NOTE: This is an archival document describing the now-obsolete 2.x version of Beautiful Soup. For the latest version, see the Beautiful Soup homepage.

How to Use Beautiful Soup

This document explains the use of Beautiful Soup: how to create a parse tree, how to navigate it, and how to search it.

Quick Start

Here's a Python session that demonstrates the basic features of Beautiful Soup.

```
>>> from BeautifulSoup import BeautifulSoup
>>> import re
>>>
>>> #Create the soup
... input = '''<html>
... <head><title>Page title</title></head>
... <body>
... <p id="firstpara" align="center">This is paragraph <b>one</b>.
... <p id="secondpara" align="blah">This is paragraph <b>two</b>.
... </html>'''
>>> soup = BeautifulSoup(input)
>>>
>>> #Search the soup
... titleTag = soup.html.head.title
>>> print titleTag
<title>Page title</title>
>>>
>>> print titleTag.string
Page title
>>>
>>> print len(soup('p'))
2
>>>
>>> print soup('p', {'align' : 'center'})
[<p id="firstpara" align="center">This is paragraph <b>one</b>.
</p>]
>>>
>>> print soup('p', {'align' : 'center'})[0]['id']
firstpara
>>>
>>> print soup.first('p', {'align' : re.compile('^b.*')})['id']
secondpara
>>>
>>> print soup.first('p').b.string
one
>>>
>>> print soup('p')[1].b.string
two
>>>
>>> #Modify the soup
... titleTag['id']='theTitle'
>>> titleTag.contents = ['New title.']
>>> print soup.html.head.title
<title id="theTitle">New title.</title>
```

Creating and Feeding the Parser

For most tasks your best bet is the `BeautifulSoup` parser. (See the section below, "Choosing a Parser" for situations when you might use the others). You can construct any of the parsers with no arguments, or you can pass in the text you want to parse.

If you don't pass in the text you want to parse to the parser constructor, you'll need to feed text into the parser afterwards, using the `feed()` method. Calling `feed()` more than once is the same as concatenating a lot of strings together and calling `feed()` once.

The `done()` method

Once you're done feeding text into the parser, call the `done()` method so that the parser knows to close any unclosed tags. If you pass in text to the parser constructor, you don't need to call `done()`; it'll happen automatically. This means that if you pass in text to the parser constructor, and *then* call `feed()`, you might not get the results you expect: there's a `done()` call in between.

Navigating the Parse Tree

When you feed a markup document into one of Beautiful Soup's parser classes, Beautiful Soup transforms the markup into a parse tree: a set of linked objects representing the structure of the document.

The parser object is the root of the parse tree. Below it are Tag objects and NavigableText objects. A Tag represents an SGML tag, as well as anything and everything encountered between that tag and its closing. A NavigableText object represents a chunk of ASCII or Unicode text. You can treat it just like a string, but it also has the navigation members so you can get to other parts of the parse tree from it.

For concreteness, here's a visual representation of the parse tree for the example HTML I introduced in the "Quick Start" section. I got this representation by calling `soup.prettify()`, and I'll use it throughout this section to illustrate the navigation. In this representation, a Tag that's underneath another tag in the parse tree is displayed with another level of indentation than its parent.

```
<html>
 <head>
  <title>Page title
  </title>
 </head>
 <body>
  <p id="firstpara" align="center">This is paragraph
   <b>one
   </b>.
  </p>
  <p id="secondpara" align="blah">This is paragraph
   <b>two
   </b>.
  </p>
 </body>
</html>
```

This is saying: we've got an `html` Tag which contains a `head` Tag and a `body` Tag. The `head` tag contains a `title` Tag, which contains a NavigableText object that says "Page title". The `body` Tag contains two `p` Tags, and so on. I got this diagram by running the HTML into a `BeautifulSoup` parser object and calling `prettify()` on it.

All Tag objects have all of the members listed below (though the actual value of the member may be `Null`). NavigableText objects have all of them except for `contents` and `string`.

