HTML Processing in Python

Overview

HTML processing in Python consists of three steps:

1. Download of the HTML file from the World Wide Web.
2. Parsing of the HTML file using a Python HTML parser.
3. Construction of the HTML parse tree for further use and traversal.

Strictly speaking, the third step is optional, but it is the third step we will concentrate on as it makes actual work with HTML documents in Python really straightforward.

HTML File Download

Both Python 2 and Python 3 come with a `urllib` library that allows for access and download of HTML files. The `urllib` library documentation for Python 2.6 is at

https://docs.python.org/2.6/library/urllib.html

while for Python 3 the documentation is found at

https://docs.python.org/3.3/library/urllib.request.html

We largely concentrate on Python 3 here. For Python3 the `urllib` librarly has been split into `urllib.request`, `urllib.parse` and `urllib.error`, which are responsible for downloading URLs, parsing them and error handling respectively. For our purposes, we need to use `urllib.request`.

The following `urllib.request` function is of relevance:

- `urllib.request.urlopen(url)`: the simplest form of the `urlopen()` function that sends an HTTP or HTTPS request to acquire the supplied `url`, presented as either a `string` or a Python `Request` object.

Usage.

```
>>> import urllib.request
>>> url = "http://www.csc.calpoly.edu/~dekhtyar/DATA301-Spring2016/test.html"
>>> testpage = urllib.request.urlopen(url)
>>> testpage
<http.client.HTTPResponse object at 0x7f563e9513d0>
>>>
```

`testpage` in this example will contain the object that can be passed as input to `BeautifulSoup` (see below).

HTML File parsing

Python has a variety of HTML parsers. A simple HTML parser reads through the input HTML document and operates on an event-based basis, not unlike a SAX XML parser: for each tag and content parsed, an HTML parser will emit a message that can be passed up to the calling functions.

We will not be using HTML file parser directly, but our HTML processor, `BeautifulSoup` uses one under the hood to produce its HTML parse tree. A parser name needs to be provided to `BeautifulSoup` in order to construct the tree.

There are two HTML parsers that can be used on our system:

- `html.parser`: the built-in Python HTML parser, available on every Python install and usable by default with `BeautifulSoup`
- `lxml` parser, a better, more robust HTML/XML/XHTML parser that may be available with some Python installs, but not with all of them.

HTML Parse Tree Creation with BeautifulSoup

`BeautifulSoup` is a Python library for HTML parse tree construction and access to HTML documents in the style of DOM. It is installed both for Python 2 and Python 3 our system. It documentation is available at

https://www.crummy.com/software/BeautifulSoup/bs4/doc/

To *ingest* an HTML document and convert it to a parse tree (often referred to as `soup`), import and use `BeautifulSoup()` constructor. The constructor can be invoked in a number of ways:

`BeautifulSoup` call	Explanation
`BeautifulSoup(URLObject, Parser)`	Build parse tree for HTML document specified by the URL object using `Parser`
`BeautifulSoup(open(filename), Parser)`	Build parse tree for HTML document from a given file using `Parser`
`BeautifulSoup(HTMLString, Parser)`	Build the parse tree for data in `HTMLString` using `Parser`
`BeautifulSoup(in)`	Build a parse tree for any object (see above) using default HTML parser

Consider the following example:

```
>>> from bs4 import BeautifulSoup
>>> import urllib.request
>>> url = "http://www.csc.calpoly.edu/~dekhtyar/DATA301-Spring2016/test.html"
>>> testpage = urllib.request.urlopen(url)
>>> soup = BeautifulSoup(testpage, 'html.parser')
>>> soup
<html>
<head><title>DATA 301 test</title></head>
<body bgcolor="#ffffff">
<h2>Test HTML file</h2>
<p> This is <a href="index.html">DATA 301</a> course.</p>
<p>
<table>
<tr><td>Step 1</td><td>Download</td>
<tr><td>Step 2</td><td>Parse</td>
<tr><td>Step 3</td><td>Build tree</td>
<tr><td>Step 4</td><td><b>?</b></td>
<tr><td>Step 5</td><td><b><u>Profit!</u></b></td>
</tr></tr></tr></tr></tr></table>
</p>
<hr>
</hr></body>
</html>
```

BeautifulSoup Objects

`BeautifulSoup` works with the following types of objects:

- `BeautifulSoup` objects representing the entire HTML parse tree. The `soup` variable in the example above is a `BeautifulSoup` object.
- `Tag` objects. `Tag` objects represent individual HTML elements and their contents. They are the analogs of the DOM element objects.
- `Attribute` objects. These represent HTML attributes of individual HTML elements. `Tag` objects include `Attribute` dictionaries.
- `Navigable String` objects represent the contents of a single HTML element (tag). These objects largely behave as strings, but they are also aware of their "position" in the parse tree and allow for navigtion.

