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.

This document covers Beautiful Soup version 4.13.3. The examples in this documentation were written for Python 3.8.

You might be looking for the documentation for Beautiful Soup 3. If so, you should know that Beautiful Soup 3 is no longer being developed and that all support for it was dropped on December 31, 2020. If you want to learn about the differences between Beautiful Soup 3 and Beautiful Soup 4, see Porting code to BS4.

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

- 这篇文档当然还有中文版.
- このページは日本語で利用できます(外部リンク)
- 이 문서는 한국어 번역도 가능합니다.
- Este documento também está disponível em Português do Brasil.
- Este documento también está disponible en una traducción al español.
- Эта документация доступна на русском языке.

Getting help

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

When reporting an error in this documentation, please mention which translation you're reading.

API documentation

This document is written like an instruction manual, but you can also read traditional API documentation generated from the Beautiful Soup source code. If you want details about Beautiful Soup's internals, or a feature not covered in this document, try the API documentation.

Quick Start

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

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

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

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

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

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

print(soup.prettify())
# <html>
#  <head>
#   <title>
#    The Dormouse's story
#   </title>
#  </head>
#  <body>
#   <p class="title">
#    <b>
#     The Dormouse's story
#    </b>
#   </p>
#   <p class="story">
#    Once upon a time there were three little sisters; and their names were
#    <a class="sister" href="http://example.com/elsie" id="link1">
#     Elsie
#    </a>
#    ,
#    <a class="sister" href="http://example.com/lacie" id="link2">
#     Lacie
#    </a>
#    and
#    <a class="sister" href="http://example.com/tillie" id="link3">
#     Tillie
#    </a>
#    ; and they lived at the bottom of a well.
#   </p>
```

Quick search

Here are some simple ways to navigate that data structure:

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

soup.title.name
# u'title'

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

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

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

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

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

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

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

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

```
for link in soup.find_all('a'):
    print(link.get('href'))
# http://example.com/elsie
# http://example.com/lacie
# http://example.com/tillie
```

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

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

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

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 python3-bs4
```

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

```
$ easy_install beautifulsoup4
```

```
$ pip install beautifulsoup4
```

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

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

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

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

I use Python 3.10 to develop Beautiful Soup, but it should work with other recent versions.

Installing a parser

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

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

```
$ easy_install lxml
```

```
$ pip install lxml
```

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

```
$ apt-get install python3-html5lib
```

```
$ pip install html5lib
```

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

Parser	Typical usage	Advantages	Disadvantages
Python's html.parser	`BeautifulSoup(markup, "html.parser")`	• Batteries included • Decent speed	• Not as fast as lxml, less lenient than html5lib.
lxml's HTML parser	`BeautifulSoup(markup, "lxml")`	• Very fast	• 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.

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:

```python
from bs4 import BeautifulSoup

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

soup = BeautifulSoup("<html>a web page</html>", 'html.parser')
```

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

```python
print(BeautifulSoup("<html><head></head><body>Sacr&eacute; bleu!</body></html>"
# <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`. These objects represent the HTML *elements* that comprise the page.

class `Tag`

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

```python
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>', 'html.parser'
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 methods of a tag are for accessing its name and attributes.

`name`

Every tag has a name:

```python
tag.name
# 'b'
```

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

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

`attrs`

An HTML or XML tag may have any number of attributes. The tag `<b`

tag's attributes by treating the tag like a dictionary:

```
tag = BeautifulSoup('<b id="boldest">bold</b>', 'html.parser').b
tag['id']
# 'boldest'
```

You can access the dictionary of attributes directly as `.attrs`:

```
tag.attrs
# {'id': 'boldest'}
tag.attrs.keys()
# dict_keys(['id'])
```

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

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

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

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

Multi-valued attributes

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

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

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

When you turn a tag back into a string, the values of any multi-valued attributes are consolidated:

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

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 stores it as a simple string:

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

You can force all attributes to be stored as strings by passing `multi_valued_attributes=None` as a keyword argument into the `BeautifulSoup` constructor:

```
no_list_soup = BeautifulSoup('<p class="body strikeout"></p>', 'html.par
no_list_soup.p['class']
# 'body strikeout'
```

You can use `get_attribute_list` to always return the value in a list container, whether it's a string or multi-valued attribute value:

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

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

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

Again, you can configure this using the `multi_valued_attributes` argument:

```
class_is_multi= { '*' : 'class'}
xml_soup = BeautifulSoup('<p class="body strikeout"></p>', 'xml', multi_
xml_soup.p['class']
# ['body', 'strikeout']
```

You probably won't need to do this, but if you do, use the defaults as a guide. They implement the rules described in the HTML specification:

```
from bs4.builder import builder_registry
```

```
            builder_registry.lookup('html').DEFAULT_CDATA_LIST_ATTRIBUTES
```

class **NavigableString**

A tag can contain strings as pieces of text. Beautiful Soup uses the **NavigableString** class to contain these pieces of text:

```
soup = BeautifulSoup('<b class="boldest">Extremely bold</b>', 'html.parser')
tag = soup.b
tag.string
# '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 `str`:

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

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
# <b class="boldest">No longer bold</b>
```

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.

The **BeautifulSoup** object represents the parsed 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](#).