`parent`

In the example above, the `parent` of the "head" Tag is the "html" Tag. The `parent` of the "html" Tag is the BeautifulSoup parser object itself. The parent of the parser object is `Null`. By following `parent` you can move up the parse tree.

`contents`

With `parent` you move up the parse tree; With `contents` you move down it. This is a list of Tag and NavigableText objects contained within a tag. Only the top-level parser object and Tag objects have `contents`; NavigableText objects don't have 'em.

In the example above, the `contents` of the first "p" Tag is a list containing a NavigableText ("This is paragraph"), a "b" Tag, and another NavigableText ("."). The `contents` of the "b" Tag is a list containing a NavigableText ("one").

`string`

For your convenience, if a tag has only one child node, and that child node is an ASCII or Unicode string, the child node is made available as `tag.string` as well as `tag.contents[0]`. In the example above, `soup.b.string` is a NavigableText representing the string "one". That's the string contained in the first "b" Tag in the parse tree. `soup.p.string` is `Null`, because the first "p" Tag in the parse tree has more than one child. `soup.head.string` is also `Null`, even though the "head" Tag has only one child, because that child is a Tag (the "title" Tag), not a string.

`nextSibling` and `previousSibling`

These members let you skip to the next or previous thing on the same level of the parse tree. For instance, the `nextSibling` of the "head" Tag is the "body" Tag, because the "body" Tag is the next thing directly beneath the "html" Tag. The `nextSibling` of the "body" tag is `Null`, because there's nothing else directly beneath the "html" Tag.

Conversely, the `previousSibling` of the "body" Tag is the "head" tag, and the `previousSibling` of the "head" Tag is `Null`.

Some more examples: the `nextSibling` of the first "p" Tag is the second "p" Tag. The `previousSibling` of the "b" Tag inside the second "p" Tag is the NavigableText "This is paragraph". The `previousSibling` of that NavigableText is `Null`, not anything inside the first "p" Tag.

`next` and `previous`

These members let you move through the document elements in the order they were processed by the parser, rather than in the order they appear in the tree. For instance, the `next` of the "head" Tag is the "title" Tag, not the "body" Tag. This is because the "title" Tag comes immediately after the "head" tag in the original document.

Where `next` and `previous` are concerned, a Tag's `contents` come before whatever is its `nextSibling`. You usually won't have to use these members, but sometimes it's the easiest way to get to something buried inside the parse tree.

Iterating over a Tag

You can iterate over the contents of a tag by treating the Tag itself as a list. `for i in soup.body:` is the same as `for i in soup.body.contents:`. Both will iterate over the direct children of the first 'body' Tag found in the parse tree. Similarly, to see how many child nodes a Tag has, you can call `len(tag)` instead of `len(tag.contents)`.

Navigate the parse tree by specifying tag names

It's easy to navigate the parse tree by referencing the name of the tag you want as a member of the parser or a Tag object. We've been doing it throughout these examples. In general, calling `tag.foo` returns the first child of that tag (direct or recursive) that happens to be a "foo" Tag. If there aren't any "foo" Tags beneath a tag, its `.foo` member is `Null`.

You can use this to traverse the parse tree, writing code like `soup.html.head.title` to get the title of an HTML document.

You can also use this to quickly jump to a certain part of a parse tree. For instance, if you're not worried about "title" Tags in weird places outside of the "head" Tag, you can just use `soup.title` to get an HTML document's title. `soup.p` jumps to the first "p" Tag inside a document, wherever it is. `soup.table.tr.td` jumps to the first column of the first row of the first table in the document.

These members actually alias to the `first()` method, which is covered below in the section "Searching the Parse Tree". I mention it here because the alias makes it very easy to zoom in on an interesting part of a well-known parse tree.