Navigating the HTML Parse Tree

A `BeautifulSoup` object is essentially a DOM tree with some shortcuts made to improve on the DOM API.

Accessing individual tags. Each `BeautifulSoup` object has attributes representing each tag inside it that can be accessed by tag name using the

```
soupVariable.tagName
```

syntax. Such invocations return *the first tag object* with a given name. For example (picking up where the previous example left off)[1]:

```
>>> soup.head
<head><title>DATA 301 test</title></head>
>>> soup.title
<title>DATA 301 test</title>
>>> soup.p
<p> This is <a href="index.html">DATA 301</a> course.</p>
>>> soup.tr
<tr><td>Step 1</td><td>Download</td>
<tr><td>Step 2</td><td>Parse</td>
<tr><td>Step 3</td><td>Build tree</td>
<tr><td>Step 4</td><td><b>?</b></td>
<tr><td>Step 5</td><td><b><u>Profit!</u></b></td>
</tr></tr></tr></tr></tr>
>>> soup.td
<td>Step 1</td>
>>>
```

Accessing all tags with a given name. To obtain a list of tags with the same name use `find_all()` method and pass the tag name to it as a `string`. For example:

```
>>> soup.find_all('head')
[<head><title>DATA 301 test</title></head>]
>>> soup.find_all('p')
[<p> This is <a href="index.html">DATA 301</a> course.</p>, <p>
<table>
<tr><td>Step 1</td><td>Download</td>
<tr><td>Step 2</td><td>Parse</td>
<tr><td>Step 3</td><td>Build tree</td>
<tr><td>Step 4</td><td><b>?</b></td>
<tr><td>Step 5</td><td><b><u>Profit!</u></b></td>
</tr></tr></tr></tr></tr></table>
</p>]
>>> soup.find_all("a")
[<a href="index.html">DATA 301</a>]
>>> soup.find_all("td")
[<td>Step 1</td>, <td>Download</td>, <td>Step 2</td>, <td>Parse</td>, <td>Step 3</td>, <td>Build tree
>>>
```

In these examples, `find_all()` returns a list of `tag` objects.

Acessing child objects. Each `tag` object has `.contents` and `.children` attributes. The child tag and string objects forming the specific tag can be accessed as a `list` via the

```
tagVariable.contents
```

attribute. The result is a list and can be accessed using standard list functionality.

[1]Notice the way `<tr>` objects are handled. This is due to `<tr>` tags not being closed in the original HTML document. The parser inserted the closing tags where it could.

```
>>> para = soup.p
>>> para
<p> This is <a href="index.html">DATA 301</a> course.</p>
>>> para.contents
[' This is ', <a href="index.html">DATA 301</a>, ' course.']
>>> para.contents[0]
' This is '
>>> para.contents[1]
<a href="index.html">DATA 301</a>
>>> len(para.contents)
3
>>>
```

To access the contents of a tag as a `generator` use the `.children` attribute:

```
>>> for piece in para.children:
...     print(piece)
...
 This is
<a href="index.html">DATA 301</a>
 course.
```

Access descendant objects. You can also access all the descendants of a tag in a single list using `.descendants` attribute. This attribute also returns a `genearator`

```
>>> for thing in para.descendants:
...    print(thing)
...
 This is
<a href="index.html">DATA 301</a>
DATA 301
 course.
>>>
```

Access string content of the tag. The `.string` attribute of a tag retrieves its string content. If the

```
tag has co>>> para.string
>>> para.contents[1].string
'DATA 301'
>>> para.contents[0].string
' This is '
>>> para.contents[2].string
' course.'
>>> mpound content, the \texttt{.string} attrbute does not return anything.
```

All string components of a tag can be accessed via `.strings` or `.stripped_strings` attributes that produce generators. Use the latter to strip the whitespace.

```
>>> para.strings
<generator object _all_strings at 0x7f563eca30f0>
>>> for s in para.strings:
...    print(s)
...
 This is
```

```
DATA 301
 course.
>>>
```

`.stripped_strings` attribute strips the whitespace from a string.

Access attributes. For each HTML element its ttributes are stored as a dictionary inside the element node. The attributes can be accessed directly using the Python dictionary notation from the element node. Additionally, the dictionary itself can be retrieved using the `.attrs` attribute of the element node.

```
>>> body = soup.body
>>> body.attrs
{'bgcolor': '#ffffff'}
>>> body['bgcolor']
'#ffffff'
```

Access parents and ancestors of an element. The `.parent` attribute of an HTML element node retrieves the parent node.

```
>>> body = soup.body
>>> body.attrs
{'bgcolor': '#ffffff'}
>>> body['bgcolor']
'#ffffff'
```

The `.parents` attribute is a generator for the list of ancestor nodes.

```
>>> for node in generations:
...     print(node)
...
<head><title>DATA 301 test</title></head>              ## parent node
<html>
<head><title>DATA 301 test</title></head>              ## grandparent node
<body bgcolor="#ffffff">
...                        ## part of output removed for brevity
</body>
</html>
```

Access siblings. HTML elements have attributes `.next_sibling` and `previous_sibling` to access sibling elements.

```
>>> tr = soup.tr
>>> tr
<tr><td>Step 1</td><td>Download</td>
<tr><td>Step 2</td><td>Parse</td>
<tr><td>Step 3</td><td>Build tree</td>
<tr><td>Step 4</td><td><b>?</b></td>
<tr><td>Step 5</td><td><b><u>Profit!</u></b></td>
</tr></tr></tr></tr></tr>
>>> td = tr.contents[0]
>>> td
<td>Step 1</td>
>>> td.next_sibling
```

```
<td>Download</td>
>>> td.next_sibling.next_sibling
'\n'
>>> td.next_sibling.next_sibling.next_sibling
<tr><td>Step 2</td><td>Parse</td>
<tr><td>Step 3</td><td>Build tree</td>
<tr><td>Step 4</td><td><b>?</b></td>
<tr><td>Step 5</td><td><b><u>Profit!</u></b></td>
</tr></tr></tr></tr>
>>> td.next_sibling.next_sibling.next_sibling.previous_sibling
'\n'
>>> td.next_sibling.next_sibling.next_sibling.previous_sibling.previous_sibling
<td>Download</td>
>>>
```

Simiarly, `.next_siblings` and `.previous_siblings` return list generators that span over the lists of next and previous siblings of the element node.

```
>>> p = soup.p
>>> p
<p> This is <a href="index.html">DATA 301</a> course.</p>
>>> t = p.contents[0]
>>> t
' This is '
>>> for f in t.next_siblings:
...    print(f)
...
<a href="index.html">DATA 301</a>
 course.
>>> t = p.contents[-1]
>>> t
' course.'
>>> for t in t.previous_siblings:
...    print(t)
...
<a href="index.html">DATA 301</a>
 This is
>>>
```

Filtering HTML Documents

Beautiful Soup has the following functions for searching inside the HTML documents:

Function	Explanation
`find_all(tag, arguments)`	Find all descendant elements that match `tag` name
`find(tag, arguments)`	Find the first descendant element that matches `tag`
`find_parents(tag, arguments)`	Find ancestors of current element that match `tag`
`find_parent(tag, arguments)`	Find first ancestor of current element that matches `tag`
`find_next_siblings(tag, arguments)`	Find all following siblings of current element that match `tag`
`find_next_sibling(tag, arguments)`	Find the first following sibling of current element that matches `tag`
`find_previous_siblings(tag, arguments)`	Find all preceding siblings of current element that match `tag`
`find_previous_sibling(tag, arguments)`	Find the first preceding sibling of current element that matches `tag`
`find_all_next(tag, arguments)`	Find all following (in document order) elemenents that match `tag`
`find_next(tag, arguments)`	Find the first following in document order element that matches `tag`
`find_all_previous(tag, arguments)`	Find all preceding in document order elements that match `tag`
`find_previous(tag, arguments)`	Find the first preceding in document order element that matches `tag`

All of these functions are separated into two categories:

- **Functions that retrieve all matching elements.** These functions, when called, match all elements in the scope of the function to the given `tag` expression and retrieve **all elements** from the scope that match `tag`. These functions are:
  - `find_all()`
  - `find_parents()`
  - `find_next_siblings()`
  - `find_previous_siblings()`
  - `find_all_next()`
  - `find_all_previous()`
- **Functions that retrieve the first matching element.** These functions, when called, find the first element in the scope of the function that matches the `tag` expression, and return just this one element. The functions that behave this way are:
  - `find()`
  - `find_parent()`
  - `find_next_sibling()`
  - `find_previous_sibling()`
  - `find_next()`
  - `find_previous()`

All the functions take the same set of arguments, and except for the scope and the category, operate in the same way.

Arguments. The full list of the arguments for each of the functions above is as follows. At least one argument must be present, but any argument is *optional*. Argument `limit` is only present in functions that retrieve *all matching elements*.

Argument	Explanation
`tag` (or `name`)	name of the tag
`attrs`	CSS class filter
`recursive`	recursive flag (if set to `False` do not recurse the search
`string`	content of element filter
`limit`	limit the search results to top elements in the list
`keyword`	attribute name filters

`name` argument. The `name` argument is an element name (tag) filter. `Beautiful Soup` takes the following types of values for this argument:

- **A single string.** The sets up the search for a single tag name.

```
>>> soup.find_all('b')
[<b>?</b>, <b><u>Profit!</u></b>]
```

- **A list of strings.** This sets up search for any element that matches at least one of the tag names.

```
>>> soup.find_all(['b', 'title'])
[<title>DATA 301 test</title>, <b>?</b>, <b><u>Profit!</u></b>]
```