you want the <head> tag, just say `soup.head`:

```
soup.head
# <head><title>The Dormouse's story</title></head>

soup.title
# <title>The Dormouse's story</title>
```

You can do use this trick again and again to zoom in on a certain part of the parse tree. This code gets the first <b> tag beneath the <body> tag:

```
soup.body.b
# <b>The Dormouse's story</b>
```

Using a tag name as an attribute will give you only the *first* tag by that name:

```
soup.a
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

If you need to get *all* the <a> tags, or anything more complicated than the first tag with a certain name, you'll need to use one of the methods described in *Searching the tree*, such as *find_all()*:

```
soup.find_all('a')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

`.contents` and `.children`

A tag's children are available in a list called `.contents`:

```
head_tag = soup.head
head_tag
# <head><title>The Dormouse's story</title></head>

head_tag.contents
[<title>The Dormouse's story</title>]

title_tag = head_tag.contents[0]
title_tag
# <title>The Dormouse's story</title>
title_tag.contents
# [u'The Dormouse's story']
```

The `BeautifulSoup` object itself has children. In this case, the <html> tag is the child of the `BeautifulSoup` object.:

```
len(soup.contents)
# 1
soup.contents[0].name
# u'html'
```

A string does not have `.contents`, because it can't contain anything:

```
text = title_tag.contents[0]
text.contents
# AttributeError: 'NavigableString' object has no attribute 'contents'
```

Instead of getting them as a list, you can iterate over a tag's children using the `.children` generator:

```
for child in title_tag.children:
    print(child)
# The Dormouse's story
```

.descendants

The `.contents` and `.children` attributes only consider a tag's *direct* children. For instance, the <head> tag has a single direct child–the <title> tag:

```
head_tag.contents
# [<title>The Dormouse's story</title>]
```

But the <title> tag itself has a child: the string "The Dormouse's story". There's a sense in which that string is also a child of the <head> tag. The `.descendants` attribute lets you iterate over *all* of a tag's children, recursively: its direct children, the children of its direct children, and so on:

```
for child in head_tag.descendants:
    print(child)
# <title>The Dormouse's story</title>
# The Dormouse's story
```

The <head> tag has only one child, but it has two descendants: the <title> tag and the <title> tag's child. The `BeautifulSoup` object only has one direct child (the <html> tag), but it has a whole lot of descendants:

```
len(list(soup.children))
# 1
len(list(soup.descendants))
# 25
```

.string

If a tag has only one child, and that child is a `NavigableString`, the child is made available as `.string`:

```
title_tag.string
# u'The Dormouse's story'
```

If a tag's only child is another tag, and *that* tag has a `.string`, then the parent tag is considered to have the same `.string` as its child:

```
head_tag.contents
# [<title>The Dormouse's story</title>]

head_tag.string
# u'The Dormouse's story'
```

If a tag contains more than one thing, then it's not clear what `.string` should refer to, so `.string` is defined to be `None`:

```
print(soup.html.string)
# None
```

.strings and stripped_strings

If there's more than one thing inside a tag, you can still look at just the strings. Use the `.strings` generator:

```
for string in soup.strings:
    print(repr(string))
# u"The Dormouse's story"
# u'\n\n'
# u"The Dormouse's story"
# u'\n\n'
# u'Once upon a time there were three little sisters; and their names were\n'
# u'Elsie'
# u',\n'
# u'Lacie'
# u' and\n'
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'\n\n'
# u'...'
# u'\n'
```

These strings tend to have a lot of extra whitespace, which you can remove by using the `.stripped_strings` generator instead:

```
for string in soup.stripped_strings:
    print(repr(string))
# u"The Dormouse's story"
# u"The Dormouse's story"
# u'Once upon a time there were three little sisters; and their names were'
# u'Elsie'
# u','
# u'Lacie'
# u'and'
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'...'
```

Here, strings consisting entirely of whitespace are ignored, and whitespace at the beginning and end of strings is removed.

Going up

Continuing the "family tree" analogy, every tag and every string has a *parent*: the tag that contains it.

.parent

You can access an element's parent with the `.parent` attribute. In the example "three sisters" document, the <head> tag is the parent of the <title> tag:

```
title_tag = soup.title
title_tag
# <title>The Dormouse's story</title>
title_tag.parent
# <head><title>The Dormouse's story</title></head>
```

The title string itself has a parent: the <title> tag that contains it:

```
title_tag.string.parent
# <title>The Dormouse's story</title>
```

The parent of a top-level tag like <html> is the `BeautifulSoup` object itself:

```
html_tag = soup.html
type(html_tag.parent)
# <class 'bs4.BeautifulSoup'>
```

And the `.parent` of a `BeautifulSoup` object is defined as None:

```
print(soup.parent)
# None
```

.parents

You can iterate over all of an element's parents with `.parents`. This example uses `.parents` to travel from an <a> tag buried deep within the document, to the very top of the document:

```
link = soup.a
link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
for parent in link.parents:
    if parent is None:
        print(parent)
    else:
        print(parent.name)
# p
# body
# html
# [document]
# None
```

Going sideways

Consider a simple document like this:

```
sibling_soup = BeautifulSoup("<a><b>text1</b><c>text2</c></b></a>")
print(sibling_soup.prettify())
# <html>
#  <body>
#   <a>
#    <b>
#     text1
#    </b>
```

```
#     <c>
#      text2
#     </c>
#    </a>
#   </body>
# </html>
```

The <b> tag and the <c> tag are at the same level: they're both direct children of the same tag. We call them *siblings*. When a document is pretty-printed, siblings show up at the same indentation level. You can also use this relationship in the code you write.

`.next_sibling` and `.previous_sibling`

You can use `.next_sibling` and `.previous_sibling` to navigate between page elements that are on the same level of the parse tree:

```
sibling_soup.b.next_sibling
# <c>text2</c>

sibling_soup.c.previous_sibling
# <b>text1</b>
```

The <b> tag has a `.next_sibling`, but no `.previous_sibling`, because there's nothing before the <b> tag *on the same level of the tree*. For the same reason, the <c> tag has a `.previous_sibling` but no `.next_sibling`:

```
print(sibling_soup.b.previous_sibling)
# None
print(sibling_soup.c.next_sibling)
# None
```

The strings "text1" and "text2" are *not* siblings, because they don't have the same parent:

```
sibling_soup.b.string
# u'text1'

print(sibling_soup.b.string.next_sibling)
# None
```

In real documents, the `.next_sibling` or `.previous_sibling` of a tag will usually be a string containing whitespace. Going back to the "three sisters" document:

```
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a>
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>
```

You might think that the `.next_sibling` of the first <a> tag would be the second <a> tag. But actually, it's a string: the comma and newline that separate the first <a> tag from the second:

```
link = soup.a
link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

link.next_sibling
# u',\n'
```

The second <a> tag is actually the `.next_sibling` of the comma:

```
link.next_sibling.next_sibling
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
```

.next_siblings and .previous_siblings

You can iterate over a tag's siblings with `.next_siblings` or `.previous_siblings`:

```
for sibling in soup.a.next_siblings:
    print(repr(sibling))
# u',\n'
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
# u' and\n'
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
# u'; and they lived at the bottom of a well.'
# None

for sibling in soup.find(id="link3").previous_siblings:
    print(repr(sibling))
# ' and\n'
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
# u',\n'
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
# u'Once upon a time there were three little sisters; and their names were\n'
# None
```

Going back and forth

Take a look at the beginning of the "three sisters" document:

```
<html><head><title>The Dormouse's story</title></head>
<p class="title"><b>The Dormouse's story</b></p>
```

An HTML parser takes this string of characters and turns it into a series of events: "open an <html> tag", "open a <head> tag", "open a <title> tag", "add a string", "close the <title> tag", "open a <p> tag", and so on. Beautiful Soup offers tools for reconstructing the initial parse of the document.

.next_element and .previous_element

The `.next_element` attribute of a string or tag points to whatever was parsed immediately afterwards. It might be the same as `.next_sibling`, but it's usually drastically different.

Here's the final <a> tag in the "three sisters" document. Its `.next_sibling` is a string: the conclusion of the sentence that was interrupted by the start of the <a> tag.:

```
last_a_tag = soup.find("a", id="link3")
last_a_tag
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>

last_a_tag.next_sibling
# '; and they lived at the bottom of a well.'
```

But the `.next_element` of that <a> tag, the thing that was parsed immediately after the <a> tag, is *not* the rest of that sentence: it's the word "Tillie":

```
last_a_tag.next_element
# u'Tillie'
```

That's because in the original markup, the word "Tillie" appeared before that semicolon. The parser encountered an <a> tag, then the word "Tillie", then the closing tag, then the semicolon and rest of the sentence. The semicolon is on the same level as the <a> tag, but the word "Tillie" was encountered first.