`soup.foo` versus `soup.fooTag`

An alternate form of this idiom lets you access the first 'foo' Tag as `.fooTag` instead of `.foo`. For instance, `soup.table.tr.td` could also be expressed as `soup.tableTag.trTag.tdTag`, or even `soup.tableTag.tr.tdTag`. This is useful if you like to be more explicit about what you're doing, or if you're parsing XML whose tags contain names that conflict with Beautiful Soup methods and members.

Suppose you were parsing XML that contained tags called "parent" or "contents". `soup.parent` won't trigger this idiom; it tries to find the parent of the parser object (which is `Null`). You can't use that idiom to find the first "parent" tag in the parse tree. Instead, use `soup.parentTag`.

The attributes of Tags

SGML tags have attributes, and so do the Tag objects created by the parser. For instance, each of the "p" Tags in the example above has an "id" attribute and an "align" attribute. You can access a Tag's attributes by treating the Tag as though it were a dictionary. `soup.p['id']` retrieves the "id" attribute of the first "p" Tag. NavigableText objects don't have attributes, only Tag objects.

The `Null` object

When navigating or searching the parse tree, you may encounter the `Null` object. `Null` is just like Python's `None`, but it's easier to work with:

- `Null.memberVariable` is `Null`.
- `Null.memberVariable.otherMemberVariable` is also `Null`.
- `Null().methodCall("arg1", "arg2")[0]['key']` is, you guessed it, `Null`.

Try doing any of that with `None` and you'd get an exception.

Why is this useful? Consider a line of Beautiful Soup code like `soup.head.title`. If Beautiful Soup used `None` instead of `Null`, that code would only work so long all your documents had a <head> tag containing a <title> tag. This is not a good assumption to make when you're dealing with real-world HTML.

For an ill-formed document, `soup.html` might return `None`, and then accessing the `title` member of `None` would throw an `AttributeException`. You'd need to check for `None` between getting the <head> and getting the <title>. But since Beautiful Soup actually returns `Null` if you ask for something that doesn't exist, and since you can access `Null.title` and get another `Null`, that code will work no matter what. You'll only have to check for `Null` once, at the end.

Searching the Parse Tree

Beautiful Soup provides a number of methods for finding `Tags` and text that match criteria you specify. These methods are available only to `Tag` objects and to the top-level parser objects, not to `NavigableText` objects. The methods in the next section are also available to `NavigableText` objects.

fetch(name, attrs, recursive, limit)

The `fetch()` method traverses the tree and finds all the tags that match the criteria you gave it.

Calling `fetch()` on the parser object searches the entire parse tree. Calling `fetch()` on a Tag object searches only the contents of that Tag.

`fetch()` takes four arguments:

- `name` describes to `fetch()` which tags the tags you want to retrieve. `name` can be any one of the following:
  1. A string. This will match only Tags with that specific name.

     For instance, if you wanted to get all of the "a" tags, you could call `soup.fetch('a')`.

  2. A list. This will match only Tags whose names are in the list.

     For instance, if you wanted to collect both "font" and "span" tags, you could call `soup.fetch(['font', 'div'])`

  3. A dictionary where the keys are the names of Tags you want. This is just like the list, but faster.

     For instance, if you wanted to collect both "font" and "span" tags, you could call `soup.fetch({'font' : None, 'div' : None})`

  4. A compiled regular expression. This will match only Tags whose names match the regular expression.

     For instance, if you wanted to get all tags whose names contained the letter "a", you could call `soup.fetch(re.compile('a'))`

  5. A callable object which takes a Tag object and returns a boolean. This object will be called once for each Tag encountered, and if it returns True then the tag is considered to match.

     For instance, if you wanted to get only tags whose 'id' attributes matched their names, you could call: `fetch(lambda(x):x.name==x['id'])`

- `attrs` is a map of restrictions on attribute values which are applied in addition to any restriction on `name`. The key of a key-value pair in the attribute map must be a string: the name of a particular attribute. The value of a key-value pair in the attribute map works like an argument to `name`: it can be a string, a list, a map, a regular expression, or a callable (a callable must take a string, not a Tag as with `name`).