You can also pass a **BeautifulSoup** object into one of the methods defined in [Modifying the tree](#), just as you would a **Tag**. This lets you do things like combine two parsed documents:

```
doc = BeautifulSoup("<document><content/>INSERT FOOTER HERE</document>", "xml")
footer = BeautifulSoup("<footer>Here's the footer</footer>", "xml")
doc.find(text="INSERT FOOTER HERE").replace_with(footer)
# 'INSERT FOOTER HERE'
print(doc)
# <?xml version="1.0" encoding="utf-8"?>
# <document><content/><footer>Here's the footer</footer></document>
```

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 reference its `.name` (such as when writing code that works with both **Tag** and **BeautifulSoup** objects), so it's been given the special `.name` "[document]":

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

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 main one you'll probably encounter is the **Comment**.

class **Comment**

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

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

```
comment
# '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>
```

For HTML documents

Beautiful Soup defines a few **NavigableString** subclasses to contain strings found inside specific HTML tags. This makes it easier to pick out the main body of the page, by

ignoring strings that probably represent programming directives found within the page. (These classes are new in Beautiful Soup 4.9.0, and the html5lib parser doesn't use them.)

class **Stylesheet**

A `NavigableString` subclass that represents embedded CSS stylesheets; that is, any strings found inside a `<style>` tag during document parsing.

class **Script**

A `NavigableString` subclass that represents embedded Javascript; that is, any strings found inside a `<script>` tag during document parsing.

class **Template**

A `NavigableString` subclass that represents embedded HTML templates; that is, any strings found inside a `<template>` tag during document parsing.

For XML documents

Beautiful Soup defines some `NavigableString` classes for holding special types of strings that can be found in XML documents. Like `Comment`, these classes are subclasses of `NavigableString` that add something extra to the string on output.

class **Declaration**

A `NavigableString` subclass representing the [declaration](#) at the beginning of an XML document.

class **Doctype**

A `NavigableString` subclass representing the [document type declaration](#) which may be found near the beginning of an XML document.

class **CData**

A `NavigableString` subclass that represents a [CData section](#).

class **ProcessingInstruction**

A `NavigableString` subclass that represents the contents of an [XML processing instruction](#).

Navigating the tree

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

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

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

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

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

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

Going down

Tags may contain strings and more 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 find a tag by name. To do this, you can use the `find()` method:

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

For convenience, just saying the name of the tag you want is equivalent to `find()` (if no built-in attribute has that name). 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 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>
```

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

If you need to get *all* the <a> tags, you can use `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>]
```

For more complicated tasks, such as pattern-matching and filtering, you can use the methods described in Searching the tree.

.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
# ['The Dormouse's story']
```

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

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

If you want to modify a tag's children, use the methods described in Modifying the tree. Don't modify the the `.contents` list directly: that can lead to problems that are subtle and difficult to spot.

.descendants

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

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

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

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

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

```
len(list(soup.children))
# 1
len(list(soup.descendants))
# 26
```

.string

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

```
title_tag.string
# '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
# '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 to see all descendant strings:

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

Newlines and spaces that separate tags are also strings. You can remove extra whitespace by using the `.stripped_strings` generator instead:

```
for string in soup.stripped_strings:
    print(repr(string))
# "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'
# ';\n and they lived at the bottom of a well.'
# '...'
```

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

Going up

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

`.parent`

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

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

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

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

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

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

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

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

`.parents`

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

```
link = soup.a
link
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
for parent in link.parents:
    print(parent.name)
# p
# body
# html
# [document]
```

.self_and_parents

The `.self_and_parents` generator is a variant of `.parents` which gives you the entire ancestry of an element, including the element itself:

```
for parent in link.parents:
    print(parent.name)
# link
# p
# body
# html
# [document]
```

Going sideways

Consider a simple document like this:

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

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
# '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> and
# <a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
```

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

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

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

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

```
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))
# ',\n'
# <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
# ' and\n'
# <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
# '; and they lived at the bottom of a well.'

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>
# ',\n'
# <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
# 'Once upon a time there were three little sisters; and their names were\n'
```

(If the argument syntax to find tags by their attribute value is unfamiliar, don't worry; this is covered later in The keyword arguments.)

Going back and forth

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

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

An HTML parser takes this string of characters and turns it into a series of events: "open an <html> tag", "open a <head> tag", "open a <title> tag", "add a string", "close the <title> tag", "open a <p> tag", and so on. The order in which the opening tags and strings are encountered is called *document order*. Beautiful Soup offers tools for searching a document's elements in document order.

.next_element and .previous_element

The `.next_element` attribute of a string or tag points to whatever was parsed immediately after the opening of the current tag or after the current string. 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
# ';\nand 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 string "Tillie" inside it:

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

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

The `.previous_element` attribute is the exact opposite of `.next_element`. It points to the opening tag or string that was parsed immediately before this one:

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

.next_elements and .previous_elements

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

```
for element in last_a_tag.next_elements:
    print(repr(element))
# 'Tillie'
# ';\nand they lived at the bottom of a well.'
# '\n'
# <p class="story">...</p>
# '...'
# '\n'
```

Searching the tree

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

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

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

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

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

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

By passing in a filter to a method like `find_all()`, you can zoom in on the parts of the

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 tag-name match against that exact string. This code finds all the <b> tags in the document:

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

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

A regular expression

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

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

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

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

True

The value `True` matches every tag 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…bottom of a well.</p>,
#  <p class="story">...</p>]
```

This function picks up only 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".

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

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

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

A list

If you pass in a list, Beautiful Soup will look for a match against *any* string, regular expression, or function 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>]
```

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

find_all()

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

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

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

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

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

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

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

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

The name argument

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

This is the simplest usage:

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

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

The keyword arguments

Any keyword argument that's not recognized will be turned into a filter that matches tags by their attributes.

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

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

Just as with tags, you can filter an attribute based on a string, a regular expression, a list, a function, or the value True.

If you pass in a regular expression object for `href`, Beautiful Soup will pattern-match against each tag's 'href' attribute value:

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

The value `True` matches every tag that defines the attribute. This code finds *all* tags with an `id` attribute:

```
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>]
```

For more complex matches, you can define a function that takes an attribute value as its only argument. The function should return `True` if the value matches, and `False` otherwise.

Here's a function that finds all `a` tags whose `href` attribute *does not* match a regular expression:

```
import re
def not_lacie(href):
    return href and not re.compile("lacie").search(href)

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

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

If you pass in a list for an argument, Beautiful Soup will look for an attribute-value match against *any* string, regular expression, or function in that list. This code finds the first and last link:

```
soup.find_all(id=["link1", re.compile("3$")])
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

You can filter against multiple attributes at once by passing multiple keyword arguments:

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

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

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

Searching by CSS class

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

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

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

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

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

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

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

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

In older versions of Beautiful Soup, which don't have the `class_` shortcut, you can use the `attrs` argument 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>]
```

To search for tags that match two or more CSS classes at once, use the `Tag.select()`

CSS selector method described here:

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

The `string` argument

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

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

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

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

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

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

If you use the `string` argument in a tag search, Beautiful Soup will find all tags whose `.string` matches your value for `string`. This code finds the <a> tags whose `.string` is "Elsie":

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

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

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

The `limit` argument

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

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

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

The `recursive` argument

By default, `mytag.find_all()` will examine all the descendants of `mytag`: its children, its children's children, and so on. To consider only direct children, you can pass in `recursive=False`. See the difference here:

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

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

Here's that part of the document:

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

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

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

Calling a tag is like calling `find_all()`

For convenience, calling a `BeautifulSoup` object or `Tag` object as a function is equivalent to calling `find_all()` (if no built-in method has the name of the tag you're looking for). These two lines of code are equivalent:

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

These two lines are also equivalent:

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

find()

Method signature: find(name, attrs, recursive, string, **kwargs)

The `find_all()` method scans the entire document looking for results, but sometimes you only want to find one result. If you know a document has only 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()

Method signature: find_parents(name, attrs, string, limit, **kwargs)

Method signature: find_parent(name, attrs, string, **kwargs)

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

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

```
a_string = soup.find(string="Lacie")
a_string
# '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
#  <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 (*ancestor*) 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 noticed a similarity between `find_parent()` and `find_parents()`, and the .parent and .parents attributes mentioned earlier. These search methods actually use the `.parents` attribute to iterate through all parents (unfiltered), checking each one against the provided filter to see if it matches.

find_next_siblings() and find_next_sibling()

Method signature: find_next_siblings(name, attrs, string, limit, **kwargs)

Method signature: find_next_sibling(name, attrs, string, **kwargs)

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

Method signature: find_previous_siblings([name](), [attrs](), [string](), [limit](), [**kwargs]())

Method signature: find_previous_sibling([name](), [attrs](), [string](), [**kwargs]())

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

Method signature: find_all_next([name](), [attrs](), [string](), [limit](), [**kwargs]())

Method signature: find_next([name](), [attrs](), [string](), [**kwargs]())

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

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

first_link.find_all_next(string=True)
# ['Elsie', ',\n', 'Lacie', ' and\n', 'Tillie',
#  ';\nand they lived at the bottom of a well.', '\n', '...', '\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 matches the filter and it shows up later in the document in [document order]().

find_all_previous() and find_previous()

Method signature: find_all_previous([name](), [attrs](), [string](), [limit](), [**kwargs]())

Method signature: find_previous([name](), [attrs](), [string](), [**kwargs]())

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

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

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

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

The call to `find_all_previous("p")` found the first paragraph in the document (the one with class="title"), but it also finds the second paragraph, the <p> tag that contains the <a> tag we started with. This shouldn't be too surprising: we're looking at all the tags that show up earlier in the document in [document order]() 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 through the `.css` property

`BeautifulSoup` and `Tag` objects support CSS selectors through their `.css` property. The actual selector implementation is handled by the [Soup Sieve]() package, available on PyPI as `soupsieve`. If you installed Beautiful Soup through `pip`, Soup Sieve was installed at the same time, so you don't have to do anything extra.

The Soup Sieve documentation lists [all the currently supported CSS selectors](), but here are some of the basics. You can find tags by name:

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

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

Find tags by ID:

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

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

Find tags contained anywhere within other tags:

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

Find tags *directly* within other tags:

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

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

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

Find all matching next siblings of tags:

```
soup.css.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>]
```

Find the next sibling tag (but only if it matches):

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

Find tags by CSS class:

```
soup.css.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.css.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 that match any selector from a list of selectors:

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

Test for the existence of an attribute:

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

soup.css.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 ctlass="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

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

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

There's also a method called `select_one()`, which finds only the first tag that matches a selector:

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

As a convenience, you can call `select()` and `select_one()` can directly on the `BeautifulSoup` or `Tag` object, omitting the `.css` property:

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

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

CSS selector support is a convenience for people who already know the CSS selector syntax. You can do all of this with the Beautiful Soup API. If CSS selectors are all you need, you should skip Beautiful Soup altogether and parse the document with `lxml`: it's a lot faster. But Soup Sieve lets you *combine* CSS selectors with the Beautiful Soup API.

Advanced Soup Sieve features

Soup Sieve offers a substantial API beyond the `select()` and `select_one()` methods, and you can access most of that API through the `.css` attribute of `Tag` or `BeautifulSoup`. What follows is just a list of the supported methods; see [the Soup Sieve documentation](#) for full documentation.