The `.previous_element` attribute is the exact opposite of `.next_element`. It points to whatever element was parsed immediately before this one:

```
last_a_tag.previous_element
# u' and\n'
last_a_tag.previous_element.next_element
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
```

`.next_elements` and `.previous_elements`

You should get the idea by now. You can use these iterators to move forward or backward in the document as it was parsed:

```
for element in last_a_tag.next_elements:
    print(repr(element))
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'\n\n'
# <p class="story">...</p>
# u'...'
# u'\n'
# None
```

CHAPTER 7

Searching the tree

Beautiful Soup defines a lot of methods for searching the parse tree, but they're all very similar. I'm going to spend a lot of time explaining the two most popular methods: `find()` and `find_all()`. The other methods take almost exactly the same arguments, so I'll just cover them briefly.

Once again, I'll be using the "three sisters" document as an example:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names␣
↪were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
```

By passing in a filter to an argument like `find_all()`, you can zoom in on the parts of the document you're interested in.

Kinds of filters

Before talking in detail about `find_all()` and similar methods, I want to show examples of different filters you can pass into these methods. These filters show up again and again, throughout the search API. You can use them to filter based on a tag's name, on its attributes, on the text of a string, or on some combination of these.

A string

The simplest filter is a string. Pass a string to a search method and Beautiful Soup will perform a match against that exact string. This code finds all the <b> tags in the document:

```
soup.find_all('b')
# [<b>The Dormouse's story</b>]
```

If you pass in a byte string, Beautiful Soup will assume the string is encoded as UTF-8. You can avoid this by passing in a Unicode string instead.

A regular expression

If you pass in a regular expression object, Beautiful Soup will filter against that regular expression using its `search()` method. This code finds all the tags whose names start with the letter "b"; in this case, the <body> tag and the <b> tag:

```
import re
for tag in soup.find_all(re.compile("^b")):
    print(tag.name)
# body
# b
```

This code finds all the tags whose names contain the letter 't':

```
for tag in soup.find_all(re.compile("t")):
    print(tag.name)
# html
# title
```

A list

If you pass in a list, Beautiful Soup will allow a string match against *any* item in that list. This code finds all the <a> tags *and* all the <b> tags:

```
soup.find_all(["a", "b"])
# [<b>The Dormouse's story</b>,
#  <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

`True`

The value `True` matches everything it can. This code finds *all* the tags in the document, but none of the text strings:

```
for tag in soup.find_all(True):
    print(tag.name)
# html
# head
# title
# body
# p
# b
```

```
# p
# a
# a
# a
# p
```

A function

If none of the other matches work for you, define a function that takes an element as its only argument. The function should return `True` if the argument matches, and `False` otherwise.

Here's a function that returns `True` if a tag defines the "class" attribute but doesn't define the "id" attribute:

```
def has_class_but_no_id(tag):
    return tag.has_attr('class') and not tag.has_attr('id')
```

Pass this function into `find_all()` and you'll pick up all the <p> tags:

```
soup.find_all(has_class_but_no_id)
# [<p class="title"><b>The Dormouse's story</b></p>,
#  <p class="story">Once upon a time there were...</p>,
#  <p class="story">...</p>]
```

This function only picks up the <p> tags. It doesn't pick up the <a> tags, because those tags define both "class" and "id". It doesn't pick up tags like <html> and <title>, because those tags don't define "class".

If you pass in a function to filter on a specific attribute like `href`, the argument passed into the function will be the attribute value, not the whole tag. Here's a function that finds all `a` tags whose `href` attribute *does not* match a regular expression:

```
def not_lacie(href):
    return href and not re.compile("lacie").search(href)
soup.find_all(href=not_lacie)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

The function can be as complicated as you need it to be. Here's a function that returns `True` if a tag is surrounded by string objects:

```
from bs4 import NavigableString
def surrounded_by_strings(tag):
    return (isinstance(tag.next_element, NavigableString)
            and isinstance(tag.previous_element, NavigableString))

for tag in soup.find_all(surrounded_by_strings):
    print tag.name
# p
# a
# a
# a
# p
```

Now we're ready to look at the search methods in detail.

find_all()

Signature: find_all(*name*, *attrs*, *recursive*, *string*, *limit*, ***kwargs*)

The `find_all()` method looks through a tag's descendants and retrieves *all* descendants that match your filters. I gave several examples in *Kinds of filters*, but here are a few more:

```
soup.find_all("title")
# [<title>The Dormouse's story</title>]

soup.find_all("p", "title")
# [<p class="title"><b>The Dormouse's story</b></p>]

soup.find_all("a")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.find_all(id="link2")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

import re
soup.find(string=re.compile("sisters"))
# u'Once upon a time there were three little sisters; and their names were\n'
```

Some of these should look familiar, but others are new. What does it mean to pass in a value for `string`, or `id`? Why does `find_all("p", "title")` find a <p> tag with the CSS class "title"? Let's look at the arguments to `find_all()`.

The name argument

Pass in a value for `name` and you'll tell Beautiful Soup to only consider tags with certain names. Text strings will be ignored, as will tags whose names that don't match.

This is the simplest usage:

```
soup.find_all("title")
# [<title>The Dormouse's story</title>]
```

Recall from *Kinds of filters* that the value to `name` can be *a string*, *a regular expression*, *a list*, *a function*, or *the value True*.

The keyword arguments

Any argument that's not recognized will be turned into a filter on one of a tag's attributes. If you pass in a value for an argument called `id`, Beautiful Soup will filter against each tag's 'id' attribute:

```
soup.find_all(id='link2')
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

If you pass in a value for `href`, Beautiful Soup will filter against each tag's 'href' attribute:

```
soup.find_all(href=re.compile("elsie"))
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

You can filter an attribute based on *a string*, *a regular expression*, *a list*, *a function*, or *the value True*.

This code finds all tags whose `id` attribute has a value, regardless of what the value is:

```
soup.find_all(id=True)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

You can filter multiple attributes at once by passing in more than one keyword argument:

```
soup.find_all(href=re.compile("elsie"), id='link1')
# [<a class="sister" href="http://example.com/elsie" id="link1">three</a>]
```

Some attributes, like the data-* attributes in HTML 5, have names that can't be used as the names of keyword arguments:

```
data_soup = BeautifulSoup('<div data-foo="value">foo!</div>')
data_soup.find_all(data-foo="value")
# SyntaxError: keyword can't be an expression
```

You can use these attributes in searches by putting them into a dictionary and passing the dictionary into `find_all()` as the `attrs` argument:

```
data_soup.find_all(attrs={"data-foo": "value"})
# [<div data-foo="value">foo!</div>]
```

You can't use a keyword argument to search for HTML's 'name' element, because Beautiful Soup uses the `name` argument to contain the name of the tag itself. Instead, you can give a value to 'name' in the `attrs` argument.

name_soup = BeautifulSoup('<input name="email"/>') name_soup.find_all(name="email") # [] name_soup.find_all(attrs={"name": "email"}) # [<input name="email"/>]

Searching by CSS class

It's very useful to search for a tag that has a certain CSS class, but the name of the CSS attribute, "class", is a reserved word in Python. Using `class` as a keyword argument will give you a syntax error. As of Beautiful Soup 4.1.2, you can search by CSS class using the keyword argument `class_`:

```
soup.find_all("a", class_="sister")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

As with any keyword argument, you can pass `class_` a string, a regular expression, a function, or `True`:

```
soup.find_all(class_=re.compile("itl"))
# [<p class="title"><b>The Dormouse's story</b></p>]

def has_six_characters(css_class):
    return css_class is not None and len(css_class) == 6

soup.find_all(class_=has_six_characters)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

Remember that a single tag can have multiple values for its "class" attribute. When you search for a tag that matches a certain CSS class, you're matching against *any* of its CSS classes:

```
css_soup = BeautifulSoup('<p class="body strikeout"></p>')
css_soup.find_all("p", class_="strikeout")
# [<p class="body strikeout"></p>]

css_soup.find_all("p", class_="body")
# [<p class="body strikeout"></p>]
```

You can also search for the exact string value of the `class` attribute:

```
css_soup.find_all("p", class_="body strikeout")
# [<p class="body strikeout"></p>]
```

But searching for variants of the string value won't work:

```
css_soup.find_all("p", class_="strikeout body")
# []
```

If you want to search for tags that match two or more CSS classes, you should use a CSS selector:

```
css_soup.select("p.strikeout.body")
# [<p class="body strikeout"></p>]
```

In older versions of Beautiful Soup, which don't have the `class_` shortcut, you can use the `attrs` trick mentioned above. Create a dictionary whose value for "class" is the string (or regular expression, or whatever) you want to search for:

```
soup.find_all("a", attrs={"class": "sister"})
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

The `string` argument

With `string` you can search for strings instead of tags. As with `name` and the keyword arguments, you can pass in *a string*, *a regular expression*, *a list*, *a function*, or *the value True*. Here are some examples:

```
soup.find_all(string="Elsie")
# [u'Elsie']

soup.find_all(string=["Tillie", "Elsie", "Lacie"])
# [u'Elsie', u'Lacie', u'Tillie']

soup.find_all(string=re.compile("Dormouse"))
[u"The Dormouse's story", u"The Dormouse's story"]

def is_the_only_string_within_a_tag(s):
    """Return True if this string is the only child of its parent tag."""
    return (s == s.parent.string)

soup.find_all(string=is_the_only_string_within_a_tag)
# [u"The Dormouse's story", u"The Dormouse's story", u'Elsie', u'Lacie', u'Tillie', u
↪'...']
```

Although `string` is for finding strings, you can combine it with arguments that find tags: Beautiful Soup will find all tags whose `.string` matches your value for `string`. This code finds the <a> tags whose `.string` is "Elsie":

```
soup.find_all("a", string="Elsie")
# [<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>]
```

The `string` argument is new in Beautiful Soup 4.4.0. In earlier versions it was called `text`:

```
soup.find_all("a", text="Elsie")
# [<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>]
```

The `limit` argument

`find_all()` returns all the tags and strings that match your filters. This can take a while if the document is large. If you don't need *all* the results, you can pass in a number for `limit`. This works just like the LIMIT keyword in SQL. It tells Beautiful Soup to stop gathering results after it's found a certain number.

There are three links in the "three sisters" document, but this code only finds the first two:

```
soup.find_all("a", limit=2)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

The `recursive` argument

If you call `mytag.find_all()`, Beautiful Soup will examine all the descendants of `mytag`: its children, its children's children, and so on. If you only want Beautiful Soup to consider direct children, you can pass in `recursive=False`. See the difference here:

```
soup.html.find_all("title")
# [<title>The Dormouse's story</title>]

soup.html.find_all("title", recursive=False)
# []
```

Here's that part of the document:

```
<html>
 <head>
  <title>
   The Dormouse's story
  </title>
 </head>
...
```

The <title> tag is beneath the <html> tag, but it's not *directly* beneath the <html> tag: the <head> tag is in the way. Beautiful Soup finds the <title> tag when it's allowed to look at all descendants of the <html> tag, but when `recursive=False` restricts it to the <html> tag's immediate children, it finds nothing.

Beautiful Soup offers a lot of tree-searching methods (covered below), and they mostly take the same arguments as `find_all()`: `name`, `attrs`, `string`, `limit`, and the keyword arguments. But the `recursive` argument is different: `find_all()` and `find()` are the only methods that support it. Passing `recursive=False` into a method like `find_parents()` wouldn't be very useful.

Calling a tag is like calling `find_all()`

Because `find_all()` is the most popular method in the Beautiful Soup search API, you can use a shortcut for it. If you treat the `BeautifulSoup` object or a `Tag` object as though it were a function, then it's the same as calling `find_all()` on that object. These two lines of code are equivalent:

```
soup.find_all("a")
soup("a")
```

These two lines are also equivalent:

```
soup.title.find_all(string=True)
soup.title(string=True)
```

`find()`

Signature: find(*name*, *attrs*, *recursive*, *string*, ***kwargs*)

The `find_all()` method scans the entire document looking for results, but sometimes you only want to find one result. If you know a document only has one <body> tag, it's a waste of time to scan the entire document looking for more. Rather than passing in `limit=1` every time you call `find_all`, you can use the `find()` method. These two lines of code are *nearly* equivalent:

```
soup.find_all('title', limit=1)
# [<title>The Dormouse's story</title>]

soup.find('title')
# <title>The Dormouse's story</title>
```

The only difference is that `find_all()` returns a list containing the single result, and `find()` just returns the result.

If `find_all()` can't find anything, it returns an empty list. If `find()` can't find anything, it returns `None`:

```
print(soup.find("nosuchtag"))
# None
```

Remember the `soup.head.title` trick from *Navigating using tag names*? That trick works by repeatedly calling `find()`:

```
soup.head.title
# <title>The Dormouse's story</title>

soup.find("head").find("title")
# <title>The Dormouse's story</title>
```

`find_parents()` and `find_parent()`

Signature: find_parents(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_parent(*name*, *attrs*, *string*, ***kwargs*)

I spent a lot of time above covering `find_all()` and `find()`. The Beautiful Soup API defines ten other methods for searching the tree, but don't be afraid. Five of these methods are basically the same as `find_all()`, and the other five are basically the same as `find()`. The only differences are in what parts of the tree they search.

First let's consider `find_parents()` and `find_parent()`. Remember that `find_all()` and `find()` work their way down the tree, looking at tag's descendants. These methods do the opposite: they work their way *up* the tree, looking at a tag's (or a string's) parents. Let's try them out, starting from a string buried deep in the "three daughters" document:

```
a_string = soup.find(string="Lacie")
a_string
# u'Lacie'

a_string.find_parents("a")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

a_string.find_parent("p")
# <p class="story">Once upon a time there were three little sisters; and their names␣
↪were
#  <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
#  and they lived at the bottom of a well.</p>

a_string.find_parents("p", class="title")
# []
```

One of the three <a> tags is the direct parent of the string in question, so our search finds it. One of the three <p> tags is an indirect parent of the string, and our search finds that as well. There's a <p> tag with the CSS class "title" *somewhere* in the document, but it's not one of this string's parents, so we can't find it with `find_parents()`.

You may have made the connection between `find_parent()` and `find_parents()`, and the *.parent* and *.parents* attributes mentioned earlier. The connection is very strong. These search methods actually use `.parents` to iterate over all the parents, and check each one against the provided filter to see if it matches.

`find_next_siblings()` and `find_next_sibling()`

Signature: find_next_siblings(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_next_sibling(*name*, *attrs*, *string*, ***kwargs*)

These methods use *.next_siblings* to iterate over the rest of an element's siblings in the tree. The `find_next_siblings()` method returns all the siblings that match, and `find_next_sibling()` only returns the first one:

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

first_link.find_next_siblings("a")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

first_story_paragraph = soup.find("p", "story")
first_story_paragraph.find_next_sibling("p")
# <p class="story">...</p>
```

find_previous_siblings() and find_previous_sibling()

Signature: find_previous_siblings(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_previous_sibling(*name*, *attrs*, *string*, ***kwargs*)

These methods use *.previous_siblings* to iterate over an element's siblings that precede it in the tree. The `find_previous_siblings()` method returns all the siblings that match, and `find_previous_sibling()` only returns the first one:

```
last_link = soup.find("a", id="link3")
last_link
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>

last_link.find_previous_siblings("a")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

first_story_paragraph = soup.find("p", "story")
first_story_paragraph.find_previous_sibling("p")
# <p class="title"><b>The Dormouse's story</b></p>
```

find_all_next() and find_next()

Signature: find_all_next(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_next(*name*, *attrs*, *string*, ***kwargs*)

These methods use *.next_elements* to iterate over whatever tags and strings that come after it in the document. The `find_all_next()` method returns all matches, and `find_next()` only returns the first match:

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

first_link.find_all_next(string=True)
# [u'Elsie', u',\n', u'Lacie', u' and\n', u'Tillie',
#  u';\nand they lived at the bottom of a well.', u'\n\n', u'...', u'\n']

first_link.find_next("p")
# <p class="story">...</p>
```

In the first example, the string "Elsie" showed up, even though it was contained within the <a> tag we started from. In the second example, the last <p> tag in the document showed up, even though it's not in the same part of the tree as the <a> tag we started from. For these methods, all that matters is that an element match the filter, and show up later in the document than the starting element.

find_all_previous() and find_previous()

Signature: find_all_previous(*name*, *attrs*, *string*, *limit*, ***kwargs*)

Signature: find_previous(*name*, *attrs*, *string*, ***kwargs*)

These methods use *.previous_elements* to iterate over the tags and strings that came before it in the document. The `find_all_previous()` method returns all matches, and `find_previous()` only returns the first match:

```
first_link = soup.a
first_link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>

first_link.find_all_previous("p")
# [<p class="story">Once upon a time there were three little sisters; ...</p>,
#  <p class="title"><b>The Dormouse's story</b></p>]

first_link.find_previous("title")
# <title>The Dormouse's story</title>
```

The call to `find_all_previous("p")` found the first paragraph in the document (the one with class="title"), but it also finds the second paragraph, the <p> tag that contains the <a> tag we started with. This shouldn't be too surprising: we're looking at all the tags that show up earlier in the document than the one we started with. A <p> tag that contains an <a> tag must have shown up before the <a> tag it contains.

