Lab 13

BeautifulSoup

Lab Objective: *Virtually everything rendered by an internet browser as a web page uses HTML. As a result, learning HTML is key to any kind of internet programming. BeautifulSoup is a Python package that helps navigate HTML documents. In this lab, we learn how to load HTML documents into BeautifulSoup and navigate the resulting BeautifulSoup object.*

HTML

HTML, or Hyper Text Markup Language is the standard markup language to create webpages. Just like XML, HTML tags describe different document content and are surrounded by angle brackets. Most tags can be combined with attributes such as `id` or `class` to help identify individual tags and make navigating the HTML tree much more simple. A list of all the current HTML tags can be found at `http://htmldog.com/reference/htmltags`. Here is an example:

```
<html>
    <body>
        <p>
            Click <a id='info' href='http://www.example.com'>here</a> for more ↩
                information.
        </p>
    </body>
</html>
```

The above example would output a single line

```
Click here for more information.
```

with 'here' being a clickable link to the website `http://www.example.com`.

While less readable, this HTML code can also be written as a single line as follows:

```
<html><body><p>Click <a id='info' href='http://www.example.com/info'>here</a> for↩
     more information.</p></body></html>
```

If a given tag doesn't contain any text or other tags, it can be written in a single pair of brackets as

```
<*tag_name*  ... *attributes*/>
```

The HTML of a website is very easy to view. Go to any website, such as `http://www.example.com`, in whatever browser is most convenient. Once on the website, right click with the mouse pointer and look for 'View Page Source' or a similarly worded option. Click it and the browser will open a new browser with the HTML code for your site. Some code is easy to follow, other code not so much.

Problem 1. Go to the website `http://www.example.com` and open the source code. What are all the tags used? What is the value of the `type` attribute associated with the `style` tag? Write a function that returns a list of all the tags used and the value of the `type` attribute.

Loading HTML into BeautifulSoup

Now that we know what HTML is, we can use BeautifulSoup to create a `BeautifulSoup` object. BeautifulSoup is a library capable of pulling data out of HTML scripts and files. BeautifulSoup works with a parser to provide commands to navigate and search the resulting HTML tree. Make sure the module `bs4` is installed in your Python packages. This section takes most of its material from `http://www.crummy.com/software/BeautifulSoup/bs4/doc/index.html`.

First we want a variable to store our HTML code as a string.

```
>>> doc = """
...     <html><body><p>
...     Click <a id='info' href='http://www.example.com/info'>here</a> for more ↩
    information.
...     </p></body></html>
...     """
```

Next, we import `BeautifulSoup` from the `bs4` module. We call `BeautifulSoup()`, which takes as parameters the HTML string and an HTML parser. It returns a `BeautifulSoup` object, which represents the document as a nested data structure.

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

Including an HTML parser is optional, but a warning is given if one is not included. If that's the case, BeautifulSoup uses the HTML parser included in Python's standard library. Although other parsers are permitted, we have no need for them in our examples.

Once the document is stored, we can use `prettify()` to view the HTML. The `prettify()` method returns a string that can be printed to represent the BeautifulSoup object in a readable format. This will be useful later to make sure we are getting the correct HTML from websites.

```
>>> print(soup.prettify())
<html>
  <body>
    <p>
      Click <a id='info' href='http://www.example.com/info'>here</a> for more ↩
          information.
    </p>
  </body>
</html>
```

Problem 2. Write a function that loads the following string into BeautifulSoup, then prettifies it. Print the prettified string.

```
html_doc = """
<html><head><title>The Three Stooges</title></head>
<body>
<p class="title"><b>The Three Stooges</b></p>

<p class="story">Have you ever met the three stooges? Their names are
<a href="http://example.com/larry" class="stooge" id="link1">Larry</a>,
<a href="http://example.com/mo" class="stooge" id="link2">Mo</a> and
<a href="http://example.com/curly" class="stooge" id="link3">Curly</a>;
and they are really hilarious.</p>

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

> **Note**
>
> Note that the `<html>` and `<body>` tags are never actually closed. The parser used with `bs4` will automatically close these hanging tags, so don't get too stressed out by this example.

Navigating the HTML Tree

Since `BeautifulSoup()` returns an object which acts like a nested data structure, navigating it is very intuitive. We will use the following for the rest of the section, unless otherwise specified.

