Chapter 22. Structured Text: HTML

Most documents on the web use HTML, the HyperText Markup Language. *Markup* is the insertion of special tokens, known as *tags*, in a text document, to structure the text. HTML is, in theory, an application of the large, general standard known as SGML, the **Standard Generalized Markup Language**. In practice, many documents on the web use HTML in sloppy or incorrect ways.

HTML was designed for presenting documents in a browser. As web content evolved, users realized it lacked the capability for *semantic markup*, in which the markup indicates the meaning of the delineated text rather than simply its appearance. Complete, precise extraction of the information in an HTML document often turns out to be unfeasible. A more rigorous standard called XHTML attempted to remedy these shortcomings. XHTML is similar to traditional HTML, but it is defined in terms of XML, the eXtensible Markup Language, and more precisely than HTML. You can handle well-formed XHTML with the tools covered in **Chapter 23**. However, as of this writing, XHTML has not enjoyed overwhelming success, getting scooped instead by the more pragmatic HTML5.

Despite the difficulties, it's often possible to extract at least some useful information from HTML documents (a task known as *web scraping*, *spidering*, or just *scraping*). Python's standard library tries to help, supplying the `html` package for the task of parsing HTML documents, whether for the purpose of presenting the documents or, more typically, as part of an attempt to extract information from them. However, when you're dealing with somewhat-broken web pages (which is almost always the case!), the third-party module **`BeautifulSoup`** usually offers your last, best hope. In this book, for pragmatic reasons, we mostly cover `BeautifulSoup`, ignoring the standard library modules competing with it. The reader looking

for alternatives should also investigate the increasingly popular **`scrapy` package**.

Generating HTML and embedding Python in HTML are also reasonably frequent tasks. The standard Python library doesn't support HTML generation or embedding, but you can use Python string formatting, and third-party modules can also help. `BeautifulSoup` lets you alter an HTML tree (so, in particular, you can build one up programmatically, even "from scratch"); an often preferable alternative approach is *templating*, supported, for example, by the third-party module **`jinja2`**, whose bare essentials we cover in **"The jinja2 Package"**.

The html.entities Module

The `html.entities` module in Python's standard library supplies a few attributes, all of them mappings (see **Table 22-1**). They come in handy whatever general approach you're using to parse, edit, or generate HTML, including the `BeautifulSoup` package covered in the following section.

Table 22-1. Attributes of `html.entities`

`codepoint2name`	A mapping from Unicode codepoints to HTML entity names. For example, `entities.codepoint2name[228]` is `'auml'`, since Unicode character 228, ä, "lowercase a with diaeresis," is encoded in HTML as `'&auml;'`.
`entitydefs`	A mapping from HTML entity names to Unicode equivalent single-character strings. For example, `entities.entitydefs['auml']` is `'ä'`, and `entities.entitydefs['sigma']` is `'σ'`.
`html5`	A mapping from HTML5 named character references to equivalent single-character strings. For example, `entities.html5['gt;']` is `'>'`. The trailing semicolon in the key *does* matter—a few, but far from all, HTML5 named character

	references can optionally be spelled without a trailing semicolon, and in those cases both keys (with and without the trailing semicolon) are present in `entities.html5`.
`name2codepoint`	A mapping from HTML entity names to Unicode codepoints. For example, `entities.name2codepoint['auml']` is `228`.

The BeautifulSoup Third-Party Package

`BeautifulSoup` lets you parse HTML even if it's rather badly formed. It uses simple heuristics to compensate for typical HTML brokenness, and succeeds at this hard task surprisingly well in most cases. The current major version of `BeautifulSoup` is version 4, also known as `bs4`. In this book, we specifically cover version 4.10; as of this writing, that's the latest stable version of `bs4`.

INSTALLING VERSUS IMPORTING BEAUTIFULSOUP

`BeautifulSoup` is one of those annoying modules whose packaging requires you to use different names inside and outside Python. You install the module by running **`pip install beautifulsoup4`** at a shell command prompt, but when you import it in your Python code, you use **`import`** `bs4`.

The BeautifulSoup Class

The `bs4` module supplies the `BeautifulSoup` class, which you instantiate by calling it with one or two arguments: first, `htmltext`—either a file-like object (which is read to get the HTML text to parse) or a string (which is the text to parse)—and second, an optional `parser` argument.