CSS selectors

Beautiful Soup supports the most commonly-used CSS selectors. Just pass a string into the `.select()` method of a `Tag` object or the `BeautifulSoup` object itself.

You can find tags:

```
soup.select("title")
# [<title>The Dormouse's story</title>]

soup.select("p:nth-of-type(3)")
# [<p class="story">...</p>]
```

Find tags beneath other tags:

```
soup.select("body a")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"  id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select("html head title")
# [<title>The Dormouse's story</title>]
```

Find tags *directly* beneath other tags:

```
soup.select("head > title")
# [<title>The Dormouse's story</title>]

soup.select("p > a")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie"  id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select("p > a:nth-of-type(2)")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]

soup.select("p > #link1")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

```
soup.select("body > a")
# []
```

Find the siblings of tags:

```
soup.select("#link1 ~ .sister")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie"  id="link3">Tillie</a>]

soup.select("#link1 + .sister")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

Find tags by CSS class:

```
soup.select(".sister")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select("[class~=sister]")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

Find tags by ID:

```
soup.select("#link1")
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

soup.select("a#link2")
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

Find tags that match any selector from a list of selectors:

soup.select("#link1,#link2") # [Elsie,
Lacie]

Test for the existence of an attribute:

```
soup.select('a[href]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

Find tags by attribute value:

```
soup.select('a[href="http://example.com/elsie"]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

soup.select('a[href^="http://example.com/"]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select('a[href$="tillie"]')
# [<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

soup.select('a[href*=".com/el"]')
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

Match language codes:

```
multilingual_markup = """
 <p lang="en">Hello</p>
 <p lang="en-us">Howdy, y'all</p>
 <p lang="en-gb">Pip-pip, old fruit</p>
 <p lang="fr">Bonjour mes amis</p>
"""
multilingual_soup = BeautifulSoup(multilingual_markup)
multilingual_soup.select('p[lang|=en]')
# [<p lang="en">Hello</p>,
#  <p lang="en-us">Howdy, y'all</p>,
#  <p lang="en-gb">Pip-pip, old fruit</p>]
```

Find only the first tag that matches a selector:

```
soup.select_one(".sister")
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

This is all a convenience for users who know the CSS selector syntax. You can do all this stuff with the Beautiful Soup API. And if CSS selectors are all you need, you might as well use lxml directly: it's a lot faster, and it supports more CSS selectors. But this lets you *combine* simple CSS selectors with the Beautiful Soup API.

CHAPTER 8

Modifying the tree

Beautiful Soup's main strength is in searching the parse tree, but you can also modify the tree and write your changes as a new HTML or XML document.

Changing tag names and attributes

I covered this earlier, in *Attributes*, but it bears repeating. You can rename a tag, change the values of its attributes, add new attributes, and delete attributes:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')
tag = soup.b

tag.name = "blockquote"
tag['class'] = 'verybold'
tag['id'] = 1
tag
# <blockquote class="verybold" id="1">Extremely bold</blockquote>

del tag['class']
del tag['id']
tag
# <blockquote>Extremely bold</blockquote>
```

Modifying `.string`

If you set a tag's `.string` attribute, the tag's contents are replaced with the string you give:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)

tag = soup.a
```

```
tag.string = "New link text."
tag
# <a href="http://example.com/">New link text.</a>
```

Be careful: if the tag contained other tags, they and all their contents will be destroyed.

append()

You can add to a tag's contents with `Tag.append()`. It works just like calling `.append()` on a Python list:

```
soup = BeautifulSoup("<a>Foo</a>")
soup.a.append("Bar")

soup
# <html><head></head><body><a>FooBar</a></body></html>
soup.a.contents
# [u'Foo', u'Bar']
```

NavigableString() and .new_tag()

If you need to add a string to a document, no problem–you can pass a Python string in to `append()`, or you can call the `NavigableString` constructor:

```
soup = BeautifulSoup("<b></b>")
tag = soup.b
tag.append("Hello")
new_string = NavigableString(" there")
tag.append(new_string)
tag
# <b>Hello there.</b>
tag.contents
# [u'Hello', u' there']
```

If you want to create a comment or some other subclass of `NavigableString`, just call the constructor:

```
from bs4 import Comment
new_comment = Comment("Nice to see you.")
tag.append(new_comment)
tag
# <b>Hello there<!--Nice to see you.--></b>
tag.contents
# [u'Hello', u' there', u'Nice to see you.']
```

(This is a new feature in Beautiful Soup 4.4.0.)

What if you need to create a whole new tag? The best solution is to call the factory method `BeautifulSoup.new_tag()`:

```
soup = BeautifulSoup("<b></b>")
original_tag = soup.b

new_tag = soup.new_tag("a", href="http://www.example.com")
original_tag.append(new_tag)
```

```
original_tag
# <b><a href="http://www.example.com"></a></b>

new_tag.string = "Link text."
original_tag
# <b><a href="http://www.example.com">Link text.</a></b>
```

Only the first argument, the tag name, is required.

insert()

`Tag.insert()` is just like `Tag.append()`, except the new element doesn't necessarily go at the end of its parent's `.contents`. It'll be inserted at whatever numeric position you say. It works just like `.insert()` on a Python list:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
tag = soup.a

tag.insert(1, "but did not endorse ")
tag
# <a href="http://example.com/">I linked to but did not endorse <i>example.com</i></a>
tag.contents
# [u'I linked to ', u'but did not endorse', <i>example.com</i>]
```

insert_before() and insert_after()

The `insert_before()` method inserts a tag or string immediately before something else in the parse tree:

```
soup = BeautifulSoup("<b>stop</b>")
tag = soup.new_tag("i")
tag.string = "Don't"
soup.b.string.insert_before(tag)
soup.b
# <b><i>Don't</i>stop</b>
```

The `insert_after()` method moves a tag or string so that it immediately follows something else in the parse tree:

```
soup.b.i.insert_after(soup.new_string(" ever "))
soup.b
# <b><i>Don't</i> ever stop</b>
soup.b.contents
# [<i>Don't</i>, u' ever ', u'stop']
```

clear()

`Tag.clear()` removes the contents of a tag:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
tag = soup.a
```

```
tag.clear()
tag
# <a href="http://example.com/"></a>
```

extract()

`PageElement.extract()` removes a tag or string from the tree. It returns the tag or string that was extracted:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

i_tag = soup.i.extract()

a_tag
# <a href="http://example.com/">I linked to</a>

i_tag
# <i>example.com</i>

print(i_tag.parent)
None
```

At this point you effectively have two parse trees: one rooted at the `BeautifulSoup` object you used to parse the document, and one rooted at the tag that was extracted. You can go on to call `extract` on a child of the element you extracted:

```
my_string = i_tag.string.extract()
my_string
# u'example.com'

print(my_string.parent)
# None
i_tag
# <i></i>
```

decompose()

`Tag.decompose()` removes a tag from the tree, then *completely destroys it and its contents*:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

soup.i.decompose()

a_tag
# <a href="http://example.com/">I linked to</a>
```

replace_with()

`PageElement.replace_with()` removes a tag or string from the tree, and replaces it with the tag or string of your choice:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

new_tag = soup.new_tag("b")
new_tag.string = "example.net"
a_tag.i.replace_with(new_tag)

a_tag
# <a href="http://example.com/">I linked to <b>example.net</b></a>
```

`replace_with()` returns the tag or string that was replaced, so that you can examine it or add it back to another part of the tree.

wrap()

`PageElement.wrap()` wraps an element in the tag you specify. It returns the new wrapper:

```
soup = BeautifulSoup("<p>I wish I was bold.</p>")
soup.p.string.wrap(soup.new_tag("b"))
# <b>I wish I was bold.</b>

soup.p.wrap(soup.new_tag("div")
# <div><p><b>I wish I was bold.</b></p></div>
```

This method is new in Beautiful Soup 4.0.5.

unwrap()

`Tag.unwrap()` is the opposite of `wrap()`. It replaces a tag with whatever's inside that tag. It's good for stripping out markup:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
a_tag = soup.a

a_tag.i.unwrap()
a_tag
# <a href="http://example.com/">I linked to example.com</a>
```

Like `replace_with()`, `unwrap()` returns the tag that was replaced.

CHAPTER 9

Output

Pretty-printing

The `prettify()` method will turn a Beautiful Soup parse tree into a nicely formatted Unicode string, with each HTML/XML tag on its own line:

```
markup = '<a href="http://example.com/">I linked to <i>example.com</i></a>'
soup = BeautifulSoup(markup)
soup.prettify()
# '<html>\n <head>\n </head>\n <body>\n  <a href="http://example.com/">\n...'

print(soup.prettify())
# <html>
#  <head>
#  </head>
#  <body>
#   <a href="http://example.com/">
#    I linked to
#    <i>
#     example.com
#    </i>
#   </a>
#  </body>
# </html>
```

You can call `prettify()` on the top-level `BeautifulSoup` object, or on any of its `Tag` objects:

```
print(soup.a.prettify())
# <a href="http://example.com/">
#  I linked to
#  <i>
#   example.com
#  </i>
# </a>
```

Non-pretty printing

If you just want a string, with no fancy formatting, you can call `unicode()` or `str()` on a `BeautifulSoup` object, or a `Tag` within it:

```
str(soup)
# '<html><head></head><body><a href="http://example.com/">I linked to <i>example.com</
↪i></a></body></html>'

unicode(soup.a)
# u'<a href="http://example.com/">I linked to <i>example.com</i></a>'
```

The `str()` function returns a string encoded in UTF-8. See *Encodings* for other options.

You can also call `encode()` to get a bytestring, and `decode()` to get Unicode.

Output formatters

If you give Beautiful Soup a document that contains HTML entities like "&lquot;", they'll be converted to Unicode characters:

```
soup = BeautifulSoup("&ldquo;Dammit!&rdquo; he said.")
unicode(soup)
# u'<html><head></head><body>\u201cDammit!\u201d he said.</body></html>'
```

If you then convert the document to a string, the Unicode characters will be encoded as UTF-8. You won't get the HTML entities back:

```
str(soup)
# '<html><head></head><body>\xe2\x80\x9cDammit!\xe2\x80\x9d he said.</body></html>'
```

By default, the only characters that are escaped upon output are bare ampersands and angle brackets. These get turned into "&", "<", and ">", so that Beautiful Soup doesn't inadvertently generate invalid HTML or XML:

```
soup = BeautifulSoup("<p>The law firm of Dewey, Cheatem, & Howe</p>")
soup.p
# <p>The law firm of Dewey, Cheatem, &amp; Howe</p>

soup = BeautifulSoup('<a href="http://example.com/?foo=val1&bar=val2">A link</a>')
soup.a
# <a href="http://example.com/?foo=val1&amp;bar=val2">A link</a>
```

You can change this behavior by providing a value for the `formatter` argument to `prettify()`, `encode()`, or `decode()`. Beautiful Soup recognizes four possible values for `formatter`.

The default is `formatter="minimal"`. Strings will only be processed enough to ensure that Beautiful Soup generates valid HTML/XML:

```
french = "<p>Il a dit &lt;&lt;Sacr&eacute; bleu!&gt;&gt;</p>"
soup = BeautifulSoup(french)
print(soup.prettify(formatter="minimal"))
# <html>
#  <body>
#   <p>
#    Il a dit &lt;&lt;Sacré bleu!&gt;&gt;
#   </p>
```

```
#  </body>
# </html>
```

If you pass in `formatter="html"`, Beautiful Soup will convert Unicode characters to HTML entities whenever possible:

```
print(soup.prettify(formatter="html"))
# <html>
#  <body>
#   <p>
#    Il a dit &lt;&lt;Sacr&eacute; bleu!&gt;&gt;
#   </p>
#  </body>
# </html>
```

If you pass in `formatter=None`, Beautiful Soup will not modify strings at all on output. This is the fastest option, but it may lead to Beautiful Soup generating invalid HTML/XML, as in these examples:

```
print(soup.prettify(formatter=None))
# <html>
#  <body>
#   <p>
#    Il a dit <<Sacré bleu!>>
#   </p>
#  </body>
# </html>

link_soup = BeautifulSoup('<a href="http://example.com/?foo=val1&bar=val2">A link</a>
↪')
print(link_soup.a.encode(formatter=None))
# <a href="http://example.com/?foo=val1&bar=val2">A link</a>
```

Finally, if you pass in a function for `formatter`, Beautiful Soup will call that function once for every string and attribute value in the document. You can do whatever you want in this function. Here's a formatter that converts strings to uppercase and does absolutely nothing else:

```
def uppercase(str):
    return str.upper()

print(soup.prettify(formatter=uppercase))
# <html>
#  <body>
#   <p>
#    IL A DIT <<SACRÉ BLEU!>>
#   </p>
#  </body>
# </html>

print(link_soup.a.prettify(formatter=uppercase))
# <a href="HTTP://EXAMPLE.COM/?FOO=VAL1&BAR=VAL2">
#  A LINK
# </a>
```

If you're writing your own function, you should know about the `EntitySubstitution` class in the `bs4.dammit` module. This class implements Beautiful Soup's standard formatters as class methods: the "html" formatter is `EntitySubstitution.substitute_html`, and the "minimal" formatter is `EntitySubstitution.substitute_xml`. You can use these functions to simulate `formatter=html` or `formatter==minimal`, but

then do something extra.

Here's an example that replaces Unicode characters with HTML entities whenever possible, but *also* converts all strings to uppercase:

```
from bs4.dammit import EntitySubstitution
def uppercase_and_substitute_html_entities(str):
    return EntitySubstitution.substitute_html(str.upper())

print(soup.prettify(formatter=uppercase_and_substitute_html_entities))
# <html>
#  <body>
#   <p>
#    IL A DIT &lt;&lt;SACR&Eacute; BLEU!&gt;&gt;
#   </p>
#  </body>
# </html>
```

One last caveat: if you create a `CData` object, the text inside that object is always presented *exactly as it appears, with no formatting*. Beautiful Soup will call the formatter method, just in case you've written a custom method that counts all the strings in the document or something, but it will ignore the return value:

```
from bs4.element import CData
soup = BeautifulSoup("<a></a>")
soup.a.string = CData("one < three")
print(soup.a.prettify(formatter="xml"))
# <a>
#  <![CDATA[one < three]]>
# </a>
```

get_text()

If you only want the text part of a document or tag, you can use the `get_text()` method. It returns all the text in a document or beneath a tag, as a single Unicode string:

```
markup = '<a href="http://example.com/">\nI linked to <i>example.com</i>\n</a>'
soup = BeautifulSoup(markup)

soup.get_text()
u'\nI linked to example.com\n'
soup.i.get_text()
u'example.com'
```

You can specify a string to be used to join the bits of text together:

```
# soup.get_text("|")
u'\nI linked to |example.com|\n'
```

You can tell Beautiful Soup to strip whitespace from the beginning and end of each bit of text:

```
# soup.get_text("|", strip=True)
u'I linked to|example.com'
```

But at that point you might want to use the *.stripped_strings* generator instead, and process the text yourself:

```
[text for text in soup.stripped_strings]
# [u'I linked to', u'example.com']
```

CHAPTER 10

Specifying the parser to use

If you just need to parse some HTML, you can dump the markup into the `BeautifulSoup` constructor, and it'll probably be fine. Beautiful Soup will pick a parser for you and parse the data. But there are a few additional arguments you can pass in to the constructor to change which parser is used.

The first argument to the `BeautifulSoup` constructor is a string or an open filehandle–the markup you want parsed. The second argument is *how* you'd like the markup parsed.

If you don't specify anything, you'll get the best HTML parser that's installed. Beautiful Soup ranks lxml's parser as being the best, then html5lib's, then Python's built-in parser. You can override this by specifying one of the following:

- What type of markup you want to parse. Currently supported are "html", "xml", and "html5".
- The name of the parser library you want to use. Currently supported options are "lxml", "html5lib", and "html.parser" (Python's built-in HTML parser).

The section *Installing a parser* contrasts the supported parsers.

If you don't have an appropriate parser installed, Beautiful Soup will ignore your request and pick a different parser. Right now, the only supported XML parser is lxml. If you don't have lxml installed, asking for an XML parser won't give you one, and asking for "lxml" won't work either.

Differences between parsers

Beautiful Soup presents the same interface to a number of different parsers, but each parser is different. Different parsers will create different parse trees from the same document. The biggest differences are between the HTML parsers and the XML parsers. Here's a short document, parsed as HTML:

```
BeautifulSoup("<a><b /></a>")
# <html><head></head><body><a><b></b></a></body></html>
```

Since an empty tag is not valid HTML, the parser turns it into a tag pair.

Here's the same document parsed as XML (running this requires that you have lxml installed). Note that the empty tag is left alone, and that the document is given an XML declaration instead of being put into an <html> tag.:

```
BeautifulSoup("<a><b /></a>", "xml")
# <?xml version="1.0" encoding="utf-8"?>
# <a><b/></a>
```

There are also differences between HTML parsers. If you give Beautiful Soup a perfectly-formed HTML document, these differences won't matter. One parser will be faster than another, but they'll all give you a data structure that looks exactly like the original HTML document.

But if the document is not perfectly-formed, different parsers will give different results. Here's a short, invalid document parsed using lxml's HTML parser. Note that the dangling tag is simply ignored:

```
BeautifulSoup("<a></p>", "lxml")
# <html><body><a></a></body></html>
```

Here's the same document parsed using html5lib:

```
BeautifulSoup("<a></p>", "html5lib")
# <html><head></head><body><a><p></p></a></body></html>
```

Instead of ignoring the dangling tag, html5lib pairs it with an opening tag. This parser also adds an empty <head> tag to the document.

Here's the same document parsed with Python's built-in HTML parser:

```
BeautifulSoup("<a></p>", "html.parser")
# <a></a>
```

Like html5lib, this parser ignores the closing tag. Unlike html5lib, this parser makes no attempt to create a well-formed HTML document by adding a <body> tag. Unlike lxml, it doesn't even bother to add an <html> tag.

Since the document "<a> " is invalid, none of these techniques is the "correct" way to handle it. The html5lib parser uses techniques that are part of the HTML5 standard, so it has the best claim on being the "correct" way, but all three techniques are legitimate.

Differences between parsers can affect your script. If you're planning on distributing your script to other people, or running it on multiple machines, you should specify a parser in the `BeautifulSoup` constructor. That will reduce the chances that your users parse a document differently from the way you parse it.

CHAPTER 11

Encodings

Any HTML or XML document is written in a specific encoding like ASCII or UTF-8. But when you load that document into Beautiful Soup, you'll discover it's been converted to Unicode:

```
markup = "<h1>Sacr\xc3\xa9 bleu!</h1>"
soup = BeautifulSoup(markup)
soup.h1
# <h1>Sacré bleu!</h1>
soup.h1.string
# u'Sacr\xe9 bleu!'
```

It's not magic. (That sure would be nice.) Beautiful Soup uses a sub-library called *Unicode, Dammit* to detect a document's encoding and convert it to Unicode. The autodetected encoding is available as the `.original_encoding` attribute of the `BeautifulSoup` object:

```
soup.original_encoding
'utf-8'
```

Unicode, Dammit guesses correctly most of the time, but sometimes it makes mistakes. Sometimes it guesses correctly, but only after a byte-by-byte search of the document that takes a very long time. If you happen to know a document's encoding ahead of time, you can avoid mistakes and delays by passing it to the `BeautifulSoup` constructor as `from_encoding`.

Here's a document written in ISO-8859-8. The document is so short that Unicode, Dammit can't get a good lock on it, and misidentifies it as ISO-8859-7:

```
markup = b"<h1>\xed\xe5\xec\xf9</h1>"
soup = BeautifulSoup(markup)
soup.h1
<h1>νεμω</h1>
soup.original_encoding
'ISO-8859-7'
```

We can fix this by passing in the correct `from_encoding`:

```
soup = BeautifulSoup(markup, from_encoding="iso-8859-8")
soup.h1
<h1></h1>
soup.original_encoding
'iso8859-8'
```

If you don't know what the correct encoding is, but you know that Unicode, Dammit is guessing wrong, you can pass the wrong guesses in as `exclude_encodings`:

```
soup = BeautifulSoup(markup, exclude_encodings=["ISO-8859-7"])
soup.h1
<h1></h1>
soup.original_encoding
'WINDOWS-1255'
```

Windows-1255 isn't 100% correct, but that encoding is a compatible superset of ISO-8859-8, so it's close enough. (`exclude_encodings` is a new feature in Beautiful Soup 4.4.0.)

In rare cases (usually when a UTF-8 document contains text written in a completely different encoding), the only way to get Unicode may be to replace some characters with the special Unicode character "REPLACEMENT CHARACTER" (U+FFFD,). If Unicode, Dammit needs to do this, it will set the `.contains_replacement_characters` attribute to `True` on the `UnicodeDammit` or `BeautifulSoup` object. This lets you know that the Unicode representation is not an exact representation of the original–some data was lost. If a document contains , but `.contains_replacement_characters` is `False`, you'll know that the was there originally (as it is in this paragraph) and doesn't stand in for missing data.

Output encoding

When you write out a document from Beautiful Soup, you get a UTF-8 document, even if the document wasn't in UTF-8 to begin with. Here's a document written in the Latin-1 encoding:

```
markup = b'''
 <html>
  <head>
   <meta content="text/html; charset=ISO-Latin-1" http-equiv="Content-type" />
  </head>
  <body>
   <p>Sacr\xe9 bleu!</p>
  </body>
 </html>
'''

soup = BeautifulSoup(markup)
print(soup.prettify())
# <html>
#  <head>
#   <meta content="text/html; charset=utf-8" http-equiv="Content-type" />
#  </head>
#  <body>
#   <p>
#    Sacré bleu!
#   </p>
#  </body>
# </html>
```

Note that the <meta> tag has been rewritten to reflect the fact that the document is now in UTF-8.

If you don't want UTF-8, you can pass an encoding into `prettify()`:

```
print(soup.prettify("latin-1"))
# <html>
#  <head>
#   <meta content="text/html; charset=latin-1" http-equiv="Content-type" />
# ...
```

You can also call encode() on the `BeautifulSoup` object, or any element in the soup, just as if it were a Python string:

```
soup.p.encode("latin-1")
# '<p>Sacr\xe9 bleu!</p>'

soup.p.encode("utf-8")
# '<p>Sacr\xc3\xa9 bleu!</p>'
```

Any characters that can't be represented in your chosen encoding will be converted into numeric XML entity references. Here's a document that includes the Unicode character SNOWMAN:

```
markup = u"<b>\N{SNOWMAN}</b>"
snowman_soup = BeautifulSoup(markup)
tag = snowman_soup.b
```

The SNOWMAN character can be part of a UTF-8 document (it looks like), but there's no representation for that character in ISO-Latin-1 or ASCII, so it's converted into "&#9731" for those encodings:

```
print(tag.encode("utf-8"))
# <b></b>

print tag.encode("latin-1")
# <b>&#9731;</b>

print tag.encode("ascii")
# <b>&#9731;</b>
```

Unicode, Dammit

You can use Unicode, Dammit without using Beautiful Soup. It's useful whenever you have data in an unknown encoding and you just want it to become Unicode:

```
from bs4 import UnicodeDammit
dammit = UnicodeDammit("Sacr\xc3\xa9 bleu!")
print(dammit.unicode_markup)
# Sacré bleu!
dammit.original_encoding
# 'utf-8'
```

Unicode, Dammit's guesses will get a lot more accurate if you install the `chardet` or `cchardet` Python libraries. The more data you give Unicode, Dammit, the more accurately it will guess. If you have your own suspicions as to what the encoding might be, you can pass them in as a list:

```
dammit = UnicodeDammit("Sacr\xe9 bleu!", ["latin-1", "iso-8859-1"])
print(dammit.unicode_markup)
```

```
# Sacré bleu!
dammit.original_encoding
# 'latin-1'
```

Unicode, Dammit has two special features that Beautiful Soup doesn't use.

Smart quotes

You can use Unicode, Dammit to convert Microsoft smart quotes to HTML or XML entities:

```
markup = b"<p>I just \x93love\x94 Microsoft Word\x92s smart quotes</p>"

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="html").unicode_markup
# u'<p>I just &ldquo;love&rdquo; Microsoft Word&rsquo;s smart quotes</p>'

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="xml").unicode_markup
# u'<p>I just &#x201C;love&#x201D; Microsoft Word&#x2019;s smart quotes</p>'
```

You can also convert Microsoft smart quotes to ASCII quotes:

```
UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="ascii").unicode_markup
# u'<p>I just "love" Microsoft Word\'s smart quotes</p>'
```

Hopefully you'll find this feature useful, but Beautiful Soup doesn't use it. Beautiful Soup prefers the default behavior, which is to convert Microsoft smart quotes to Unicode characters along with everything else:

```
UnicodeDammit(markup, ["windows-1252"]).unicode_markup
# u'<p>I just \u201clove\u201d Microsoft Word\u2019s smart quotes</p>'
```

Inconsistent encodings

Sometimes a document is mostly in UTF-8, but contains Windows-1252 characters such as (again) Microsoft smart quotes. This can happen when a website includes data from multiple sources. You can use `UnicodeDammit.detwingle()` to turn such a document into pure UTF-8. Here's a simple example:

```
snowmen = (u"\N{SNOWMAN}" * 3)
quote = (u"\N{LEFT DOUBLE QUOTATION MARK}I like snowmen!\N{RIGHT DOUBLE QUOTATION␣
↪MARK}")
doc = snowmen.encode("utf8") + quote.encode("windows_1252")
```

This document is a mess. The snowmen are in UTF-8 and the quotes are in Windows-1252. You can display the snowmen or the quotes, but not both:

```
print(doc)
# I like snowmen!

print(doc.decode("windows-1252"))
# â~fâ~fâ~f"I like snowmen!"
```

Decoding the document as UTF-8 raises a `UnicodeDecodeError`, and decoding it as Windows-1252 gives you gibberish. Fortunately, `UnicodeDammit.detwingle()` will convert the string to pure UTF-8, allowing you to decode it to Unicode and display the snowmen and quote marks simultaneously:

```
new_doc = UnicodeDammit.detwingle(doc)
print(new_doc.decode("utf8"))
# “I like snowmen!”
```

`UnicodeDammit.detwingle()` only knows how to handle Windows-1252 embedded in UTF-8 (or vice versa, I suppose), but this is the most common case.

Note that you must know to call `UnicodeDammit.detwingle()` on your data before passing it into `BeautifulSoup` or the `UnicodeDammit` constructor. Beautiful Soup assumes that a document has a single encoding, whatever it might be. If you pass it a document that contains both UTF-8 and Windows-1252, it's likely to think the whole document is Windows-1252, and the document will come out looking like `â~ƒâ~ƒâ~ƒ“I like snowmen!”`.

`UnicodeDammit.detwingle()` is new in Beautiful Soup 4.1.0.

CHAPTER 12

Comparing objects for equality

Beautiful Soup says that two `NavigableString` or `Tag` objects are equal when they represent the same HTML or XML markup. In this example, the two tags are treated as equal, even though they live in different parts of the object tree, because they both look like "pizza":

```
markup = "<p>I want <b>pizza</b> and more <b>pizza</b>!</p>"
soup = BeautifulSoup(markup, 'html.parser')
first_b, second_b = soup.find_all('b')
print first_b == second_b
# True

print first_b.previous_element == second_b.previous_element
# False
```

If you want to see whether two variables refer to exactly the same object, use *is*:

```
print first_b is second_b
# False
```

CHAPTER 13

Copying Beautiful Soup objects

You can use `copy.copy()` to create a copy of any `Tag` or `NavigableString`:

```
import copy
p_copy = copy.copy(soup.p)
print p_copy
# <p>I want <b>pizza</b> and more <b>pizza</b>!</p>
```

The copy is considered equal to the original, since it represents the same markup as the original, but it's not the same object:

```
print soup.p == p_copy
# True

print soup.p is p_copy
# False
```

The only real difference is that the copy is completely detached from the original Beautiful Soup object tree, just as if `extract()` had been called on it:

```
print p_copy.parent
# None
```

This is because two different `Tag` objects can't occupy the same space at the same time.

CHAPTER 14

Parsing only part of a document

Let's say you want to use Beautiful Soup look at a document's <a> tags. It's a waste of time and memory to parse the entire document and then go over it again looking for <a> tags. It would be much faster to ignore everything that wasn't an <a> tag in the first place. The `SoupStrainer` class allows you to choose which parts of an incoming document are parsed. You just create a `SoupStrainer` and pass it in to the `BeautifulSoup` constructor as the `parse_only` argument.

(Note that *this feature won't work if you're using the html5lib parser*. If you use html5lib, the whole document will be parsed, no matter what. This is because html5lib constantly rearranges the parse tree as it works, and if some part of the document didn't actually make it into the parse tree, it'll crash. To avoid confusion, in the examples below I'll be forcing Beautiful Soup to use Python's built-in parser.)

SoupStrainer

The `SoupStrainer` class takes the same arguments as a typical method from *Searching the tree*: *name*, *attrs*, *string*, and ***kwargs*. Here are three `SoupStrainer` objects:

```
from bs4 import SoupStrainer

only_a_tags = SoupStrainer("a")

only_tags_with_id_link2 = SoupStrainer(id="link2")

def is_short_string(string):
    return len(string) < 10

only_short_strings = SoupStrainer(string=is_short_string)
```

I'm going to bring back the "three sisters" document one more time, and we'll see what the document looks like when it's parsed with these three `SoupStrainer` objects:

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
```

```
<body>
<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names␣
↪were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class="story">...</p>
"""

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_a_tags).prettify())
# <a class="sister" href="http://example.com/elsie" id="link1">
#  Elsie
# </a>
# <a class="sister" href="http://example.com/lacie" id="link2">
#  Lacie
# </a>
# <a class="sister" href="http://example.com/tillie" id="link3">
#  Tillie
# </a>

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_tags_with_id_link2).
↪prettify())
# <a class="sister" href="http://example.com/lacie" id="link2">
#  Lacie
# </a>

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_short_strings).
↪prettify())
# Elsie
# ,
# Lacie
# and
# Tillie
# ...
#
```

You can also pass a `SoupStrainer` into any of the methods covered in *Searching the tree*. This probably isn't terribly useful, but I thought I'd mention it:

```
soup = BeautifulSoup(html_doc)
soup.find_all(only_short_strings)
# [u'\n\n', u'\n\n', u'Elsie', u',\n', u'Lacie', u' and\n', u'Tillie',
#  u'\n\n', u'...', u'\n']
```

CHAPTER 15

Troubleshooting

diagnose()

If you're having trouble understanding what Beautiful Soup does to a document, pass the document into the `diagnose()` function. (New in Beautiful Soup 4.2.0.) Beautiful Soup will print out a report showing you how different parsers handle the document, and tell you if you're missing a parser that Beautiful Soup could be using:

```
from bs4.diagnose import diagnose
with open("bad.html") as fp:
    data = fp.read()
diagnose(data)

# Diagnostic running on Beautiful Soup 4.2.0
# Python version 2.7.3 (default, Aug  1 2012, 05:16:07)
# I noticed that html5lib is not installed. Installing it may help.
# Found lxml version 2.3.2.0
#
# Trying to parse your data with html.parser
# Here's what html.parser did with the document:
# ...
```

Just looking at the output of diagnose() may show you how to solve the problem. Even if not, you can paste the output of `diagnose()` when asking for help.

Errors when parsing a document

There are two different kinds of parse errors. There are crashes, where you feed a document to Beautiful Soup and it raises an exception, usually an `HTMLParser.HTMLParseError`. And there is unexpected behavior, where a Beautiful Soup parse tree looks a lot different than the document used to create it.

Almost none of these problems turn out to be problems with Beautiful Soup. This is not because Beautiful Soup is an amazingly well-written piece of software. It's because Beautiful Soup doesn't include any parsing code. Instead,

it relies on external parsers. If one parser isn't working on a certain document, the best solution is to try a different parser. See *Installing a parser* for details and a parser comparison.

The most common parse errors are `HTMLParser.HTMLParseError: malformed start tag` and `HTMLParser.HTMLParseError: bad end tag`. These are both generated by Python's built-in HTML parser library, and the solution is to *install lxml or html5lib.*

The most common type of unexpected behavior is that you can't find a tag that you know is in the document. You saw it going in, but `find_all()` returns `[]` or `find()` returns `None`. This is another common problem with Python's built-in HTML parser, which sometimes skips tags it doesn't understand. Again, the solution is to *install lxml or html5lib.*

Version mismatch problems

- `SyntaxError: Invalid syntax` (on the line `ROOT_TAG_NAME = u'[document]'`): Caused by running the Python 2 version of Beautiful Soup under Python 3, without converting the code.
- `ImportError: No module named HTMLParser` - Caused by running the Python 2 version of Beautiful Soup under Python 3.
- `ImportError: No module named html.parser` - Caused by running the Python 3 version of Beautiful Soup under Python 2.
- `ImportError: No module named BeautifulSoup` - Caused by running Beautiful Soup 3 code on a system that doesn't have BS3 installed. Or, by writing Beautiful Soup 4 code without knowing that the package name has changed to `bs4`.
- `ImportError: No module named bs4` - Caused by running Beautiful Soup 4 code on a system that doesn't have BS4 installed.

Parsing XML

By default, Beautiful Soup parses documents as HTML. To parse a document as XML, pass in "xml" as the second argument to the `BeautifulSoup` constructor:

```
soup = BeautifulSoup(markup, "xml")
```

You'll need to *have lxml installed*.

Other parser problems

- If your script works on one computer but not another, or in one virtual environment but not another, or outside the virtual environment but not inside, it's probably because the two environments have different parser libraries available. For example, you may have developed the script on a computer that has lxml installed, and then tried to run it on a computer that only has html5lib installed. See *Differences between parsers* for why this matters, and fix the problem by mentioning a specific parser library in the `BeautifulSoup` constructor.
- Because HTML tags and attributes are case-insensitive, all three HTML parsers convert tag and attribute names to lowercase. That is, the markup <TAG></TAG> is converted to <tag></tag>. If you want to preserve mixed-case or uppercase tags and attributes, you'll need to *parse the document as XML.*

Miscellaneous

- `UnicodeEncodeError: 'charmap' codec can't encode character u'\xfoo' in position bar` (or just about any other `UnicodeEncodeError`) - This is not a problem with Beautiful Soup. This problem shows up in two main situations. First, when you try to print a Unicode character that your console doesn't know how to display. (See this page on the Python wiki for help.) Second, when you're writing to a file and you pass in a Unicode character that's not supported by your default encoding. In this case, the simplest solution is to explicitly encode the Unicode string into UTF-8 with `u.encode("utf8")`.
- `KeyError: [attr]` - Caused by accessing `tag['attr']` when the tag in question doesn't define the `attr` attribute. The most common errors are `KeyError: 'href'` and `KeyError: 'class'`. Use `tag.get('attr')` if you're not sure `attr` is defined, just as you would with a Python dictionary.
- `AttributeError: 'ResultSet' object has no attribute 'foo'` - This usually happens because you expected `find_all()` to return a single tag or string. But `find_all()` returns a _list_ of tags and strings–a `ResultSet` object. You need to iterate over the list and look at the `.foo` of each one. Or, if you really only want one result, you need to use `find()` instead of `find_all()`.
- `AttributeError: 'NoneType' object has no attribute 'foo'` - This usually happens because you called `find()` and then tried to access the *.foo*' attribute of the result. But in your case, `find()` didn't find anything, so it returned `None`, instead of returning a tag or a string. You need to figure out why your `find()` call isn't returning anything.

Improving Performance

Beautiful Soup will never be as fast as the parsers it sits on top of. If response time is critical, if you're paying for computer time by the hour, or if there's any other reason why computer time is more valuable than programmer time, you should forget about Beautiful Soup and work directly atop lxml.

That said, there are things you can do to speed up Beautiful Soup. If you're not using lxml as the underlying parser, my advice is to *start*. Beautiful Soup parses documents significantly faster using lxml than using html.parser or html5lib.

You can speed up encoding detection significantly by installing the cchardet library.

Parsing only part of a document won't save you much time parsing the document, but it can save a lot of memory, and it'll make *searching* the document much faster.

CHAPTER 16

Beautiful Soup 3

Beautiful Soup 3 is the previous release series, and is no longer being actively developed. It's currently packaged with all major Linux distributions:

```
$ apt-get install python-beautifulsoup
```

It's also published through PyPi as `BeautifulSoup`.:

```
$ easy_install BeautifulSoup
```

```
$ pip install BeautifulSoup
```

You can also download a tarball of Beautiful Soup 3.2.0.

If you ran `easy_install beautifulsoup` or `easy_install BeautifulSoup`, but your code doesn't work, you installed Beautiful Soup 3 by mistake. You need to run `easy_install beautifulsoup4`.

The documentation for Beautiful Soup 3 is archived online.

Porting code to BS4

Most code written against Beautiful Soup 3 will work against Beautiful Soup 4 with one simple change. All you should have to do is change the package name from `BeautifulSoup` to `bs4`. So this:

```
from BeautifulSoup import BeautifulSoup
```

becomes this:

```
from bs4 import BeautifulSoup
```

- If you get the `ImportError` "No module named BeautifulSoup", your problem is that you're trying to run Beautiful Soup 3 code, but you only have Beautiful Soup 4 installed.
- If you get the `ImportError` "No module named bs4", your problem is that you're trying to run Beautiful Soup 4 code, but you only have Beautiful Soup 3 installed.

Although BS4 is mostly backwards-compatible with BS3, most of its methods have been deprecated and given new names for PEP 8 compliance. There are numerous other renames and changes, and a few of them break backwards compatibility.

Here's what you'll need to know to convert your BS3 code and habits to BS4:

You need a parser

Beautiful Soup 3 used Python's `SGMLParser`, a module that was deprecated and removed in Python 3.0. Beautiful Soup 4 uses `html.parser` by default, but you can plug in lxml or html5lib and use that instead. See *Installing a parser* for a comparison.

Since `html.parser` is not the same parser as `SGMLParser`, you may find that Beautiful Soup 4 gives you a different parse tree than Beautiful Soup 3 for the same markup. If you swap out `html.parser` for lxml or html5lib, you may find that the parse tree changes yet again. If this happens, you'll need to update your scraping code to deal with the new tree.

Method names

- `renderContents` -> `encode_contents`
- `replaceWith` -> `replace_with`
- `replaceWithChildren` -> `unwrap`
- `findAll` -> `find_all`
- `findAllNext` -> `find_all_next`
- `findAllPrevious` -> `find_all_previous`
- `findNext` -> `find_next`
- `findNextSibling` -> `find_next_sibling`
- `findNextSiblings` -> `find_next_siblings`
- `findParent` -> `find_parent`
- `findParents` -> `find_parents`
- `findPrevious` -> `find_previous`
- `findPreviousSibling` -> `find_previous_sibling`
- `findPreviousSiblings` -> `find_previous_siblings`
- `nextSibling` -> `next_sibling`
- `previousSibling` -> `previous_sibling`

Some arguments to the Beautiful Soup constructor were renamed for the same reasons:

- `BeautifulSoup(parseOnlyThese=...)` -> `BeautifulSoup(parse_only=...)`
- `BeautifulSoup(fromEncoding=...)` -> `BeautifulSoup(from_encoding=...)`

I renamed one method for compatibility with Python 3:

- `Tag.has_key()` -> `Tag.has_attr()`

I renamed one attribute to use more accurate terminology:

- `Tag.isSelfClosing` -> `Tag.is_empty_element`

I renamed three attributes to avoid using words that have special meaning to Python. Unlike the others, these changes are *not backwards compatible*. If you used these attributes in BS3, your code will break on BS4 until you change them.

- `UnicodeDammit.unicode` -> `UnicodeDammit.unicode_markup`
- `Tag.next` -> `Tag.next_element`
- `Tag.previous` -> `Tag.previous_element`

Generators

I gave the generators PEP 8-compliant names, and transformed them into properties:

- `childGenerator()` -> `children`
- `nextGenerator()` -> `next_elements`
- `nextSiblingGenerator()` -> `next_siblings`
- `previousGenerator()` -> `previous_elements`
- `previousSiblingGenerator()` -> `previous_siblings`
- `recursiveChildGenerator()` -> `descendants`
- `parentGenerator()` -> `parents`

So instead of this:

```
for parent in tag.parentGenerator():
    ...
```

You can write this:

```
for parent in tag.parents:
    ...
```

(But the old code will still work.)

Some of the generators used to yield `None` after they were done, and then stop. That was a bug. Now the generators just stop.

There are two new generators, *.strings and .stripped_strings*. `.strings` yields NavigableString objects, and `.stripped_strings` yields Python strings that have had whitespace stripped.

XML

There is no longer a `BeautifulStoneSoup` class for parsing XML. To parse XML you pass in "xml" as the second argument to the `BeautifulSoup` constructor. For the same reason, the `BeautifulSoup` constructor no longer recognizes the `isHTML` argument.

Beautiful Soup's handling of empty-element XML tags has been improved. Previously when you parsed XML you had to explicitly say which tags were considered empty-element tags. The `selfClosingTags` argument to the constructor is no longer recognized. Instead, Beautiful Soup considers any empty tag to be an empty-element tag. If you add a child to an empty-element tag, it stops being an empty-element tag.

Entities

An incoming HTML or XML entity is always converted into the corresponding Unicode character. Beautiful Soup 3 had a number of overlapping ways of dealing with entities, which have been removed. The `BeautifulSoup` constructor no longer recognizes the `smartQuotesTo` or `convertEntities` arguments. (*Unicode, Dammit* still has `smart_quotes_to`, but its default is now to turn smart quotes into Unicode.) The constants `HTML_ENTITIES`, `XML_ENTITIES`, and `XHTML_ENTITIES` have been removed, since they configure a feature (transforming some but not all entities into Unicode characters) that no longer exists.

If you want to turn Unicode characters back into HTML entities on output, rather than turning them into UTF-8 characters, you need to use an *output formatter*.

Miscellaneous

Tag.string now operates recursively. If tag A contains a single tag B and nothing else, then A.string is the same as B.string. (Previously, it was None.)

Multi-valued attributes like `class` have lists of strings as their values, not strings. This may affect the way you search by CSS class.

If you pass one of the `find*` methods both *string* *and* a tag-specific argument like *name*, Beautiful Soup will search for tags that match your tag-specific criteria and whose *Tag.string* matches your value for *string*. It will *not* find the strings themselves. Previously, Beautiful Soup ignored the tag-specific arguments and looked for strings.

The `BeautifulSoup` constructor no longer recognizes the *markupMassage* argument. It's now the parser's responsibility to handle markup correctly.

The rarely-used alternate parser classes like `ICantBelieveItsBeautifulSoup` and `BeautifulSOAP` have been removed. It's now the parser's decision how to handle ambiguous markup.

The `prettify()` method now returns a Unicode string, not a bytestring.