```
>>> soup = BeautifulSoup(html_doc, 'html.parser')
```

where `html_doc` is the document defined in problem 2.

By Tag Name

Because of the way BeautifulSoup objects store HTML tags, accessing them is as simple as calling the tag name. The output will be the called tag plus any nested

tags or text. Below are some examples.

```
>>> soup.head
<head><title>The Three Stooges</title></head>

>>> soup.title
<title>The Three Stooges</title>
```

It is even possible to continue navigation down the tree through tags contained within tags.

```
>>> soup.body.b
<b>The Three Stooges</b>
```

Notice there are three `<a>` tags. When there are two or more tags of the same name, calling that tag only returns the *first* tag by that name.

```
>>> soup.a
<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>
```

Tag Properties

In addition to viewing the entire tag, we can choose to access only certain properties. These properties include its name, attributes and strings (if applicable).

A tag's name is found using `.name`.

```
>>> tag = soup.a
>>> tag.name
u'a'
```

The attributes of a tag, if it has them, are stored in a dictionary and can be accessed as such. Accessing all the tags at once can be done through `.attrs`. Individual tags values can be reached by calling the key associated with it. If we try to access a key that is not an attribute, we get a `KeyError` in return.

```
>>> tag.attrs
{u'class': [u'stooge'], u'href': u'http://example.com/larry', u'id': u'link1'}

>>> tag['class']
[u'stooge']

>>> tag['href']
u'http://example.com/larry'

>>> tag['id']
u'link1'
```

Note that `class` returns a list. This is because the `class` attribute can have more than one value. This may show up in some HTML trees, but is not very common.

If a tag contains any text, it can accessed with `.string`.

```
>>> tag.string
u'Larry'
```

Problem 3. Using what you have just learned, write a function that returns the following from the Three Stooges example:

```
[u'title']
```

(Hint: This is the attribute in the eighth line of the prettified string from the previous problem.)

By Family Relations

Once we have selected a tag, we have several options available to navigate up, down, and sideways through an HTML tree.

Going Down

As mentioned before, a tag may contain other nested tags or text. These are called its children. Calling `.contents` returns the children of the parent tag in a list.

```
>>> head_tag = soup.head
>>> head_tag
<head><title>The Three Stooges</title></head>

>>> head_tag.contents
[<title>The Three Stooges</title>]

>>> title_tag = head_tag.contents[0]
>>> title_tag
<title>The Three Stooges</title>

>>> title_tag.contents
[u'The Three Stooges']
```

Note that the child of `title_tag` is the string `'The Three Stooges'`. Since strings cannot cannot have children, calling `.contents` on this string will return an error.

NOTE

One thing to note is the following:

```
>>> children_doc = """
...    <html><head>The Three Little Pigs</head>
...    <body>
...    <p>The first little piggy</p>
...    <p>The second little piggy</p>
...    <p>The third little piggy</p>
...    </body>
...    </html>"""
>>> pig_soup = BeautifulSoup(children_doc, 'html.parser')
>>> pig_soup.body.contents
[u'\n',
```

```
<p>The first little piggy</p>,
u'\n',
<p>The second little piggy</p>,
u'\n',
<p>The third little piggy</p>,
u'\n']
```

In this example, each new line character in the `<body>` tag is counted as a child of `<body>`. This will be very important when trying to navigate between *siblings*, or children of a common tag.

In addition to creating a list of children with `.contents`, we can use `.children` to create a generator of children tags. Using the previous example, we get the following (remember the extra carriage returns):

```
>>> for pig in pig_soup.body.children:
...     print(repr(pig)) #use repr() to ignore escape sequences
u'\n'
<p>The first little piggy</p>
u'\'n
<p>The second little piggy</p>
u'\n'
<p>The third little piggy</p>
u'\n'
```

There is a `.descendants` attribute which will recursively go through a tag's children, then the children's children, etc. It is left to the student to look at the online documentation for this attribute.

If a tag has only one child, and that child is a string, the child is available using `.string`. If a tag has one tag, and that tag has a single string as a child, then the parent tag can use `.string` to access the string as well.