Which parser BeautifulSoup uses

If you don't pass a `parser` argument, `BeautifulSoup` "sniffs around" to pick the best parser (but you may get a `GuessedAtParserWarning` warning in this case). If you haven't installed any other parser, `BeautifulSoup` defaults to `html.parser` from the Python standard library; if you have other parsers installed, `BeautifulSoup` defaults to one of them (`lxml` is currently the preferred one). Unless specified otherwise, the following examples use the default Python `html.parser`. To get more control and to avoid the differences between parsers mentioned in the `BeautifulSoup` **documentation**, pass the name of the parser library to use as the second argument as you instantiate `BeautifulSoup`.[1]

For example, if you have installed the third-party package `html5lib` (to parse HTML in the same way as all major browsers do, albeit slowly), you may call:

```
soup = bs4.BeautifulSoup(thedoc, 'html5lib')
```

When you pass `'xml'` as the second argument, you must already have the third-party package `lxml` installed. `BeautifulSoup` then parses the document as XML, rather than as HTML. In this case, the attribute `is_xml` of `soup` is **`True`**; otherwise, `soup.is_xml` is **`False`**. You can also use `lxml` to parse HTML, if you pass `'lxml'` as the second argument. More generally, you may need to install the appropriate parser library depending on the second argument you choose to pass to a call to `bs4.BeautifulSoup`; `BeautifulSoup` reminds you with a warning message if you don't.

Here's an example of using different parsers on the same string:

```
>>> import bs4, lxml, html5lib
>>> sh = bs4.BeautifulSoup('<p>hello', 'html.parser')
>>> sx = bs4.BeautifulSoup('<p>hello', 'xml')
>>> sl = bs4.BeautifulSoup('<p>hello', 'lxml')
>>> s5 = bs4.BeautifulSoup('<p>hello', 'html5lib')
```

```
>>> for s in [sh, sx, sl, s5]:
...   print(s, s.is_xml)
...
```

```
<p>hello</p> False
<?xml version="1.0" encoding="utf-8"?>
<p>hello</p> True
<html><body><p>hello</p></body></html> False
<html><head></head><body><p>hello</p></body></html> False
```

DIFFERENCES BETWEEN PARSERS IN FIXING INVALID HTML INPUT

In the preceding example, `'html.parser'` simply inserts the end tag `</p>`, missing from the input. Other parsers vary in the degree to which they repair invalid HTML input by adding required tags, such as `<html>`, `<head>`, and `<body>`, as you can see in the example.

BeautifulSoup, Unicode, and encoding

`BeautifulSoup` uses Unicode, deducing or guessing the encoding[2] when the input is a bytestring or binary file. For output, the `prettify` method returns a `str` representation of the tree, including tags and their attributes. `prettify` formats the string with whitespace and newlines added to indent elements, displaying the nesting structure. To have it instead return a `bytes` object (a bytestring) in a given encoding, pass it the encoding name as an argument. If you don't want the result to be "prettified," use the `encode` method to get a bytestring, and the `decode` method to get a Unicode string. For example:

```
>>> s = bs4.BeautifulSoup('<p>hello', 'html.parser')
>>> print(s.prettify())
```

```
<p>
```

```
 hello
</p>
```

```
>>> print(s.decode())
```

```
<p>hello</p>
```

```
>>> print(s.encode())
```

```
b'<p>hello</p>'
```

The Navigable Classes of bs4

An instance *b* of class `BeautifulSoup` supplies attributes and methods to "navigate" the parsed HTML tree, returning instances of *navigable classes* `Tag` and `NavigableString`, along with subclasses of `NavigableString` (`CData`, `Comment`, `Declaration`, `Doctype`, and `ProcessingInstruction`, differing only in how they are emitted when you output them).

Each instance of a navigable class lets you keep navigating—i.e., dig for more information—with pretty much the same set of navigational attributes and search methods as *b* itself. There are differences: instances of `Tag` can have HTML attributes and child nodes in the HTML tree, while instances of `NavigableString` cannot (instances of `NavigableString` always have one text string, a parent `Tag`, and zero or more siblings, i.e., other children of the same parent tag).

NAVIGABLE CLASS TERMINOLOGY

When we say "instances of `NavigableString`," we include instances of any of its subclasses; when we say "instances of `Tag`," we include instances of `BeautifulSoup` since the latter is a subclass of `Tag`. Instances of navigable classes are also known as the *elements* or *nodes* of the tree.

All instances of navigable classes have attribute `name`: it's the tag string for `Tag` instances, `'[document]'` for `BeautifulSoup` instances, and **`None`** for instances of `NavigableString`.

Instances of `Tag` let you access their HTML attributes by indexing, or you can get them all as a `dict` via the `.attrs` Python attribute of the instance.

Indexing instances of Tag

When *`t`* is an instance of `Tag`, *`t`*`['foo']` looks for an HTML attribute named `foo` within *`t`*'s HTML attributes and returns the string for the `foo` attribute. When *`t`* has no HTML attribute named `foo`, *`t`*`['foo']` raises a `KeyError` exception; just like on a `dict`, call *`t`*`.get('foo', default=`**`None`**`)` to get the value of the `default` argument instead of an exception.

A few attributes, such as `class`, are defined in the HTML standard as being able to have multiple values (e.g., `<body class="foo bar">...</body>`). In these cases, the indexing returns a `list` of values—for example, `soup.body['class']` would be `['foo', 'bar']` (again, you get a `KeyError` exception when the attribute isn't present at all; use the `get` method, instead of indexing, to get a default value instead).

To get a `dict` that maps attribute names to values (or, in a few cases defined in the HTML standard, `lists` of values), use the attribute *`t`*`.attrs`:

```
>>> s = bs4.BeautifulSoup('<p foo="bar" class="ic">baz')
>>> s.get('foo')
>>> s.p.get('foo')
```

```
'bar'
```

```
>>> s.p.attrs
```

```
{'foo': 'bar', 'class': ['ic']}
```

HOW TO CHECK IF A TAG INSTANCE HAS A CERTAIN ATTRIBUTE

To check if a `Tag` instance *`t`*'s HTML attributes include one named `'foo'`, *don't* use `if 'foo' in t`:—the `in` operator on `Tag` instances looks among the `Tag`'s *children*, *not* its *attributes*. Rather, use `if 'foo' in t.attrs:` or, better, `if t.has_attr('foo'):`.

Getting an actual string

When you have an instance of `NavigableString`, you often want to access the actual text string it contains. When you have an instance of `Tag`, you may want to access the unique string it contains, or, should it contain more than one, all of them—perhaps with their text stripped of any whitespace surrounding it. Here's how you can best accomplish these tasks.

When you have a `NavigableString` instance *`s`* and you need to stash or process its text somewhere, without further navigation on it, call `str(s)`. Or, use `s.encode(codec='utf8')` to get a bytestring, or `s.decode()` to get a text string (i.e., Unicode). These give you the actual string, without references to the `BeautifulSoup` tree that would impede garbage collection (*`s`* supports all methods of Unicode strings, so call those directly if they do all that you need).

Given an instance *`t`* of `Tag` containing a single `NavigableString` instance *`s`*, you can use *`t`*`.string` to fetch *`s`* (or, to just get the text you want from *`s`*,

use `t.string.decode()`). `t.string` only works when `t` has a single child that's a `NavigableString`, or a single child that's a `Tag` whose only child is a `NavigableString`; otherwise, `t.string` is **`None`**.

As an iterator on *all* contained (navigable) strings, use `t.strings`. You can use `''.join(t.strings)` to get all the strings concatenated into one, in a single step. To ignore whitespace around each contained string, use the iterator `t.stripped_strings` (which also skips all-whitespace strings).

Alternatively, call `t.get_text()`: this returns a single (Unicode) string with all the text in `t`'s descendants, in tree order (equivalent to accessing the attribute `t.text`). You can optionally pass, as the only positional argument, a string to use as separator. The default is the empty string, `''`. Pass the named parameter `strip=True` to have each string stripped of surrounding whitespace and all-whitespace strings skipped.

The following examples demonstrate these methods for getting strings from within tags:

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> print(soup.p.string)
```

```
None
```

```
>>> print(soup.p.b.string)
```

```
bold
```

```
>>> print(''.join(soup.strings))
```

```
Plain bold
```

```
>>> print(soup.get_text())
```

```
Plain bold
```

```
>>> print(soup.text)
```

```
Plain bold
```

```
>>> print(soup.get_text(strip=True))
```

```
Plainbold
```

Attribute references on instances of BeautifulSoup and Tag

The simplest, most elegant way to navigate down an HTML tree or sub-tree in `bs4` is to use Python's attribute reference syntax (as long as each tag you name is unique, or you care only about the first tag so named at each level of descent).

Given any instance *`t`* of `Tag`, a construct like *`t`*`.foo.bar` looks for the first tag `foo` within *`t`*'s descendants and gets a `Tag` instance *`ti`* for it, then looks for the first tag `bar` within *`ti`*'s descendants and returns a `Tag` instance for the `bar` tag.

It's a concise, elegant way to navigate down the tree, when you know there's a single occurrence of a certain tag within a navigable instance's descendants, or when the first occurrence of several is all you care about. But beware: if any level of lookup doesn't find the tag it's looking for, the attribute reference's value is **None**, and then any further attribute reference raises `AttributeError`.

BEWARE OF TYPOS IN ATTRIBUTE REFERENCES ON TAG INSTANCES

Due to this `BeautifulSoup` behavior, any typo you make in an attribute reference on a `Tag` instance gives a value of **None**, not an `AttributeError` exception—so, be especially careful!

`bs4` also offers more general ways to navigate down, up, and sideways along the tree. In particular, each navigable class instance has attributes that identify a single "relative" or, in plural form, an iterator over all relatives of that ilk.

contents, children, and descendants

Given an instance *t* of `Tag`, you can get a list of all of its children as `t.contents`, or an iterator on all children as `t.children`. For an iterator on all *descendants* (children, children of children, and so on), use `t.descendants`:

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> list(t.name for t in soup.p.children)
```

```
[None, 'b']
```

```
>>> list(t.name for t in soup.p.descendants)
```

```
[None, 'b', None]
```

The `names` that are **`None`** correspond to the `NavigableString` nodes; only the first one of them is a *child* of the `p` tag, but both are *descendants* of that tag.

parent and parents

Given an instance *n* of any navigable class, its parent node is *n*`.parent`:

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> soup.b.parent.name
```

```
'p'
```

An iterator on all ancestors, going upwards in the tree, is *n*`.parents`. This includes instances of `NavigableString`, since they have parents, too. An instance *b* of `BeautifulSoup` has *b*`.parent` **`None`**, and *b*`.parents` is an empty iterator.

next_sibling, previous_sibling, next_siblings, and previous_siblings

Given an instance *n* of any navigable class, its sibling node to the immediate left is *n*`.previous_sibling`, and the one to the immediate right is *n*`.next_sibling`; either or both can be **`None`** if *n* has no such sibling. An iterator on all left siblings, going leftward in the tree, is *n*`.previous_siblings`; an iterator on all right siblings, going rightward in the tree, is *n*`.next_siblings` (either or both iterators can be empty). This includes instances of `NavigableString`, since they have siblings, too. For an instance *b* of `BeautifulSoup`, *b*`.previous_sibling` and *b*`.next_sibling` are both **`None`**, and both of its sibling iterators are empty:

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> soup.b.previous_sibling, soup.b.next_sibling
```

```
('Plain ', None)
```

next_element, previous_element, next_elements, and previous_elements

Given an instance *n* of any navigable class, the node parsed just before it is *n*`.previous_element`, and the one parsed just after it is *n*`.next_element`; either or both can be **None** when *n* is the first or last node parsed, respectively. An iterator on all previous elements, going backward in the tree, is *n*`.previous_elements`; an iterator on all following elements, going forward in the tree, is *n*`.next_elements` (either or both iterators can be empty). Instances of `NavigableString` have such attributes, too. For an instance *b* of `BeautifulSoup`, *b*`.previous_element` and *b*`.next_element` are both **None**, and both of its element iterators are empty:

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> soup.b.previous_element, soup.b.next_element
```

```
('Plain ', 'bold')
```

As shown in the previous example, the `b` tag has no `next_sibling` (since it's the last child of its parent); however, it does have a `next_element` (the node parsed just after it, which in this case is the `'bold'` string it contains).

bs4 find... Methods (aka Search Methods)

Each navigable class in `bs4` offers several methods whose names start with `find`, known as *search methods*, to locate tree nodes that satisfy specified conditions.

Search methods come in pairs—one method of each pair walks all the relevant parts of the tree and returns a list of nodes satisfying the conditions, while the other one stops and returns a single node satisfying all the conditions as soon as it finds it (or **`None`** when it finds no such node). Calling the latter method is therefore like calling the former one with argument `limit=1`, then indexing the resulting one-item list to get its single item, but faster and more elegant.

So, for example, for any `Tag` instance *`t`* and any group of positional and named arguments represented by `...`, the following equivalence always holds:

```
just_one = t.find(...)
other_way_list = t.find_all(..., limit=1)
other_way = other_way_list[0] if other_way_list else None
assert just_one == other_way
```

The method pairs are listed in **Table 22-2**.

Table 22-2. `bs4` `find...` method pairs

`find`, `find_all`	*`b`*`.find(...)`, *`b`*`.find_all(...)` Searches the *descendants* of *`b`* or, when you pass named argument `recursive=`**`False`** (available only for these two methods, not for other search methods), *`b`*'s *children* only. These methods are not available on `NavigableString` instances, since they have no descendants; all other search methods are available on `Tag` and

	NavigableString instances. Since find_all is frequently needed, bs4 offers an elegant shortcut: calling a tag is like calling its find_all method. In other words, when *b* is a Tag, *b*(...) is the same as *b*.find_all(...). Another shortcut, already mentioned in **"Attribute references on instances of BeautifulSoup and Tag"**, is that *b*.foo.bar is like *b*.find('foo').find('bar').
find_next, find_all_next	*b*.find_next(...), *b*.find_all_next(...) Searches the next_elements of *b*.
find_next_ sibling, find_next_ siblings	*b*.find_next_sibling(...), *b*.find_next_siblings(...) Searches the next_siblings of *b*.
find_parent, find_parents	*b*.find_parent(...), *b*.find_parents(...) Searches the parents of *b*.
find_previous, find_all_ previous	*b*.find_previous(...), *b*.find_all_previous(...) Searches the previous_elements of *b*.
find_previous_ sibling, find_previous_ siblings	*b*.find_previous_sibling(...), *b*.find_previous_siblings(...) Searches the previous_siblings of *b*.

Arguments of search methods

Each search method has three optional arguments: *name*, *attrs*, and *string*. *name* and *string* are *filters*, as described in the following subsec-

tion; *attrs* is a dict, as described later in this section. In addition, as mentioned in **Table 22-2**, find and find_all only (not the other search methods) can optionally be called with the named argument recursive=**False**, to limit the search to children, rather than all descendants.

Any search method returning a list (i.e., one whose name is plural or starts with find_all) can optionally take the named argument limit: its value, if any, is an integer, putting an upper bound on the length of the list it returns (when you pass limit, the returned list result is truncated if necessary).

After these optional arguments, each search method can optionally have any number of arbitrary named arguments: the argument name can be any identifier (except the name of one of the search method's specific arguments), while the value is a filter.

Filters

A *filter* is applied against a *target* that can be a tag's name (when passed as the *name* argument), a Tag's string or a NavigableString's textual content (when passed as the *string* argument), or a Tag's attribute (when passed as the value of a named argument, or in the *attrs* argument). Each filter can be:

A Unicode string
: The filter succeeds when the string exactly equals the target.

A bytestring
: It's decoded to Unicode using utf-8, and the filter succeeds when the resulting Unicode string exactly equals the target.

A regular expression object (as produced by re.compile, covered in **"Regular Expressions and the re Module"**)
: The filter succeeds when the search method of the RE, called with the target as the argument, succeeds.

A list of strings
: The filter succeeds if any of the strings exactly equals the target (if any of the strings are bytestrings, they're decoded to Unicode using utf-8).

A function object

The filter succeeds when the function, called with the `Tag` or `NavigableString` instance as the argument, returns `True`.

`True`

The filter always succeeds.

As a synonym of "the filter succeeds," we also say "the target matches the filter."

Each search method finds all relevant nodes that match all of its filters (that is, it implicitly performs a logical **`and`** operation on its filters on each candidate node). (Don't confuse this logic with that of a specific filter having a `list` as an argument value. That one filter matches when any of the items in the `list` do; that is, the filter implicitly performs a logical **`or`** operation on the items of the `list` that is its argument value.)

name

To look for `Tags` whose `name` matches a filter, pass the filter as the first positional argument to the search method, or pass it as `name=`*`filter`*:

```
# return all instances of Tag 'b' in the document
soup.find_all('b') # or soup.find_all(name='b')

# return all instances of Tags 'b' and 'bah' in the document
soup.find_all(['b', 'bah'])

# return all instances of Tags starting with 'b' in the document
soup.find_all(re.compile(r'^b'))

# return all instances of Tags including string 'bah' in the document
soup.find_all(re.compile(r'bah'))

# return all instances of Tags whose parent's name is 'foo'
def child_of_foo(tag):
    return tag.parent.name == 'foo'

soup.find_all(child_of_foo)
```

string

To look for `Tag` nodes whose `.string`'s text matches a filter, or `NavigableString` nodes whose text matches a filter, pass the filter as `string=`*`filter`*:

```
# return all instances of NavigableString whose text is 'foo'
soup.find_all(string='foo')

# return all instances of Tag 'b' whose .string's text is 'foo'
soup.find_all('b', string='foo')
```

attrs

To look for `Tag` nodes that have attributes whose values match filters, use a `dict` *d* with attribute names as keys, and filters as the corresponding values. Then, pass *d* as the second positional argument to the search method, or pass `attrs=`*d*.

As a special case, you can use, as a value in *d*, **`None`** instead of a filter; this matches nodes that *lack* the corresponding attribute.

As a separate special case, if the value *f* of `attrs` is not a `dict`, but a filter, that is equivalent to having `attrs={'class': `*f*`}`. (This convenient shortcut helps because looking for tags with a certain CSS class is a frequent task.)

You cannot apply both special cases at once: to search for tags without any CSS class, you must explicitly pass `attrs={'class': None}` (i.e., use the first special case, but not at the same time as the second one):