  For instance, if you wanted to get only "a" Tags that had non-empty "href" attributes, you would call `soup.fetch('a', {'href':re.compile('.+')})`. If you wanted to get all tags that had an "width" attribute of 100, you would call `soup.fetch(attrs={'width':100})`

- `recursive` controls whether to search all the children of your starting `Tag`, or whether to search only the direct children of your starting Tag. By default, searches are recursive.
- `limit` lets you stop the search when a certain number of results have been found. If there are a thousand tables in your page, but you only need the fourth one, pass in 4 to `limit` and you'll save time. By default, there is no limit.

If you call a Tag object as though it were a function, you're actually calling that Tag's `fetch()` method. `tag('tag1', {'attr1':'val1'})` is the same as `tag.fetch('tag1', {'attr1':'val1'})`, and it's a little more concise.

`attrs` convenience alias

When scraping HTML, the most common use of `attrs` is to find tags with a particular CSS class. If you pass in a string (or a list, or a regular expression, or a callable) instead of a map to `attrs`, `fetch` will assume you want to match the "class" attribute. Therefore, `soup.fetch('foo', 'bar')` is the same as `soup.fetch('foo', {'class' : 'bar'})`. Using this alias can make your code look neater.

first(name, attrs, recursive)

The `first()` method traverses the tree and returns the first Tag that matches.

The arguments to `first()` are the same as to `fetch()`. It's basically a wrapper to `fetch()`, which returns either the first match or `Null` if there are no matches.

As with `fetch`, calling `first()` on the parser object searches the entire parse tree. Calling `first()` on a Tag object searches only the contents of that Tag.

As mentioned earlier, accessing the `foo` or `fooTag` member of the parser or a Tag object returns the first "foo" Tag in the parse tree. It's the same as calling `first("foo")`.

fetchText(text, recursive, limit)

This is like `fetch()` for finding text strings (NavigableText objects) instead of Tag objects. The `text` argument is like the `name` argument of `fetch()`. As with `name`, you can pass in any of five objects:

- A string you want to find.
- A list of strings you want to find.
- A dictionary where the keys are the strings you want to find. This is just like the list, but faster.
- A regular expression that matches the strings you want to find.
- A callable object that takes a string as input and returns whether or not you want to match that string.

The `recursive` and `limit` arguments works just like in `fetch()`.

firstText(text, recursive)

As `first()` is to `fetch()`, so is `firstText()` to `fetchText()`. It calls `fetchText()` and returns the first item in the list, or `Null` if there were no matches.

Searching Inside the Parse Tree

You can do most Beautiful Soup operations with the four methods in the previous section. However, sometimes you can't use them to get directly to the `Tag` or `NavigableText` you want. For example, consider some HTML like this:

```
from BeautifulSoup import BeautifulSoup
soup = BeautifulSoup('''<ul>
 <li>An unrelated list
</ul>

<h1>Heading</h1>
<p>This is <b>the list you want</b>:</p>
<ul>
 <li>The data you want
</ul>''')
```

There are a number of ways to navigate to that li tag that contains the data you want. The most obvious is this:

```
soup('li', limit=2)[1]
```

It should be equally obvious that that's not a very stable way to get that li tag. If you're only scraping this page once it doesn't matter, but if you're going to scrape it many times over a long period, such considerations become important. If the irrelevant list grows another li tag, you'll get that tag instead of the one you want, and your script will break or give the wrong data.

```
soup('ul', limit=2)[1].li
```

This is a little better, in that it can survive changes to the irrelevant list, but if the document grows another irrelevant list at the top, you'll get the first li tag of that list instead of the one you want. A more reliable way of referring to the ul tag you want would better reflect that tag's place in the structure of the document.