```
>>> head_tag = soup.head
>>> print(head_tag)
<head><title>The Three Stooges</head></title>
>>> title_tag = head_tag.contents[0]
>>> print(title_tag)
<title>The Three Stooges</title>
>>>head_tag.string
u'The Three Stooges'
>>>title_tag.string
u'The Three Stooges'
```

If a tag contains more than one string, `.string` return `None`. However, `.strings` returns a generator that iterates through all strings contained within a tag. Check the online documentation for examples.

Going Up

Just as tags can have *childern*, tags can also have a *parent*. To access a tag's parent, we use the `.parent` attribute.

```
>>> title_tag = soup.title
>>> title_tag
<title>The Three Stooges</title>
>>> title_tag.parent
<head><title>The Three Stooges</title></head>
```

The parent of a string is the tag that contains it.

```
>>> tag = title_tag.string
>>> print(tag)
The Three Stooges

>>> tag.parent
<title>The Three Stooges</title>
```

Calling `.parents` iterates through all parents of a given tag. Examples can be found in the online documentation.

Going Sideways

Consider the following document, taken from the online documentation:

```
>>> sibling_soup = BeautifulSoup("<a><b>text 1</b><c>text 2</c></a>")
>>> print(sibling_soup.prettify())
<html>
 <body>
  <a>
   <b>
    text 1
   </b>
   <c>
    text 2
   </c>
  </a>
 </body>
</html>
```

Note the `<b>` and `<c>` tags are on the same level, underneath the `<a>` tag. These tags are considered *siblings*. Siblings in an HTML tree will always appear with the same indentation underneath a parent tag.

We use the attributes `.next_sibling` and `.previous_sibling` to navigate between these sibling elements. If a sibling has no next or previous sibling, calling these attributes returns `None`.

```
>>> sibling_soup.b
<b>text 1</b>

>>> sibling_soup.b.next_sibling
<c>text 2</c>

>>> sibling_soup.c.previous_sibling
<b>text 1</b>

>>> sibling_soup.c.next_sibling #<c> has no next sibling
None
```

```
>>> sibling_soup.b.string
u'text 1'

>>> print(sibling_soup.b.string.next_sibling) #text 1 and text 2 are not siblings
None
```

Recall that in the `pig_soup` example we saw extra carriage returns between the `<p>` tags.

```
>>> pig_soup.body.contents
[u'\n',
 <p>The first little piggy</p>,
 u'\n',
 <p>The second little piggy</p>,
 u'\n',
 <p>The third little piggy</p>,
 u'\n']
```

What do you expect `pig_soup.body.p.next_sibling` to return?

```
>>> pig_soup.body.p.next_sibling
u'\n'

>>> pig_soup.body.p.next_sibling.next_sibling
<p>The second little piggy</p>
```

We need to make two calls to `.next_sibling` in order to get the next `<p>` tag. Keep this in mind for future questions as you navigate across siblings.

Just as with parents and children, there are also sibling generators `.next_siblings` and `.previous_siblings` to iterate through all the siblings of a given tag. These generators can be useful when multiple calls to `.next_sibling` must be made. As before, check the online documentation for more information.

Problem 4. Using the Three Stooges example and navigation by family relations, write a function that returns the following:

```
u'Mo'
```

Problem 5. Download the 'example.htm' file associated with the lab into your working directory (you can go to `http://example.com` to see the site this originates from). The following code opens and loads a file into a `BeautifulSoup` object:

```
>>> example_soup = BeautifulSoup(open('example.htm'), 'html.parser')
```

Write a function that returns the following line using two different methods.

```
u'More information...'
```

Have the function accept an integer `method`. If `method` is 1, return the line using your first method. If `method` is 2, return the line using the other.

By `find()`

In actual website HTML, often there are many tags that share a common name. It would be nice to find characteristics that might be unique for a given tag. Look back at our previous examples and think about what characteristics differentiate tags with the same name. BeautifulSoup uses `.find()` to allow you to search for a tag not only by name, but also by a specific attribute value or strings. The following examples refer back to the "Three Stooges" HTML document in problem 2.

Search by name:

```
>>> soup.find('b') #Pass in tag names, just like soup.b
<b>The Three Stooges</b>

>>> #or use the name parameter
>>> soup.find(name='a') #Still only returns the first instance
<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>
```

Search by attribute:

```
>>> soup.find(id='link3') #Search by unique id attribute
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>

>>> #class is a Python keyword. Use 'class_' for the attribute key.
>>> soup.find(class_='title')
<p class="title"><b>The Three Stooges</b></p>

>>> #use the attrs parameter
>>> soup.find(attrs={'id':'link3'})
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>

>>> #combine attributes
>>> soup.find(attrs={'class':'stooge', 'href':'http://example.com/curly'})
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>

>>> soup.find(class_='stooge', href='http://example.com/curly')
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

Search by string:

```
>>> soup.find(string='Mo') #Recall strings act as individual units
u'Mo'

>>> soup.find(string='Mo').parent #access the tag through the parent
<a class="stooge" href="http://example.com/mo" id="link2">Mo</a>
```

Search by combining parameters:

```
>>> soup.find('a', attrs={'id':'link2','class':'stooge'})
<a class="stooge" href="http://example.com/mo" id="link2">Mo</a>
```

Problem 6. Refer to the `example.htm` file. Load it using BeautifulSoup. Write a function that returns the tag associated with the "More information..." link using two different methods. At least one of these methods should use the `.find()` function. As before, have the function accept an integer `method`. If `method` is 1, use the first method. If `method` is 2, use the second method.

Problem 7. Download 'SanDiegoWeather.htm' and load it into BeautifulSoup. You can find the corresponding website at `http://www.wunderground.com/history/airport/KSAN/2015/1/1/DailyHistory.html?req_city=San+Diego&req_state=CA&req_statename=California&reqdb.zip=92101&reqdb.magic=1&reqdb.wmo=99999&MR=1`. Using the `.find()` method, write a function that prints the tags referred to in the following questions:

1. What is the tag which contains the date, Thursday, January 1, 2015?
2. What are the tags which contain the links 'Previous Day' and 'Next Day'?
3. What is the tag which contains the number associated with the Actual Max Temperature?

(Hint: You can do a `ctrl+f` to find where the text is in the HTML, then study the tags around it.)

By `find_all()`

Recall that when a tag appeared multiple times, calling that tag name would return the first tag of that name.

```
>>> soup.a
<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>
```

To get all instances of a certain tag, use the `find_all()` command.

```
>>> soup.find_all('a')
[<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>,
 <a class="stooge" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="stooge" href="http://example.com/curly" id="link3">Curly</a>]
```

This works with all the same arguments as the `.find()` function. You may refer to the online documentation for explicit examples.

Advanced Techniques

The following examples are techniques that can aid you in your search for specific tags. Consider the "Three Stooges" example from before. Suppose you want to find

the tag that includes the url `http://example.com/curly`. You could search for it using the following:

```
>>> soup.find(href='http://example.com/curly)
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

This could be annoying to type out the whole website. Instead you could use regular expressions as follows:

```
>>> import re
>>> soup.find(href=re.compile('curly')) #find href containing 'curly' in it.
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

This method can be used for tag names, attributes, and strings as well.

```
>>> soup.find(string=re.compile('Cu')).parent #find tag with string that starts ↩
    with 'Cu'.
<a class="stooge" href="http://example.com/curly" id="link3">Curly</a>
```

To find a tag that has an attribute with a value, regardless of what the value is, we can use `True` and `False` in place of actual values. The following returns all tags that have some value associated with their `href` attribute:

```
>>> soup.find_all(href=True)
[<a class="stooge" href="http://example.com/larry" id="link1">Larry</a>,
 <a class="stooge" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="stooge" href="http://example.com/curly" id="link3">Curly</a>]
```

Problem 8. Use BeautifulSoup to load the 'Big Data dates' file. This page can be found at `https://www.federalreserve.gov/releases/lbr/`. Note that the actual website may include more dates than the file provided. Notice all the release dates of bank data, ranging from 2003 to 2014 in the file you downloaded. Using `find_all()` and `re`, find all the links to bank data from September 30, 2003 to December 31, 2014. Write a function that loads the file into BeautifulSoup and returns a list of all of the tags containing these links.