```
# return all instances of Tag 'b' w/an attribute 'foo' and no 'bar'
soup.find_all('b', {'foo': True, 'bar': None})
```

MATCHING TAGS WITH MULTIPLE CSS CLASSES

Unlike most attributes, a tag's `'class'` attribute can have multiple values. These are shown in HTML as a space-separated string (e.g., `'<p class='foo bar baz'>...'`), and in `bs4` as a list of strings (e.g., *`t`*`['class']` being `['foo', 'bar', 'baz']`).

When you filter by CSS class in any search method, the filter matches a tag if it matches *any* of the multiple CSS classes of such a tag.

To match tags by multiple CSS classes, you can write a custom function and pass it as the filter to the search method; or, if you don't need other added functionality of search methods, you can eschew search methods and instead use the method *`t`*`.select`, covered in the following section, and go with the syntax of CSS selectors.

Other named arguments

Named arguments, beyond those whose names are known to the search method, are taken to augment the constraints, if any, specified in `attrs`. For example, calling a search method with *`foo=bar`* is like calling it with `attrs={'`*`foo`*`':` *`bar`*`}`.

bs4 CSS Selectors

`bs4` tags supply the methods `select` and `select_one`, roughly equivalent to `find_all` and `find` but accepting as the single argument a string that is a **CSS selector** and returning, respectively, the list of `Tag` nodes satisfying that selector or the first such `Tag` node. For example:

```
def foo_child_of_bar(t):
    return t.name=='foo' and t.parent and t.parent.name=='bar'

# return tags with name 'foo' children of tags with name 'bar'
soup.find_all(foo_child_of_bar)
```

```
# equivalent to using find_all(), with no custom filter function needed
soup.select('bar > foo')
```

`bs4` supports only a subset of the rich CSS selector functionality, and we do not cover CSS selectors further in this book. (For complete coverage of CSS, we recommend O'Reilly's ***CSS: The Definitive Guide***, by Eric Meyer and Estelle Weyl.) In most cases, the search methods covered in the previous section are better choices; however, in a few special cases, calling `select` can save you the (small) trouble of writing a custom filter function.

An HTML Parsing Example with BeautifulSoup

The following example uses `bs4` to perform a typical task: fetch a page from the web, parse it, and output the HTTP hyperlinks in the page:

```
import urllib.request, urllib.parse, bs4

f = urllib.request.urlopen('http://www.python.org')
b = bs4.BeautifulSoup(f)

seen = set()
for anchor in b('a'):
    url = anchor.get('href')
    if url is None or url in seen:
        continue
    seen.add(url)
    pieces = urllib.parse.urlparse(url)
    if pieces[0].startswith('http'):
        print(urllib.parse.urlunparse(pieces))
```

We first call the instance of class `bs4.BeautifulSoup` (equivalent to calling its `find_all` method) to obtain all instances of a certain tag (here, tag `'<a>'`), then the `get` method of instances of the tag in question to obtain the value of an attribute (here, `'href'`), or **`None`** when that attribute is missing.

Generating HTML

Python does not come with tools specifically meant to generate HTML, nor with ones that let you embed Python code directly within HTML pages. Development and maintenance are eased by separating logic and presentation issues through *templating*, covered in **"Templating"**. An alternative is to use `bs4` to create HTML documents in your Python code by gradually altering very minimal initial documents. Since these alterations rely on `bs4` *parsing* some HTML, using different parsers affects the output, as mentioned in **"Which parser BeautifulSoup uses"**.

Editing and Creating HTML with bs4

You have various options for editing an instance *`t`* of `Tag`. You can alter the tag name by assigning to *`t`*`.name`, and you can alter *`t`*'s attributes by treating *`t`* as a mapping: assign to an indexing to add or change an attribute, or delete the indexing to remove an attribute (for example, **`del`** *`t`*`['foo']` removes the attribute `foo`). If you assign some `str` to *`t`*`.string`, all previous *`t`*`.contents` (`Tag`s and/or strings—the whole subtree of *`t`*'s descendants) are discarded and replaced with a new `NavigableString` instance with that `str` as its textual content.

Given an instance *`s`* of `NavigableString`, you can replace its textual content: calling *`s`*`.replace_with('other')` replaces *`s`*'s text with `'other'`.

Building and adding new nodes

Altering existing nodes is important, but creating new ones and adding them to the tree is crucial for building an HTML document from scratch.

To create a new `NavigableString` instance, call the class with the text content as the single argument:

```
s = bs4.NavigableString(' some text ')
```

To create a new `Tag` instance, call the `new_tag` method of a `BeautifulSoup` instance, with the tag name as the single positional argument and (optionally) named arguments for attributes:

```
>>> soup = bs4.BeautifulSoup()
>>> t = soup.new_tag('foo', bar='baz')
>>> print(t)
```

```
<foo bar="baz"></foo>
```

To add a node to the children of a `Tag`, use the `Tag`'s `append` method. This adds the node after any existing children:

```
>>> t.append(s)
>>> print(t)
```

```
<foo bar="baz"> some text </foo>
```

If you want the new node to go elsewhere than at the end, at a certain index among *t*'s children, call `t.insert(n, s)` to put *s* at index *n* in `t.contents` (`t.append` and `t.insert` work as if *t* is a list of its children).

If you have a navigable element *b* and want to add a new node *x* as *b*'s `previous_sibling`, call `b.insert_before(x)`. If instead you want *x* to become *b*'s `next_sibling`, call `b.insert_after(x)`.

If you want to wrap a new parent node *t* around *b*, call `b.wrap(t)` (which also returns the newly wrapped tag). For example:

```
>>> print(t.string.wrap(soup.new_tag('moo', zip='zaap')))
```

```
<moo zip="zaap"> some text </moo>
```

```
>>> print(t)
```

```
<foo bar="baz"><moo zip="zaap"> some text </moo></foo>
```

Replacing and removing nodes

You can call `t.replace_with` on any tag `t`: the call replaces `t`, and all its previous contents, with the argument, and returns `t` with its original contents. For example:

```
>>> soup = bs4.BeautifulSoup(
...         '<p>first <b>second</b> <i>third</i></p>', 'lxml')
>>> i = soup.i.replace_with('last')
>>> soup.b.append(i)
>>> print(soup)
```

```
<html><body><p>first <b>second<i>third</i></b> last</p></body></html>
```

You can call `t.unwrap` on any tag `t`: the call replaces `t` with its contents, and returns `t` "emptied" (that is, without contents). For example:

```
>>> empty_i = soup.i.unwrap()
>>> print(soup.b.wrap(empty_i))
```

```
<i><b>secondthird</b></i>
```

```
>>> print(soup)
```

```
<html><body><p>first <i><b>secondthird</b></i> last</p></body></html>
```

`t.clear` removes *t*'s contents, destroys them, and leaves *t* empty (but still in its original place in the tree). `t.decompose` removes and destroys both *t* itself, and its contents:

```
>>> # remove everything between <i> and </i> but leave tags
>>> soup.i.clear()
>>> print(soup)
```

```
<html><body><p>first <i></i> last</p></body></html>
```

```
>>> # remove everything between <p> and </p> incl. tags
>>> soup.p.decompose()
>>> print(soup)
```

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

```
>>> # remove <body> and </body>
>>> soup.body.decompose()
>>> print(soup)
```

```
<html></html>
```

Lastly, `t.extract` extracts and returns `t` and its contents, but does not destroy anything.

Building HTML with bs4

Here's an example of how to use `bs4`'s methods to generate HTML. Specifically, the following function takes a sequence of "rows" (sequences) and returns a string that's an HTML table to display their values:

```
def mktable_with_bs4(seq_of_rows):
    tabsoup = bs4.BeautifulSoup('<table>')
    tab = tabsoup.table
    for row in seq_of_rows:
        tr = tabsoup.new_tag('tr')
        tab.append(tr)
        for item in row:
            td = tabsoup.new_tag('td')
            tr.append(td)
            td.string = str(item)
    return tab
```

Here is an example using the function we just defined:

```
>>> example = (
...     ('foo', 'g>h', 'g&h'),
...     ('zip', 'zap', 'zop'),
... )
>>> print(mktable_with_bs4(example))
```

```
<table><tr><td>foo</td><td>g&gt;h</td><td>g&amp;h</td></tr>
<tr><td>zip</td><td>zap</td><td>zop</td></tr></table>
```

Note that `bs4` automatically converts markup characters such as `<`, `>`, and `&` to their corresponding HTML entities; for example, `'g>h'` renders as `'g>h'`.

Templating

To generate HTML, the best approach is often *templating*. You start with a *template*—a text string (often read from a file, database, etc.) that is almost valid HTML, but includes markers, known as *placeholders*, where dynamically generated text must be inserted—and your program generates the needed text and substitutes it into the template.

In the simplest case, you can use markers of the form `{name}`. Set the dynamically generated text as the value for key `'name'` in some dictionary `d`. The Python string formatting method `.format` (covered in **"String Formatting"**) lets you do the rest: when `t` is the template string, `t.format(d)` is a copy of the template with all values properly substituted.

In general, beyond substituting placeholders, you'll also want to use conditionals, perform loops, and deal with other advanced formatting and presentation tasks; in the spirit of separating "business logic" from "presentation issues," you'd prefer it if all of the latter were part of your templating. This is where dedicated third-party templating packages come in. There are many of them, but all of this book's authors, having used and **authored** some in the past, currently prefer **`jinja2`**, covered next.

The jinja2 Package

For serious templating tasks, we recommend `jinja2` (available on **PyPI**, like other third-party Python packages, so, easily installable with **`pip install jinja2`**).

The **`jinja2` docs** are excellent and thorough, covering the templating language itself (conceptually modeled on Python, but with many differences to support embedding it in HTML, and the peculiar needs specific to presentation issues); the API your Python code uses to connect to `jinja2`, and expand or extend it if necessary; as well as other issues, from installation to internationalization, from sandboxing code to porting from other templating engines—not to mention, precious tips and tricks.

In this section, we cover only a tiny subset of `jinja2`'s power, just what you need to get started after installing it. We earnestly recommend studying `jinja2`'s docs to get the huge amount of extra, useful information they effectively convey.

The jinja2.Environment class

When you use `jinja2`, there's always an `Environment` instance involved—in a few cases you could let it default to a generic "shared environment," but that's not recommended. Only in very advanced usage, when you're getting templates from different sources (or with different templating language syntax), would you ever define multiple environments—usually, you instantiate a single `Environment` instance *env*, good for all the templates you need to render.

You can customize *env* in many ways as you build it, by passing named arguments to its constructor (including altering crucial aspects of templating language syntax, such as which delimiters start and end blocks, variables, comments, etc.). The one named argument you'll almost always pass in real-life use is `loader` (the others are rarely set).

An environment's `loader` specifies where to load templates from, on request—usually some directory in a filesystem, or perhaps some database (you'd have to code a custom subclass of `jinja2.Loader` for the latter purpose), but there are other possibilities. You need a `loader` to let templates enjoy some of `jinja2`'s most powerful features, such as ***template inheritance***.

You can equip *env*, as you instantiate it, with custom **filters**, **tests**, **extensions**, and so on (each of those can also be added later).

In the examples presented later, we assume *env* was instantiated with nothing but `loader=jinja2.FileSystemLoader('/path/to/templates')`, and not further enriched—in fact, for simplicity, we won't even make use of the `loader` argument.

env.get_template(*name*) fetches, compiles, and returns an instance of jinja2.Template based on what env.loader(*name*) returns. In the examples at the end of this section, for simplicity, we'll instead use the rarely warranted env.from_string(*s*) to build an instance of jinja2.Template from string *s*.

The jinja2.Template class

An instance *t* of jinja2.Template has many attributes and methods, but the one you'll be using almost exclusively in real life is:

render — *t*.render(...*context*...)
The *context* argument(s) are the same you might pass to a dict constructor—a mapping instance, and/or named arguments enriching and potentially overriding the mapping's key-to-value connections.
t.render(*context*) returns a (Unicode) string resulting from the *context* arguments applied to the template *t*.

Building HTML with jinja2

Here's an example of how to use a jinja2 template to generate HTML. Specifically, just like in **"Building HTML with bs4"**, the following function takes a sequence of "rows" (sequences) and returns an HTML table to display their values:

```
TABLE_TEMPLATE = '''\
<table>
{% for s in s_of_s %}
  <tr>
  {% for item in s %}
    <td>{{item}}</td>
  {% endfor %}
  </tr>
{% endfor %}
</table>'''
```

```
def mktable_with_jinja2(s_of_s):
    env = jinja2.Environment(
        trim_blocks=True,
        lstrip_blocks=True,
        autoescape=True)
    t = env.from_string(TABLE_TEMPLATE)
    return t.render(s_of_s=s_of_s)
```

The function builds the environment with option `autoescape=True`, to automatically "escape" strings containing markup characters such as <, >, and &; for example, with `autoescape=True`, `'g>h'` renders as `'g>h'`.

The options `trim_blocks=True` and `lstrip_blocks=True` are purely cosmetic, just to ensure that both the template string and the rendered HTML string can be nicely formatted; of course, when a browser renders HTML, it does not matter whether the HTML text itself is nicely formatted.

Normally, you would always build the environment with the `loader` argument and have it load templates from files or other storage with method calls such as *`t`* `=` *`env`*`.get_template(`*`template_name`*`)`. In this example, just to present everything in one place, we omit the loader and build the template from a string by calling the method *`env`*`.from_string` instead. Note that `jinja2` is not HTML- or XML-specific, so its use alone does not guarantee the validity of the generated content, which you should carefully check if standards conformance is a requirement.

The example uses only the two most common features out of the many dozens that the `jinja2` templating language offers: *loops* (that is, blocks enclosed in `{% for ... %}` and `{% endfor %}`) and *parameter substitution* (inline expressions enclosed in `{{` and `}}`).

Here is an example use of the function we just defined:

```
>>> example = (
...   ('foo', 'g>h', 'g&h'),
...   ('zip', 'zap', 'zop'),
```

```
... )
>>> print(mktable_with_jinja2(example))
```

```
<table>
  <tr>
    <td>foo</td>
    <td>g&gt;h</td>
    <td>g&amp;h</td>
  </tr>
  <tr>
    <td>zip</td>
    <td>zap</td>
    <td>zop</td>
  </tr>
</table>
```

1 The `BeautifulSoup` **documentation** provides detailed information about installing various parsers.

2 As explained in the `BeautifulSoup` **documentation**, which also shows various ways to guide, or completely override, `BeautifulSoup`'s guesses about encoding.