When you look at that HTML, you think of the list you want as 'the ul tag beneath the h1 tag'. The problem is that the tag isn't *contained* inside the h1 tag; it just comes after it. It's easy enough to get the h1 tag, but there's no way to get to the ul tag from there using `first()` and `fetch()`. You need to navigate to it with the `next` or `nextSibling` members:

```
s = soup.h1
while getattr(s, 'name', None) != 'ul':
    s = s.nextSibling
li = s.li
```

Or, if you think this might be more stable:

```
s = soup.firstText('Heading')
while getattr(s, 'name', None) != 'ul':
    s = s.next
li = s.li
```

Both of those examples are more trouble than you should need to go through, so the methods in this section provide a useful shorthand. These methods can be used whenever you find yourself wanting to write a while loop over one of the navigation members. Given a starting point somewhere in the tree, they navigate the tree in some way and keep track of `Tag` or `NavigableText` objects that match the criteria you specify. Instead of the loops in the example code above, you can just write this:

```
soup.h1.findNextSibling('ul').li
```

Or this:

```
soup.firstText('Heading').findNext('ul').li
```

All of these methods take the same arguments as `first` or `fetch`: an optional way of matching a tag name, an optional way of matching tag attributes, and an optional way of matching text.

There are two methods for each navigation member. The methods whose names look like `fetchFoos` take an optional limit, like `fetch`, and return a list of matches. The methods whose names look like `findFoo` are convenience methods which will stop searching the tree after encountering a match, and will return that match as a scalar.

These methods are available on the tree as a whole, and also on `Tag` and `NavigableText` objects. `first` and `fetch`, the methods covered in the previous section, are not available for `NavigableText` objects, because those methods search through a `Tag`'s children, and `NavigableText` objects can't have any children.

`findNextSibling(name, attrs, text)` and `fetchNextSiblings(name, attrs, text, limit)`

These methods repeatedly follow an object's `nextSibling` member, gathering `Tag` or `NavigableText` objects that match the criteria you specify.

`findPreviousSibling(name, attrs, text)` and `fetchPreviousSiblings(name, attrs, text, limit)`

These methods repeatedly follow an object's `previousSibling` member, gathering `Tag` or `NavigableText` objects that match the criteria you specify.

`findNext(name, attrs, text)` and `fetchNext(name, attrs, text, limit)`

These methods repeatedly follow an object's `next` member, gathering `Tag` or `NavigableText` objects that match the criteria you specify.

`findPrevious(name, attrs, text)` and `fetchPrevious(name, attrs, text, limit)`

These methods repeatedly follow an object's `previous` member, gathering `Tag` or `NavigableText` objects that match the criteria you specify.

`findParent(name, attrs)` and `fetchParents(name, attrs, limit)`

These methods repeatedly follow an object's `parent` member, gathering `Tag` objects that match the criteria you specify. Since a `NavigableText` can have no children, you'll never get a `NavigableText` object while calling `findParent` or `fetchParents`. That's why these methods don't take a `text` argument.

Printing out the parse tree

If you need to make changes to the parse tree and print it back out, or just look at how Beautiful Soup decided to parse some bad HTML, you have a couple options for turning the parse tree back into a string.

`str()` and `unicode()`

The parser objects, as well as each Tag and NavigableText object, can be printed out as strings. This string will have no unneccessary whitespace, and all tags will either be self-closing or have corresponding closing tags in what Beautiful Soup guesses is the right place. One useful thing you can do with this is clean up HTML into something approaching XHTML.

`prettify()`

The `prettify()` method turns the parse tree (or a portion of it) into a pretty-printed string. This is just like the regular string you'd get with `print soup`, except it uses whitespace to show the structure of the parse tree. Every tag will start a new line, and a tag's children will be indented one more level than its parent.

Remember from earlier examples that Beautiful Soup turned this:

```
<html>
<head><title>Page title</title></head>
<body>
<p id="firstpara" align="center">This is paragraph <b>one</b>.
<p id="secondpara" align="blah">This is paragraph <b>two</b>.
</html>
```

into this:

```
 <html>
  <head>
   <title>Page title
   </title>
  </head>
  <body>
   <p id="firstpara" align="center">This is paragraph
    <b>one
    </b>.
   </p>
   <p id="secondpara" align="blah">This is paragraph
    <b>two
    </b>.
   </p>
  </body>
 </html>
```

Choosing a parser

Beautiful Soup provides four classes that implement different parsing strategies. You'll need to choose the right one depending on your task. For most tasks you'll be able to use `BeautifulSoup`, but sometimes one of the other classes might make things easier for you.