The `iselect()` method works the same as `select()`, but it returns a generator instead of a list:

```
[tag['id'] for tag in soup.css.iselect(".sister")]
# ['link1', 'link2', 'link3']
```

The `closest()` method returns the nearest parent of a given `Tag` that matches a CSS selector, similar to Beautiful Soup's `find_parent()` method:

```
elsie = soup.css.select_one(".sister")
elsie.css.closest("p.story")
# <p class="story">Once upon a time there were three little sisters; and their
#  <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>
```

The `match()` method returns a Boolean depending on whether or not a specific `Tag` matches a selector:

```
# elsie.css.match("#link1")
True

# elsie.css.match("#link2")
False
```

The `filter()` method returns the subset of a tag's direct children that match a selector:

```
[tag.string for tag in soup.find('p', 'story').css.filter('a')]
# ['Elsie', 'Lacie', 'Tillie']
```

The `escape()` method escapes CSS identifiers that would otherwise be invalid:

```
soup.css.escape("1-strange-identifier")
# '\\31 -strange-identifier'
```

Namespaces in CSS selectors

If you've parsed XML that defines namespaces, you can use them in CSS selectors.:

```
from bs4 import BeautifulSoup
xml = """<tag xmlns:ns1="http://namespace1/" xmlns:ns2="http://namespace2/">
 <ns1:child>I'm in namespace 1</ns1:child>
 <ns2:child>I'm in namespace 2</ns2:child>
</tag> """
namespace_soup = BeautifulSoup(xml, "xml")

namespace_soup.css.select("child")
# [<ns1:child>I'm in namespace 1</ns1:child>, <ns2:child>I'm in namespace 2</ns

namespace_soup.css.select("ns1|child")
# [<ns1:child>I'm in namespace 1</ns1:child>]
```

Beautiful Soup tries to use namespace prefixes that make sense based on what it saw while parsing the document, but you can always provide your own dictionary of abbreviations:

```
namespaces = dict(first="http://namespace1/", second="http://namespace2/")
namespace_soup.css.select("second|child", namespaces=namespaces)
# [<ns1:child>I'm in namespace 2</ns1:child>]
```

History of CSS selector support

The `.css` property was added in Beautiful Soup 4.12.0. Prior to this, only the `.select()` and `.select_one()` convenience methods were supported.

The Soup Sieve integration was added in Beautiful Soup 4.7.0. Earlier versions had the `.select()` method, but only the most commonly-used CSS selectors were supported.

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 `Tag.attrs`, 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>', 'html.parser')
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 to a new string, the tag's contents are replaced with that string:

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

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>", 'html.parser')
new_string = soup.a.append("Bar")

soup
# <a>FooBar</a>
soup.a.contents
# ['Foo', 'Bar']
new_string
# 'Bar'
```

`Tag.append()` returns the newly appended element.

extend()

Starting in Beautiful Soup 4.7.0, `Tag` also supports a method called `.extend()`, which adds every element of a list to a `Tag`, in order:

```
soup = BeautifulSoup("<a>Soup</a>", 'html.parser')
soup.a.extend(["'s", " ", "on"])

soup
# <a>Soup's on</a>
soup.a.contents
# ['Soup', "'s", ' ', 'on']
```

`Tag.extend()` returns the list of appended elements.

NavigableString() and .new_tag()

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

```
from bs4 import NavigableString
soup = BeautifulSoup("<b></b>", 'html.parser')
tag = soup.b
tag.append("Hello")
new_string = NavigableString(" there")
tag.append(new_string)
tag
# <b>Hello there.</b>
tag.contents
# ['Hello', ' there']
```

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

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

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

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

```
soup = BeautifulSoup("<b></b>", 'html.parser')
original_tag = soup.b

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

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

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

(The ``string`` argument to ``new_tag`` was introduced in Beautiful Soup 4.13.0.)

Because insertion methods return the newly inserted element, you can create, insert, and obtain an element in one step:

```
soup = BeautifulSoup("<html></html>", 'html.parser')
html = soup.html

title = html.append(soup.new_tag("title"))
```

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 will be inserted at whatever numeric position you say, similar to `.insert()` on a Python list:

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

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

You can pass more than one element into `Tag.insert()`. All the elements will be inserted, starting at the numeric position you provide.

`Tag.insert()` returns the list of newly inserted elements.

insert_before() and insert_after()

The `insert_before()` method inserts tags or strings immediately before something else in the parse tree:

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

The `insert_after()` method inserts tags or strings immediately after something else in the parse tree:

```
div = soup.new_tag('div')
div.string = 'ever'
soup.b.i.insert_after(" you ", div)
soup.b
# <b><i>Don't</i> you <div>ever</div> leave</b>
soup.b.contents
# [<i>Don't</i>, ' you', <div>ever</div>, 'leave']
```

Both methods return the list of newly inserted elements.

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, 'html.parser')
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, 'html.parser')
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
# 'example.com'

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

decompose()

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

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

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

The behavior of a decomposed `Tag` or `NavigableString` is not defined and you should not use it for anything. If you're not sure whether something has been decomposed, you can check its `.decomposed` property *(new in Beautiful Soup 4.9.0)*:

```
i_tag.decomposed
# True

a_tag.decomposed
# False
```

replace_with()

`PageElement.replace_with()` extracts a tag or string from the tree, then replaces it with one or more tags or strings of your choice:

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

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

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

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

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

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

The ability to pass multiple arguments into replace_with() is new in Beautiful Soup 4.10.0.

wrap()

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

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

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

This method is new in Beautiful Soup 4.0.5.

unwrap()

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

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

smooth()

After calling a bunch of methods that modify the parse tree, you may end up with two or more `NavigableString` objects next to each other. Beautiful Soup doesn't have any problems with this, but since it can't happen in a freshly parsed document, you might not expect behavior like the following:

```
soup = BeautifulSoup("<p>A one</p>", 'html.parser')
soup.p.append(", a two")

soup.p.contents
# ['A one', ', a two']

print(soup.p.encode())
# b'<p>A one, a two</p>'

print(soup.p.prettify())
# <p>
#  A one
#  , a two
# </p>
```

You can call `Tag.smooth()` to clean up the parse tree by consolidating adjacent strings:

```
soup.smooth()

soup.p.contents
# ['A one, a two']

print(soup.p.prettify())
# <p>
#  A one, a two
# </p>
```

This method is new in Beautiful Soup 4.8.0.

Output

Pretty-printing

The `prettify()` method will turn a Beautiful Soup parse tree into a nicely formatted Unicode string, with a separate line for each tag and each string:

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

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

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

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

Since it adds whitespace (in the form of newlines), `prettify()` changes the meaning of an HTML document and should not be used to reformat one. The goal of `prettify()` is to help you visually understand the structure of the documents you work with.

Non-pretty printing

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

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

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

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

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

Output formatters

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

```
soup = BeautifulSoup("&ldquo;Dammit!&rdquo; he said.", 'html.parser')
str(soup)
# '"Dammit!" he said.'
```

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

```
soup.encode("utf8")
# b'\xe2\x80\x9cDammit!\xe2\x80\x9d he said.'
```

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

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

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

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

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

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

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

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

If you pass in `formatter="html5"`, it's similar to `formatter="html"`, but Beautiful Soup will omit the closing slash in HTML void tags like "br":

```python
br = BeautifulSoup("<br>", 'html.parser').br

print(br.encode(formatter="html"))
# b'<br/>'

print(br.encode(formatter="html5"))
# b'<br>'
```

In addition, any attributes whose values are the empty string will become HTML-style Boolean attributes:

```python
option = BeautifulSoup('<option selected=""></option>').option
print(option.encode(formatter="html"))
# b'<option selected=""></option>'

print(option.encode(formatter="html5"))
# b'<option selected></option>'
```

(This behavior is new as of Beautiful Soup 4.10.0.)

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:

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

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

Formatter objects

If you need more sophisticated control over your output, you can instantiate one of Beautiful Soup's formatter classes and pass that object in as `formatter`.

class **HTMLFormatter**

Used to customize the formatting rules for HTML documents.

Here's a formatter that converts strings to uppercase, whether they occur in a string object or an attribute value:

```python
from bs4.formatter import HTMLFormatter
def uppercase(str):
    return str.upper()

formatter = HTMLFormatter(uppercase)

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

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

Here's a formatter that increases the indentation width when pretty-printing:

```python
formatter = HTMLFormatter(indent=8)
```

```
    print(link_soup.a.prettify(formatter=formatter))
    # <a href="http://example.com/?foo=val1&bar=val2">
    #     A link
    # </a>
```

class **XMLFormatter**

Used to customize the formatting rules for XML documents.

Writing your own formatter

Subclassing **HTMLFormatter** or **XMLFormatter** will give you even more control over the output. For example, Beautiful Soup sorts the attributes in every tag by default:

```
    attr_soup = BeautifulSoup(b'<p z="1" m="2" a="3"></p>', 'html.parser')
    print(attr_soup.p.encode())
    # <p a="3" m="2" z="1"></p>
```

To turn this off, you can subclass the `Formatter.attributes()` method, which controls which attributes are output and in what order. This implementation also filters out the attribute called "m" whenever it appears:

```
    class UnsortedAttributes(HTMLFormatter):
        def attributes(self, tag):
            for k, v in tag.attrs.items():
                if k == 'm':
                    continue
                yield k, v

    print(attr_soup.p.encode(formatter=UnsortedAttributes()))
    # <p z="1" a="3"></p>
```

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 your entity substitution function, just in case you've written a custom function 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>", 'html.parser')
    soup.a.string = CData("one < three")
    print(soup.a.prettify(formatter="html"))
    # <a>
    #  <![CDATA[one < three]]>
    # </a>
```

get_text()

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

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

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

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

```
    # soup.get_text("|")
    '\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)
    '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]
    # ['I linked to', 'example.com']
```

As of Beautiful Soup version 4.9.0, when lxml or html.parser are in use, the contents of <script>, <style>, and <template> tags are generally not considered to be 'text', since those tags are not part of the human-visible content of the page.

As of Beautiful Soup version 4.10.0, you can call get_text(), .strings, or .stripped_strings on a NavigableString object. It will either return the object itself, or nothing, so the only reason to do this is when you're iterating over a mixed list.

As of Beautiful Soup version 4.13.0, you can call .string on a NavigableString object. It will return the object itself, so again, the only reason to do this is when you're iterating over a mixed list.

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 source of 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 values 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 ask for a parser that isn't installed, Beautiful Soup will raise an exception so that you don't inadvertently parse a document under an unknown set of rules. For example, 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 using the parser that comes with Python:

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

Since a standalone tag is not valid HTML, html.parser turns it into a tag pair.

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

```
print(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 <a> tag gets wrapped in <body> and <html> tags, and the dangling </p> tag is simply ignored:

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

Here's the same document parsed using html5lib:

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

Instead of ignoring the dangling </p> tag, html5lib pairs it with an opening <p> tag. html5lib also adds an empty <head> tag; lxml didn't bother.

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

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

Like lxml, this parser ignores the closing </p> tag. Unlike html5lib or lxml, this parser makes no attempt to create a well-formed HTML document by adding <html> or <body> tags.

Since the document "<a></p>" 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 = b"<h1>Sacr\xc3\xa9 bleu!</h1>"
soup = BeautifulSoup(markup, 'html.parser')
soup.h1
# <h1>Sacré bleu!</h1>
soup.h1.string
# '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'
```

If `.original_encoding` is `None`, that means the document was already Unicode when it was passed into Beautiful Soup:

```
markup = "<h1>Sacré bleu!</h1>"
soup = BeautifulSoup(markup, 'html.parser')
print(soup.original_encoding)
# None
```

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 lock on it, and misidentifies it as ISO-8859-7:

```
markup = b"<h1>\xed\xe5\xec\xf9</h1>"
soup = BeautifulSoup(markup, 'html.parser')
print(soup.h1)
# <h1>νεμω</h1>
print(soup.original_encoding)
# iso-8859-7
```

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

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

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

```
soup = BeautifulSoup(markup, 'html.parser', exclude_encodings=["iso-8859-7"])
print(soup.h1)
# <h1>םולש</h1>
print(soup.original_encoding)
# WINDOWS-1255
```

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

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

Output encoding

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

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

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

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

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

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

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

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

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

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

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

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

```
print(tag.encode("utf-8"))
# b'<b>\xe2\x98\x83</b>'

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

print(tag.encode("ascii"))
# b'<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(b"\xc2\xabSacr\xc3\xa9 bleu!\xc2\xbb")
print(dammit.unicode_markup)
# «Sacré bleu!»
dammit.original_encoding
# 'utf-8'
```

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

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

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

Smart quotes

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

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

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

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

You can also convert Microsoft smart quotes to ASCII quotes:

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

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

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

Inconsistent encodings

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

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

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

```
print(doc)
# ☃☃☃I like snowmen!
print(doc.decode("windows-1252"))
# â˜ƒâ˜ƒâ˜ƒ"I like snowmen!"
```

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

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

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

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

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

Line numbers

The `html.parser` and `html5lib` parsers can keep track of where in the original document each `Tag` was found. You can access this information as `Tag.sourceline` (line number) and `Tag.sourcepos` (position of the start tag within a line):

```python
markup = "<p\n>Paragraph 1</p>\n    <p>Paragraph 2</p>"
soup = BeautifulSoup(markup, 'html.parser')
for tag in soup.find_all('p'):
    print(repr((tag.sourceline, tag.sourcepos, tag.string)))
# (1, 0, 'Paragraph 1')
# (3, 4, 'Paragraph 2')
```

Note that the two parsers mean slightly different things by `sourceline` and `sourcepos`. For html.parser, these numbers represent the position of the initial less-than sign. For html5lib, these numbers represent the position of the final greater-than sign:

```python
soup = BeautifulSoup(markup, 'html5lib')
for tag in soup.find_all('p'):
    print(repr((tag.sourceline, tag.sourcepos, tag.string)))
# (2, 0, 'Paragraph 1')
# (3, 6, 'Paragraph 2')
```

You can shut off this feature by passing `store_line_numbers=False` into the `BeautifulSoup` constructor:

```python
markup = "<p\n>Paragraph 1</p>\n    <p>Paragraph 2</p>"
soup = BeautifulSoup(markup, 'html.parser', store_line_numbers=False)
print(soup.p.sourceline)
# None
```

This feature is new in 4.8.1, and the parsers based on lxml don't support it.

Comparing objects for equality

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

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

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

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

```python
print(first_b is second_b)
# False
```

Copying Beautiful Soup objects

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

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

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

```python
print(soup.p == p_copy)
# True

print(soup.p is p_copy)
# False
```

The only real difference is that the copy is completely detached from the original Beautiful Soup object tree, just as if `extract()` had been called on it. This is because two different `Tag` objects can't occupy the same space at the same time.

```python
print(p_copy.parent)
```

```
     # None
```

You can use `Tag.copy_self()` to create a copy of a `Tag` without copying its contents.

```
original = BeautifulSoup('<a id="a_tag" class="link">the <i>link</i></a>',
print(original.a)
# <a class="link" id="a_tag">the <i>link</a>
print(original.a.copy_self())
# <a class="link" id="a_tag"></a>
```

(Tag.copy_self() is introduced in Beautiful Soup 4.13.0.)

Low-level search interface

Almost everyone who uses Beautiful Soup to extract information from a document can get what they need using the methods described in <u>Searching the tree</u>. However, there's a lower-level interface that lets you define any matching behavior you want. Behind the scenes, the parts of the Beautiful Soup API that most people use--`find_all()` and the like—are actually using this low-level interface, and you can use it directly.

(Access to the low-level search interface is a new feature in Beautiful Soup 4.13.0.)

Custom element filtering

class `ElementFilter`

The `ElementFilter` class is your entry point to the low-level interface. To use it, define a function that takes a `PageElement` object (which could be either a `Tag` or a `NavigableString`). The function must return `True` if the element matches your custom criteria, and `False` if it doesn't.

This example function looks for content-containing tags and strings, but skips whitespace-only strings:

```
from bs4 import Tag, NavigableString
def non_whitespace_element_func(tag_or_string):
    """
    return True for:
    * all Tag objects
    * NavigableString objects that contain non-whitespace text
    """
    return (
        isinstance(tag_or_string, Tag) or
        (isinstance(tag_or_string, NavigableString) and
            tag_or_string.strip() != ""))
```

Once you have a function, pass it into the `ElementFilter` constructor:

```
from bs4.filter import ElementFilter
non_whitespace_filter = ElementFilter(non_whitespace_element_func)
```

You can then use this `ElementFilter` object as the first argument to any of the <u>Searching the tree</u> methods. Whatever criteria you defined in your function will be used instead of the default Beautiful Soup match logic:

```
from bs4 import BeautifulSoup
small_doc = """
<p>
  <b>bold</b>
  <i>italic</i>
  and
  <u>underline</u>
</p>
"""

soup = BeautifulSoup(small_doc, 'html.parser')

soup.find('p').find_all(non_whitespace_filter, recursive=False)
# [<b>bold</b>, <i>italic</i>, '\n  and\n  ', <u>underline</u>]

soup.find("b").find_next(non_whitespace_filter)
# 'bold'

soup.find("i").find_next_siblings(non_whitespace_filter)
# ['\n  and\n  ', <u>underline</u>]
```

Every potential match will be run through your function, and the only `PageElement` objects returned will be the ones where your function returned `True`.

To summarize the function-based matching behaviors,

- A function passed as the first argument to a search method (or equivalently, using the `name` argument) considers only `Tag` objects.
- A function passed to a search method using the `string` argument considers only `NavigableString` objects.
- A function passed to a search method using an `ElementFilter` object considers both `Tag` and `NavigableString` objects.

Custom element iteration

`ElementFilter.filter()`

By passing an `ElementFilter` instance into Beautiful Soup's tree-searching methods, you can completely customize what it means for Beautiful Soup to match an element as it iterates over the parse tree. By using the `ElementFilter.filter()` method, you can also completely customize what it means for Beautiful Soup to iterate over the parse tree in the first place.

The `ElementFilter.filter()` method takes a generator that yields a stream of `PageElement` objects. There is no restriction on which `PageElement` objects show up, how many times they show up, or in which order. Theoretically, they don't even need to be

from the same `BeautifulSoup` document. You can do whatever makes sense for you.

Here's a silly example: a generator that walks randomly back and forth through the parse tree:

```python
import random
def random_walk(starting_location):
    location = starting_location
    while location is not None:
        yield location
        if random.random() < 0.5:
            location = location.next_element
        else:
            location = location.previous_element
        if location is None:
            return
```

Pass this generator into the example `ElementFilter.filter()` and Beautiful Soup will wander randomly around the parse tree, applying the `non_whitespace_filter` function to every element it finds, and yielding all of the matches—potentially yielding a given object more than once:

```python
[x for x in non_whitespace_filter.filter(random_walk(soup.b))]
# [<b>bold</b>, 'bold', <b>bold</b>, <p><b>bold</b>...]

[x for x in non_whitespace_filter.filter(random_walk(soup.b))]
# [<b>bold</b>, <b>bold</b>, 'bold', <i>italic</i>, <i>italic</i>, ...]
```

(Note that unlike the other code examples in this documentation, this example can give different results every time you run it, thanks to the random element. It's very unlikely, but this function could wander around the parse tree forever and *never* complete.)

Advanced parser customization

Beautiful Soup offers a number of ways to customize how the parser treats incoming HTML and XML. This section covers the most commonly used customization techniques.

Parsing only part of a document

Let's say you want to use Beautiful Soup to 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.)

class **SoupStrainer**

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

```python
from bs4 import SoupStrainer

only_a_tags = SoupStrainer("a")

only_tags_with_id_link2 = SoupStrainer(id="link2")

def is_short_string(string):
    return string is not None and len(string) < 10

only_short_strings = SoupStrainer(string=is_short_string)
```

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

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

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

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

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

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

print(BeautifulSoup(html_doc, "html.parser", parse_only=only_short_strings).pre
# Elsie
```

```
#    ,
# Lacie
# and
# Tillie
# ...
#
```

The **SoupStrainer** behavior is as follows:

- When a tag matches, it is kept (including all its contents, whether they also match or not).
- When a tag does not match, the tag itself is not kept, but parsing continues into its contents to look for other tags that do match.

Customizing multi-valued attributes

In an HTML document, an attribute like `class` is given a list of values, and an attribute like `id` is given a single value, because the HTML specification treats those attributes differently:

```
markup = '<a class="cls1 cls2" id="id1 id2">'
soup = BeautifulSoup(markup, 'html.parser')
soup.a['class']
# ['cls1', 'cls2']
soup.a['id']
# 'id1 id2'
```

You can turn this off by passing in `multi_valued_attributes=None`. Than all attributes will be given a single value:

```
soup = BeautifulSoup(markup, 'html.parser', multi_valued_attributes=None)
soup.a['class']
# 'cls1 cls2'
soup.a['id']
# 'id1 id2'
```

You can customize this behavior quite a bit by passing in a dictionary for `multi_valued_attributes`. If you need this, look at `HTMLTreeBuilder.DEFAULT_CDATA_LIST_ATTRIBUTES` to see the configuration Beautiful Soup uses by default, which is based on the HTML specification.

(This is a new feature in Beautiful Soup 4.8.0.)

Handling duplicate attributes

When using the `html.parser` parser, you can use the `on_duplicate_attribute` constructor argument to customize what Beautiful Soup does when it encounters a tag that defines the same attribute more than once:

```
markup = '<a href="http://url1/" href="http://url2/">'
```

The default behavior is to use the last value found for the tag:

```
soup = BeautifulSoup(markup, 'html.parser')
soup.a['href']
# http://url2/

soup = BeautifulSoup(markup, 'html.parser', on_duplicate_attribute='replace')
soup.a['href']
# http://url2/
```

With `on_duplicate_attribute='ignore'` you can tell Beautiful Soup to use the *first* value found and ignore the rest:

```
soup = BeautifulSoup(markup, 'html.parser', on_duplicate_attribute='ignore')
soup.a['href']
# http://url1/
```

(lxml and html5lib always do it this way; their behavior can't be configured from within Beautiful Soup.)

If you need more control, you can pass in a function that's called on each duplicate value:

```
def accumulate(attributes_so_far, key, value):
    if not isinstance(attributes_so_far[key], list):
        attributes_so_far[key] = [attributes_so_far[key]]
    attributes_so_far[key].append(value)

soup = BeautifulSoup(markup, 'html.parser', on_duplicate_attribute=accumulate)
soup.a['href']
# ["http://url1/", "http://url2/"]
```

(This is a new feature in Beautiful Soup 4.9.1.)

Instantiating custom subclasses

When a parser tells Beautiful Soup about a tag or a string, Beautiful Soup will instantiate a **Tag** or **NavigableString** object to contain that information. Instead of that default behavior, you can tell Beautiful Soup to instantiate *subclasses* of **Tag** or **NavigableString**, subclasses you define with custom behavior:

```
from bs4 import Tag, NavigableString
class MyTag(Tag):
    pass
```

```
class MyString(NavigableString):
    pass


markup = "<div>some text</div>"
soup = BeautifulSoup(markup, 'html.parser')
isinstance(soup.div, MyTag)
# False
isinstance(soup.div.string, MyString)
# False

my_classes = { Tag: MyTag, NavigableString: MyString }
soup = BeautifulSoup(markup, 'html.parser', element_classes=my_classes)
isinstance(soup.div, MyTag)
# True
isinstance(soup.div.string, MyString)
# True
```

This can be useful when incorporating Beautiful Soup into a test framework.

(This is a new feature in Beautiful Soup 4.8.1.)

Troubleshooting

diagnose()

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

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

diagnose(data)

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

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

Errors when parsing a document

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

These problems are almost never problems with Beautiful Soup itself. This is not because Beautiful Soup is an amazingly well-written piece of software. It's because Beautiful Soup doesn't include any parsing code. Instead, it relies on external parsers. If one parser isn't working on a certain document, the best solution is to try a different parser. See Installing a parser for details and a parser comparison. If this doesn't help, you might need to inspect the document tree found inside the `BeautifulSoup` object, to see where the markup you're looking for actually ended up.

Version mismatch problems

- `SyntaxError: Invalid syntax` (on the line `ROOT_TAG_NAME = '[document]'`): Caused by running an old Python 2 version of Beautiful Soup under Python 3, without converting the code.
- `ImportError: No module named HTMLParser` - Caused by running an old 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 in an environment 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 in an environment that doesn't have BS4 installed.

Parsing XML

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

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

You'll need to have lxml installed.

Other parser problems

- If your script works on one computer but not another, or in one virtual environment but not another, or outside the virtual environment but not inside, it's probably because the two environments have different parser libraries available. For example, you may have developed the script on a computer that has lxml installed, and then tried to run it on a computer that only has html5lib installed. See

[Differences between parsers](#) for why this matters, and fix the problem by mentioning a specific parser library in the `BeautifulSoup` constructor.

- Because [HTML tags and attributes are case-insensitive](#), all three HTML parsers convert tag and attribute names to lowercase. That is, the markup <TAG></TAG> is converted to <tag></tag>. If you want to preserve mixed-case or uppercase tags and attributes, you'll need to [parse the document as XML.](#)

Miscellaneous

- `UnicodeEncodeError: 'charmap' codec can't encode character '\xfoo' in position bar` (or just about any other `UnicodeEncodeError`) - This problem shows up in two main situations. First, when you try to print a Unicode character that your console doesn't know how to display. (See [this page on the Python wiki](#) for help.) Second, when you're writing to a file and you pass in a Unicode character that's not supported by your default encoding. In this case, the simplest solution is to explicitly encode the Unicode string into UTF-8 with `u.encode("utf8")`.
- `KeyError: [attr]` - Caused by accessing `tag['attr']` when the tag in question doesn't define the `attr` attribute. The most common errors are `KeyError: 'href'` and `KeyError: 'class'`. Use `tag.get('attr')` if you're not sure `attr` is defined, just as you would with a Python dictionary.
- `AttributeError: 'ResultSet' object has no attribute 'foo'` - This usually happens because you expected `find_all()` to return a single tag or string. But `find_all()` returns a *list* of tags and strings—a `ResultSet` object. You need to iterate over the list and look at the `.foo` of each one. Or, if you really only want one result, you need to use `find()` instead of `find_all()`.
- `AttributeError: 'NoneType' object has no attribute 'foo'` - This usually happens because you called `find()` and then tried to access the `.foo` attribute of the result. But in your case, `find()` didn't find anything, so it returned `None`, instead of returning a tag or a string. You need to figure out why your `find()` call isn't returning anything.
- `AttributeError: 'NavigableString' object has no attribute 'foo'` - This usually happens because you're treating a string as though it were a tag. You may be iterating over a list, expecting that it contains nothing but tags, when it actually contains both tags and strings.

Improving Performance

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

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

You can speed up encoding detection significantly by installing the [cchardet](#) library.

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

Translating this documentation

New translations of the Beautiful Soup documentation are greatly appreciated. Translations should be licensed under the MIT license, just like Beautiful Soup and its English documentation are.

There are two ways of getting your translation into the main code base and onto the Beautiful Soup website:

1. Create a branch of the Beautiful Soup repository, add your translation, and propose a merge with the main branch, the same as you would do with a proposed change to the source code.
2. Send a message to the Beautiful Soup discussion group with a link to your translation, or attach your translation to the message.

Use the Chinese or Brazilian Portuguese translations as your model. In particular, please translate the source file `doc/index.rst`, rather than the HTML version of the documentation. This makes it possible to publish the documentation in a variety of formats, not just HTML.

Beautiful Soup 3

Beautiful Soup 3 is the previous release series, and is no longer supported. Development of Beautiful Soup 3 stopped in 2012, and the package was completely discontinued in 2021. There's no reason to install it unless you're trying to get very old software to work, but it's published through PyPi as `BeautifulSoup`:

```
$ pip install BeautifulSoup
```

You can also download [a tarball of the final release, 3.2.2.](#)

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

[The documentation for Beautiful Soup 3 is archived online.](#)

Porting code to BS4

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

```
from BeautifulSoup import BeautifulSoup
```

becomes this:

```
from bs4 import BeautifulSoup
```

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

Although BS4 is mostly backward-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 backward compatibility.

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

You need a parser

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

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

Property names

I renamed three attributes to avoid using words that have special meaning to Python. Unlike my changes to method names (which you'll see in the form of deprecation warnings), these changes *did not preserve backwards compatibility*. If you used these attributes in BS3, your code will break in BS4 until you change them.

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

Generators

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

XML

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

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

Entities

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

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

Miscellaneous

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

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

`Tag` objects now implement the `__hash__` method, such that two `Tag` objects are considered equal if they generate the same markup. This may change your script's behavior if you put `Tag` objects into a dictionary or set.

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

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

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

ambiguous markup.

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