# 22. Structured Text: HTML - Python in a Nutshell, 3rd Edition



## 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 General Markup Language. In practice, many documents on the web use HTML in sloppy or incorrect ways. Browsers have evolved heuristics over the years to compensate for this, but even so, it still happens that a browser displays a wrongly marked-up web page in weird ways (don't blame the browser, if it's a modern one: 9 times out of 10, the blame is on the web page's author).

HTML is not suitable for much more than presenting documents on a browser. Complete, precise extraction of the information in the document, working backward from what most often amounts to the document's presentation, often turns out to be unfeasible. To tighten things up, HTML tried evolving into a more rigorous standard called XHTML. XHTML is similar to traditional HTML, but it is defined in terms of XML, and more precisely than HTML. You can handle well-formed XHTML with the tools covered in Chapter 23. However, as of this writing, XHTML does not appear to have enjoyed overwhelming success, getting scooped instead by the (non-XML) newest version, HTML5.

Despite the difficulties, it's often possible to extract at least some useful information from HTML documents (a task known as *screen-scraping*, or just *scraping*). Python's standard library tries to help, in both v2 and v3, supplying the `sgmllib`, `htmllib`, and `HTMLParser` modules in v2, and the `html` package in v3, for the task of parsing HTML documents, whether this parsing is for the purpose of presenting the documents, or, more typically, as part of an attempt to extract ("scrape") information. However, when you're dealing with somewhat-broken web pages, the third-party module BeautifulSoup usually offers your last, best hope. In this book, we mostly cover BeautifulSoup, ignoring most of the Python's standard library offerings "competing" with it.

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 alternative approach is *templating*, supported, for example, by the third-party module jinja2, covered in "The jinja2 Package".

## The html.entities (v2: htmlentitydefs) Module

The `html.entities` module in Python's standard library (in v2, the module is named `htmlentitydefs`) supplies a few attributes, all of them mappings. They come in handy whatever general approach you're using to parse, edit, or generate HTML, including the BeautifulSoup package covered in "The BeautifulSoup Third-Party Package".

`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 `'ä'`.

`entitydefs`

> In v3, a mapping from HTML entity names to Unicode equivalent single-character strings. For example, in v3, `entities.entitydefs['auml']` is `'ä'`, and `entities.entitydefs['sigma']` is `'σ'`. In v2, the equivalence is limited to the Latin-1 (AKA ISO-8859-1) encoding, and HTML character references are used

otherwise; therefore, in v2, `htmlentitydefs.entitydefs['auml']` is `'\xe4'`, and `htmlentitydefs.entitydefs['sigma']` is `'σ'`.

`html5` (v3 only)

In v3 only, `html5` is 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 —`BeautifulSoup` uses simple heuristics to compensate for likely HTML brokenness, and succeeds at this hard task with surprisingly good frequency. We strongly recommend BeautifulSoup version 4, also known as `bs4`, which supports both v2 and v3 smoothly and equally; we only cover `bs4` in this book (specifically, we document version 4.5 of `bs4`).

# Installing versus importing BeautifulSoup

You install the module, for example, by running, at a shell command prompt, `pip install beautifulsoup4`; but when you import it, in your Python code, use `import bs4`. If you have both v2 and v3, you may need to explicitly use `pip3` to install to v3, and/or `pip2` for v2.

## 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 next, an optional `parser` argument.

### Which parser BeautifulSoup uses

If you don't pass a `parser` argument, `BeautifulSoup` "sniffs around" to pick the best parser (you may get a warning in this case). If you haven't installed any other parser, `BeautifulSoup` defaults to `html.parser` from the Python standard library (to specify that parser explicitly, use the string `'html.parser'`, which is the module's name in v3; that also works in v2, where it uses the module `HTMLParser`). To get more control, say, 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`. Unless specified otherwise, the following examples use the default Python `html.parser`.