`BeautifulSoup`

The most popular Beautiful Soup class, this class parses HTML as seen in the real world. It contains heuristics about common HTML usage and mis-usage.

Raw	Parsed with `BeautifulSoup`

`<i>This <span title="a">is<br> some <html>invalid</htl %> HTML.`
`<sarcasm>It's so great!</sarcasm>`
	`<i>This`
	` <span title="a">is`
	`  <br /> some`
	`  <html>invalid HTML.`
	`   <sarcasm>It's so great!`
	`   </sarcasm>`
	`  </html>`
	` </span>`
	`</i>`

`BeautifulStoneSoup`

This class parses any XML-like language. It contains no special language- or schema-specific heuristics. If you want to define a set of self-closing tags for your XML schema, you'll need to subclass this class.

Raw	Parsed with `BeautifulStoneSoup`
`<foo key1="value1">This is some <bar>invalid</baz> XML.`	`<foo key1="value1">This is some` `<bar>invalid XML.` `</bar>` `</foo>`

`ICantBelieveItsBeautifulSoup`

This is a subclass of `BeautifulSoup` with different heuristics. It's geared towards dealing with bizarre but valid HTML, like HTML that contains nested inline tags that don't do anything when you nest them:

Raw	Parsed with `BeautifulSoup`	Parsed with `ICantBelieveItsBeautifulSoup`
`<b>This text is <b>bolded twice</b> for some reason.`	`<b>This text is` `</b>` `<b>bolded twice` `</b> for some reason.`	`<b>This text is` `<b>bolded twice` `</b> for some reason.` `</b>`

`BeautifulSOAP`

This is a convenience subclass of `BeautifulStoneSoup` which makes it easier to deal with XML documents (like SOAP messages) that put data in tiny sub-elements when it would be more convenient to put them in attributes of the parent element.

Raw	Parsed with `BeautifulSOAP`
`<foo><bar>baz</bar></foo>`	`<foo bar="baz">` `<bar>baz` `</bar>` `</foo>`

Advanced Topics: Building a custom parser

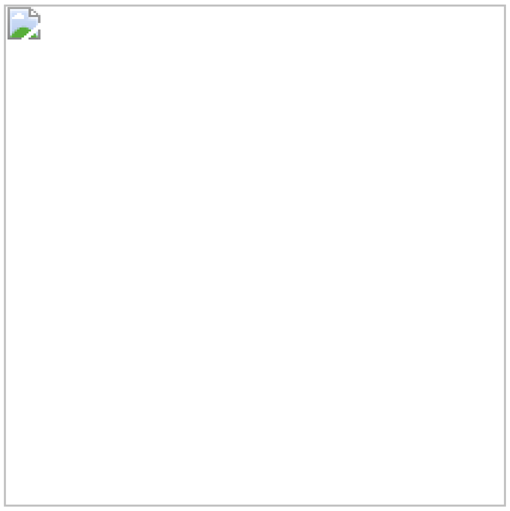
As befits an "advanced topics" section, I haven't written this yet.

Sanitizing Bad Data with Regexes

...

Customizing the Tag Maps

...



The Fish-Footman began by... saying, in a solemn tone, "For the Duchess. An invitation from the Queen to play croquet." The Frog-Footman repeated, in the same solemn tone, only changing the order of the words a little, "From the Queen. An invitation for the Duchess to play croquet."

Then they both bowed low, and their curls got entangled together.

- Homepage
- Download
- What's new
- Documentation
- Examples
- FAQ
- To-do list
- Contributors

Document tree:

http://www.crummy.com/

software/

BeautifulSoup/

documentation.html

Site Search: