# Beautiful Soup Documentation



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 you want to learn about the differences between Beautiful Soup 3 and Beautiful Soup 4, see Porting code to BS4.

## Getting help

If you have questions about Beautiful Soup, or run into problems, send mail to the discussion group.

# **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*:

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

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html doc)
print(soup.prettify())
# <html>
  <head>
   <title>
   The Dormouse's story
  </title>
# </head>
#
  <body>
#
  #
    \langle b \rangle
#
    The Dormouse's story
#
    </b>
#
   #
   #
   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>
#
#
    <a class="sister" href="http://example.com/tillie" id="link2">
#
    Tillie
#
    </a>
#
    ; and they lived at the bottom of a well.
#
   #
   #
   </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
# <b>The Dormouse's story</b>
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.

# 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-beautifulsoup4
```

Beautiful Soup 4 is published through PyPi, so if you can't install it with the system packager, you can install it with <code>easy\_install</code> or <code>pip</code>. The package name is <code>beautifulsoup4</code>, and the same package works on Python 2 and 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 <code>easy\_install</code> or <code>pip</code> installed, you can download the Beautiful Soup 4 source tarball and install it with <code>setup.py</code>.

```
$ 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

If you're using Python 2, 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")	<ul> <li>Batteries included</li> <li>Decent speed</li> <li>Lenient (as of Python 2.7.3 and 3.2.)</li> </ul>	•
lxml's HTML parser	BeautifulSoup(markup, "lxml")	<ul><li>Very fast</li><li>Lenient</li></ul>	<ul> <li>External C dependency</li> </ul>
lxml's XML parser	<pre>BeautifulSoup(markup, ["lxml", "xml"]) BeautifulSoup(markup, "xml")</pre>	<ul><li>Very fast</li><li>The only currently supported XML</li></ul>	External C dependency

html5lib	BeautifulSoup(markup, html5lib)	<ul> <li>Extremely lenient</li> </ul>	<ul><li>Very slow</li></ul>
		<ul><li>Parses pages</li></ul>	<ul> <li>External</li> </ul>
		the same way a web browser does • Creates valid HTML5	Python dependency • Python 2 only

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.

# 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
soup = BeautifulSoup(open("index.html"))
soup = BeautifulSoup("<html>data</html>")
```

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

```
BeautifulSoup("Sacré 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.)

# 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.

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 class="boldest"> has an attribute "class" whose value is "boldest". You can access a tag's attributes by treating the tag like a dictionary:

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

You can access that dictionary directly as .attrs:

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

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

```
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>

tag['class']
# KeyError: 'class'
print(tag.get('class'))
# 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('')
css_soup.p['class']
# ["body", "strikeout"]

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

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('')
id_soup.p['id']
# 'my id'
```

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

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

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

```
xml_soup = BeautifulSoup('', '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.

### 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>
```

# Navigating the tree

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

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 <a href="https://example.com/html">https://example.com/html</a> 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</title>
```

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' \setminus n \setminus n'
# u"The Dormouse's story"
\# u' \setminus n \setminus 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' \setminus n \setminus 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"Once upon a time there were three little sisters; and their names were'
# u'Elsie'
# u','
# u'Lacie'
# u'and'
# 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>
```

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

And the .parent of a Beautiful Soup 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
# c/b>
# text2
# c/c>
# dext2
# c/c>
# c/body>
# c/btml>
```

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 <code>.next\_sibling</code> 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>
<b>The Dormouse's story</b>
```

An HTML parser takes this string of characters and turns it into a series of events: "open an <a href="https://example.com/html">httml> tag"</a>, "open a <a href="https://example.com/html">https://example.com/html> tag"</a>, "open a <a href="https://example.com/html">https://example.com/html</a>, "open a <a href="https://example.

"open a 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 </a> 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>
```

```
. {\tt next\_elements} and . {\tt 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'
# ...
# u'...'
# u'\n'
# None
```

# 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 explain 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>

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

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.

class="story">...
"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc)
```

By passing in a filter to an argument like find\_all(), you can isolate whatever parts of the document you're interested.