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 have installed the third-party package `lxml`, mentioned in "ElementTree", and `BeautifulSoup` 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`. (If you have installed `lxml`, you can also use it to parse HTML, by passing as the second argument `'lxml'`).

```
>>> import bs4>>> s = bs4.BeautifulSoup('<p>hello', 'html.parser')>>> sx = bs4.
BeautifulSoup('<p>hello', 'xml')>>> sl = bs4.BeautifulSoup('<p>hello', 'lxml')>>> s5 =
 bs4.BeautifulSoup('<p>hello', 'html5lib')>>> print(s, s.is_xml)<p>hello</p> False>>>
                    <?xml version="1.0" encoding="utf-8"?
print(sx, sx.is_xml)>                                     <p>hello</p> True>>> print(
sl, sl.is_xml)<html><body><p>hello</p></body></html> False>>> print(s5, s5.is_xml)
<html><head></head><body><p>hello</p></body></html> False
```

## Differences between parsers in fixing invalid HTML input

In the example, the `'html.parser'` tree builder inserts end-tag `</p>`, missing from the input. As also shown, other tree builders go further in repairing invalid HTML input, adding required tags such as `<body>` and `<html>`, to different extents depending on the parser.

### BeautifulSoup, Unicode, and encoding

`BeautifulSoup` uses Unicode, deducing or guessing the encoding when the input is a bytestring or binary file. For output, the `prettify` method returns a Unicode string representation of the tree, including tags, with attributes, plus extra whitespace and newlines to indent elements, to show the nesting structure; to have it instead return 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, in v3:

```
>>> 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>'
```

(In v2, the last output would be just `<p>hello</p>`, since v2 does not use the `b'...'` notation to print bytestrings.)

### The Navigable Classes of bs4

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

## Navigable classes 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`. We also call instances of navigable classes as *elements* or *nodes* in the tree.

Each instance of a "navigable class" lets you keep navigating, or 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 children 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).

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 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`, a construct like *`t`*`['foo']` looks for an HTML attribute named `foo` within `t`'s HTML attributes, and returns the string for the `foo` HTML 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, when `t` has no HTML attribute named `foo`.

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, `list`s of values), use the attribute *`t`*`.attrs`:

```
>>> s = bs4.BeautifulSoup('<p foo="bar"class= "ic">baz')>>> s.get('foo')>>> s.p.get('
                          {'foo': 'bar', 'class':
foo')'bar'>>> s.p.attrs['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 `if t.has_attr('foo'):`.

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.

### Getting an actual string

If 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, in v2, `unicode(`*`s`*`)`). Or, use *`s`*`.encode(`*`codec`*`='utf8')` to get a bytestring, and *`s`*`.decode()` to get text (Unicode). These give you the actual string, without references to the `BeautifulSoup` tree impeding garbage collection (`s` supports all methods of Unicode strings, so call those directly if they do all you need).

Given an instance `t` of `Tag`, you can get its single contained `NavigableString` instance with `t.string` (so `t.string.decode()` could be the actual text you're looking for). `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` (`''.join(t.strings)` could be the string you want). To ignore whitespace around each contained string, use the iterator `t.stripped_strings` (it also skips strings that are all-whitespace).

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

```
                              <p>Plain <b>bold</b>
>>> soup = bs4.BeautifulSoup('</p>                    ')>>> print(soup.p.string)None
>>> print(soup.p.b.string)bold>>> print(soup.get_text())Plain bold>>> print(soup.text)
Plain bold>>> print(soup.get_text(strip=True))Plainbold
```

The simplest, most elegant way to navigate down an HTML tree or subtree 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).

### Attribute references on instances of BeautifulSoup and Tag

Given any instance `t` of a `Tag`, a construct like `t.foo.bar` looks for the first tag `foo` within `t`'s descendants, gets a `Tag` instance `ti` for it, 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 look-up 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 typos in attribute references on Tag instances

Due to this BeautifulSoup behavior, any typo you may 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, 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`.

```
                              <p>Plain <b>bold</b>
>>> soup = bs4.BeautifulSoup('</p>                    ')>>> list(t.name for t in soup.
          [None,                                            [None, 'b',
p.children)'b']       >>> list(t.name for t in soup.p.descendants)None]
```

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

### parent, parents

Given an instance `n` of any navigable class, its parent node is *`n`*`.parent`; an iterator on all ancestors, going up 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.

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

### next_sibling, previous_sibling, next_siblings, 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. An instance `b` of `BeautifulSoup` has *`b`*`.previous_sibling` and *`b`*`.next_sibling` both `None`, and both of its sibling iterators are empty.

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

### next_element, previous_element, next_elements, 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. An instance `b` of `BeautifulSoup` has *`b`*`.previous_element` and *`b`*`.next_element` both `None`, and both of its element iterators are empty.

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

As shown in the previous example, the `b` tag has no `next_sibling` (since it's the last child of its parent); however, as shown here, 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 ("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 conditions you specify.

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, and the other one stops and returns a single node satisfying the conditions as soon as it finds it (or `None` when it finds no such node). So, calling the latter method is like calling the former one with argument `limit=1` , and indexing the resulting one-item list to get its single item, but a bit 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 Noneassert just_one == other_way
```

The method pairs are:

| | |
|---|---|
| **find, find_all** | *`b`*`.find(...)` *`b`*`.find_all(...)`<br><br>Searches the *descendants* of `b`, except that, if you pass as one of the named arguments `recursive=False` (available only for these two methods, not for other search methods), it searches `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.<br><br>Since `find_all` is frequently needed, `bs4` offers an elegant shortcut: calling a tag is like calling its `find_all` method. That is, *`b`*`(...)` is the same as *`b`*`.find_all(...)`.<br><br>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(...)`<br><br>Searches the `next_elements` of `b`. |
| **find_next_sibling, find_next_siblings** | *`b`*`.find_next_sibling(...)` *`b`*`.find_next_siblings(...)`<br><br>Searches the `next_siblings` of `b`. |
| **find_parent, find_parents** | *`b`*`.find_parent(...)` *`b`*`.find_parents(...)`<br><br>Searches the `parents` of `b`. |
| **find_previous, find_all_previous** | *`b`*`.find_previous(...)` *`b`*`.find_all_previous(...)`<br><br>Searches the `previous_elements` of `b`. |
| **find_previous_sibling, find_previous_siblings** | *`b`*`.find_previous_sibling(...)` *`b`*`.find_previous_siblings(...)`<br><br>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 later in this section; `attrs` is a `dict`, also described later in this section. In addition, `find` and `find_all` only (not the other search methods) can optionally be called with the 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 have the named argument `limit`, whose value, if passed, is an integer, putting an upper bound on the length of the list it returns.

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

### Search method arguments: 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 `utf8`, and then the filter succeeds when the resulting Unicode string exactly equals the target

A regular expression object (AKA RE, 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 `utf8`)

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 the relevant nodes that match all of its filters (that is, it implicitly performs a logical `and` operation on its filters on each candidate node).

### Search method arguments: name

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

```
                    # or
soup.find_all('b')  soup.find_all(name='b')
# returns all instances of Tag 'b' in the
document
soup.find_all(['b', 'bah'])
# returns all instances of Tags 'b' and 'bah' in the
document
soup.find_all(re.compile(r'^b'))
# returns all instances of Tags starting with 'b' in the
document
soup.find_all(re.compile(r'bah'))
# returns all instances of Tags including string 'bah' in the
document
def child_of_foo(tag):
  return tag.parent == 'foo'
soup.find_all(name=child_of_foo)
# returns all instances of Tags whose parent's name is
'foo'
```

### Search method arguments: 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`*:

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

### Search method arguments: attrs

To look for tag nodes who have attributes whose values match filters, use a `dict d` with attribute names as keys, and filters as the corresponding values. Then, pass it as the second positional argument to the search method, or pass it as `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 an `attrs` of `{'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):

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

## Matching tags with multiple CSS classes

Differently from most attributes, a tag can have multiple values for its attribute 'class'. 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 "bs4 CSS Selectors", and go with the syntax of CSS selectors.

### Search method arguments: 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's a CSS selector and returning the list of tag nodes satisfying that selector, or, respectively, the first such tag node.

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 the book *CSS: The Definitive Guide*, 3rd Edition [O'Reilly].) In most cases, the search methods covered in "bs4 find... Methods ("Search Methods")" are better choices; however, in a few special cases, calling select can save you the (small) trouble of writing a custom filter function:

```
def foo_child_of_bar(t):
    return t.name=='foo' and t.parent and t.parent.name=='bar'
soup(foo_child_of_bar)
# returns tags with name 'foo' children of tags with name
'bar'
soup.select('foo < bar')
# exactly equivalent, with no custom filter function
needed
```

## 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. In v2, the code is:

```
import urllib, urlparse, bs4

f = urllib.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 = urlparse.urlparse(url)
    if pieces[0]=='http':
        print(urlparse.urlunparse(pieces))
```

In v3, the code is the same, except that the function `urlopen` now lives in the module `urllib.request`, and the functions `urlparse` and `urlunparse` live in the module `urllib.parse`, rather than in the modules `urllib` and `urlparse`, respectively, as they did in v2 (covered in "URL Access" )—issues not germane to `BeautifulSoup` itself.

The example calls 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 class `Tag` 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 altering what start out as very minimal documents. Since these alterations rely on `bs4` *parsing* some HTML, using different parsers affects the output, as covered in "Which parser BeautifulSoup uses".

### Editing and Creating HTML with bs4

You can alter the tag name of an instance *`t`* of `Tag` by assigning to *`t`*`.name`; 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—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 tossed 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, just call the class, with the textual 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:

```
t = soup.new_tag('foo', bar='baz')print(t)<foo bar="baz"></foo>
```

To add a node to the children of a `Tag`, you can use the `Tag`'s `append` method to add the node at the end of the existing children, if any:

```
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 certan 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` was 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`: that 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`: that 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. For example:

```
soup.i.clear()print(soup)<html><body><p>first <i></i> last</p></body></html>soup.p.
decompose()print(soup)<html><body></body></html>
```

Lastly, `t.extract()` removes `t` and its contents, but—doing no actual destruction—returns `t` with its original contents.

## Building HTML with bs4

Here's an example of how to use bs4's tree-building 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(s_of_s):
  tabsoup = bs4.BeautifulSoup('<table>', 'html.parser')
  tab = tabsoup.table
  for s in s_of_s:
    tr = tabsoup.new_tag('tr')
    tab.append(tr)
    for item in s:
      td = tabsoup.new_tag('td')
      tr.append(td)
      td.string = str(item)
  return tab
```

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

```
example = (  ('foo', 'g>h', 'g&h'),  ('zip', 'zap', 'zop'),)print(mktable_with_bs4(
example))# prints: <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 "escapes" strings containing mark-up characters such as `<`, `>`, and `&`; for example, `'g>h'` renders as `'g>h'`.

## Templating

To generate HTML, the best approach is often *templating*. 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. 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 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 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.), but the one named argument you'll almost always pass in real-life use is `loader=...`.

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 powerful features, such as *template inheritance* (which we do not cover in this book).

You can equip `env`, as you instantiate it, with custom filters, tests, extensions, and so on; each of those can also be added later, and we do not cover them further in this book.

In the following sections' examples, 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. In real life, however, the loader is almost invariably set; other options, seldom.

`env.get_template(name)` fetches, compiles, and returns an instance of `jinja2.Template` based on what `env.loader(name)` returns. In the following examples, for simplicity, we'll actually use the rarely warranted `env.from_string(s)` to build an instance of `jinja2.Template` from (ideally Unicode, though a bytestring encoded in `utf8` will do) 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 previously 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>
 %  or s in s_of_s
{f  %}

<tr>
   %  or item in s
{  f  %}
    <td>{{item}}
</td>
   %
{  e  ndfor %}

</tr>
 %
{e  ndfor %}
</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 mark-up 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 itself is nicely formatted.

Normally, you would always build the environment with option `loader=...`, 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 in order to present everything in one place, we omit the loader and build the template from a string by calling method `env.from_string` instead. Note that `jinja2` is not HTML- or XML-specific, so its use alone does not guarantee validity of the generated content, which should be carefully checked 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))# prints: <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>
```

Except perhaps for its latest version (HTML5) when properly applied

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

As explained in the `BeautifulSoup` documentation, which also shows various ways to guide or override `BeautifulSoup`’s guesses.