		speed • Lenient (as of Python 2.7.3 and 3.2.)	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.

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("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.)

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

css_soup = BeautifulSoup('<p class="body"></p>')
```

```
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('<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>
```

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>
```

Navigating the tree

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

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

<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)
```

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

```
# u';\nand they lived at the bottom of a well.'
# u'\n\n'
# <p class="story">...</p>
# 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 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>

<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)
```

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 `match()` 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".

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*, *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")
# [<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(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 <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>]
```

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 `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 that find 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 `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(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 `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(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")
# <p class="story">Once upon a time there were three little sisters; and their names wer
#  <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*, *text*, *limit*, ***kwargs*)

Signature: find_next_sibling(*name*, *attrs*, *text*, ***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*, *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")
```

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

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 *.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(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")
# <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*, *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 `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>]
```

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>]
```

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: it's a lot faster, and it supports more CSS selectors. 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>
```