## Kinds of filters

Before talking in detail about <code>find\_all()</code> 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>]
```

## A regular expression

If you pass in a regular expression object, Beautiful Soup will filter against that regular expression. 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
```

## 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_key('class') and not tag.has_key('id')
```

Pass this function into find\_all() and you'll pick up all the tags:

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

This function only picks up the 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".

Here's a function that returns True if a tag is surrounded by string objects:

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

```
find all()
```

Signature: find\_all(name, attrs, recursive, text, 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")
# [<b>The Dormouse's story</b>]
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/lacie" 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(text=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 text, or id? Why does find\_all("p", "title") find a 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 that have an id attribute, 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>]
```

## Searching by CSS class

Instead of using keyword arguments, you can filter tags based on their attributes by passing a dictionary in for attrs. These two lines of code are equivalent:

```
soup.find_all(href=re.compile("elsie"), id='link1')
soup.find_all(attrs={'href' : re.compile("elsie"), 'id': 'link1'})
```

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

You can use attrs to search by CSS 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>]
```

But that's a lot of code for such a common operation. Instead, you can pass a string *attrs* instead of a dictionary. The string will be used to restrict the CSS class:

```
soup.find_all("a", "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>]
```

You can also pass in a regular expression, a function or True. Anything you pass in for attrs that's not a dictionary will be used to search against the CSS class:

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

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

soup.find_all(attrs=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('')
css_soup.find_all("p", "strikeout")
# []

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

Searching for the string value of the class attribute won't work:

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

## The text argument

With text 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(text="Elsie")
# [u'Elsie']

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

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

```
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, text, 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 Beautiful Soup 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(text=True)
soup.title(text=True)
```

### find()

Signature: find(name, attrs, recursive, text, \*\*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, text, limit, \*\*kwargs)

Signature: find\_parent(name, attrs, text, \*\*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 <code>find\_parents()</code> and <code>find\_parent()</code>. Remember that <code>find\_all()</code> and <code>find()</code> work their way down the tree, looking at tag's descendants. These methods do the opposite: they work their way <code>up</code> 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(text="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")
# Once upon a time there were three little sisters; and their names we described 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.

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 tags is an indirect parent of the string, and our search finds that as well. There's a 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 <code>find\_parent()</code> and <code>find\_parents()</code>, and the .parent and .parents attributes mentioned earlier. The connection is very strong. These search methods actually use <code>.parents</code> 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, text, limit, \*\*kwargs)

Signature: find\_next\_sibling(name, attrs, text, \*\*kwargs)

These methods use <code>.next\_siblings</code> to iterate over the rest of an element's siblings in the tree. The <code>find\_next\_siblings()</code> method returns all the siblings that match, and <code>find\_next\_sibling()</code> 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")
# ...
```

## find previous siblings() and find previous sibling()

Signature: find\_previous\_siblings(name, attrs, text, limit, \*\*kwargs)

Signature: find\_previous\_sibling(name, attrs, text, \*\*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")
# <b>The Dormouse's story</b>
```

## find\_all\_next() and find\_next()

Signature: find\_all\_next(name, attrs, text, limit, \*\*kwargs)

Signature: find\_next(name, attrs, text, \*\*kwargs)

These methods use <code>.next\_elements</code> to iterate over whatever tags and strings that come after it in the document. The <code>find\_all\_next()</code> method returns all matches, and <code>find\_next()</code> 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(text=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")
# ...
```

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 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, text, limit, \*\*kwargs)

Signature: find\_previous(name, attrs, text, \*\*kwargs)

These methods use *.previous\_elements* to iterate over the tags and strings that came before it in the document. The <code>find\_all\_previous()</code> method returns all matches, and <code>find\_previous()</code> 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")
# [Once upon a time there were three little sisters; ...,
# <b>The Dormouse's story</b>]
first_link.find_previous("title")
# <title>The Dormouse's story</title>
```

The call to <code>find\_all\_previous("p")</code> found the first paragraph in the document (the one with class="title"), but it also finds the second paragraph, the 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 tag that contains an <a> tag must have shown up before the <a> tag it contains.</code>

### CSS selectors

Beautiful Soup supports a subset of the CSS selector standard. Just construct the selector as a string and pass it into the .select() method of a Tag or the BeautifulSoup object itself.

#### You can find tags:

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

#### 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("body > 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>]
```

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

This is 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, because it's faster. But this lets you *combine* simple CSS selectors with the Beautiful Soup API.

# 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']
```

## BeautifulSoup.new string() 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 factory method BeautifulSoup.new\_string():

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

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>
# <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 <code>insert\_before()</code> 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>></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("I wish I was bold.")
soup.p.string.wrap(soup.new_tag("b"))
# <b>I wish I was bold.</b>

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

(In earlier versions of Beautiful Soup, unwrap() was called replace\_with\_children(), and that name will still work.)

# Output

## **Pretty-printing**

The prettify() method will turn a Beautiful Soup parse tree into a nicely formatted bytestring, 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
     example.com
    </i>
  </a>
  </body>
# </html>
```

You can call prettify() on the top-level Beautiful Soup 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>
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 <code>encode()</code> to get a bytestring, and <code>decode()</code> 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(""Dammit!" 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 "&gt;", so that Beautiful Soup doesn't inadvertently generate invalid HTML or XML:

```
soup = BeautifulSoup("The law firm of Dewey, Cheatem, & Howe")
soup.p
# The law firm of Dewey, Cheatem, & amp; Howe
soup = BeautifulSoup('<a href="http://example.com/?foo=vall&bar=val2">A link</a>')
soup.a
# <a href="http://example.com/?foo=vall&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 = "Il a dit <&lt;Sacr&eacute; bleu!&gt;&gt;"
soup = BeautifulSoup(french)
print(soup.prettify(formatter="minimal"))
# <html>
# <body>
# 
# Il a dit &lt;&lt;Sacré bleu!&gt;&gt;
# 
# </body>
# </body>
# </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>
# 
# Il a dit &lt;&lt;Sacr&eacute; bleu!&gt;&gt;
# 
# </body>
# </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>
# 
# Il a dit <<Sacré bleu!>>
# 
# </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>
```

) **4** | **b** |

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>
# 
# IL A DIT <<SACRÉ BLEU!>>
# 
# 
# </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>
# 
# IL A DIT &lt;&lt;SACR&Eacute; BLEU!&gt;&gt;
# 
# </body>
# </bo
```

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>

If you only want the text part of a document or tag, you can use the <code>get\_text()</code> 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']
```

# 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 <b /> tag is not valid HTML, the parser turns it into a <b></b> tag pair.

Here's the same document parsed as XML (running this requires that you have lxml installed). Note that the empty <b /> 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>", "lxml")
# <html><body><a></a></body></html>
```

Here's the same document parsed using html5lib:

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

## **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>סולש</h1>
soup.original_encoding
'iso8859-8'
```

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>
  Sacr\xe9 bleu!
  </body>
 </html>
soup = BeautifulSoup(markup)
print(soup.prettify())
# <html>
  <head>
   <meta content="text/html; charset=utf-8" http-equiv="Content-type" />
# </head>
# <body>
   >
   Sacré bleu!
  # </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")
# 'Sacr\xe9 bleu!'
soup.p.encode("utf-8")
# 'Sacr\xc3\xa9 bleu!'
```

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  $ext{@}$ ), 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'
```

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 one special feature that Beautiful Soup doesn't use. You can use it to convert Microsoft smart quotes to HTML or XML entities:

```
markup = b"I just \x93love\x94 Microsoft Word\x92s smart quotes"

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="html").unicode_markup
# u'I just "love" Microsoft Word's smart quotes'

UnicodeDammit(markup, ["windows-1252"], smart_quotes_to="xml").unicode_markup
# u'I just "love" Microsoft Word's smart quotes'
```

You can also convert Microsoft smart quotes to ASCII quotes:

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

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'I just \u201clove\u201d Microsoft Word\u2019s smart quotes'
```

# 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, text, 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(text=is_short_string)</pre>
```

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>
<b>The Dormouse's story</b>
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.
...
0.00
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 class="sister" href="http://example.com/tillie" id="link3">
# Tillie
# </a>
print(BeautifulSoup(html_doc, "html.parser", parse_only=only_tags_with_id_link2).pretti
# <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
                                                                               ) 4 | b |
```

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']
```

## **Troubleshooting**

### 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 Beautiful Soup 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, it's probably because the two computers 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.
- If you can't find a tag that you know is in the document (that is, find\_all() returned [] or find() returned None), you're probably using Python's built-in HTML parser, which sometimes skips tags it doesn't understand. Solution: Install Ixml or html5lib.

### Miscellaneous

- 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.
- 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: you're trying to print a Unicode character that your console doesn't know how to display. See this

page on the Python wiki for help. One easy solution is to write the text to a file and then look at the file.

## 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 |xml.

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.

Sometimes Unicode, Dammit can only detect the encoding of a file by doing a byte-by-byte examination of the file. This slows Beautiful Soup to a crawl. My tests indicate that this only happened on 2.x versions of Python, and that it happened most often with documents using Russian or Chinese encodings. If this is happening to you, you can fix it by using Python 3 for your script. Or, if you happen to know a document's encoding, you can pass it into the BeautifulSoup Constructor as from\_encoding.

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.

## 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. If your first language is Chinese, it might be easier for you to read the Chinese translation of the Beautiful Soup 3 documentation, then read this document to find out about the changes made in Beautiful Soup 4.

### 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, it will treat invalid markup differently. Usually the "difference" is that html.parser crashes. In that case, you'll need to install another parser. But sometimes html.parser just creates a different parse tree than SGMLParser would. If this happens, you may need to update your BS3 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.)

If you want to turn those 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 text 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 *text*. 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 <code>ICantBelieveItsBeautifulsoup</code> and <code>Beautifulsoap</code> have been removed. It's now the parser's decision how to handle ambiguous markup.