**Object Types**

BeautifulSoup breaks the HTML page into several types of objects.

**Tags**

A Tag corresponds to an HTML Tag in the original document. These lines of code:

soup = BeautifulSoup('<div id="example">An example div</div><p>An example p tag</p>')

print(soup.div)

Would produce output that looks like:

<div id="example">An example div</div>

Accessing a tag from the BeautifulSoup object in this way will get the *first* tag of that type on the page.

You can get the name of the tag using .name and a dictionary representing the attributes of the tag using .attrs:

print(soup.div.name)

print(soup.div.attrs)

div

{'id': 'example'}

**NavigableStrings**

NavigableStrings are the pieces of text that are in the HTML tags on the page. You can get the string inside of the tag by calling .string:

print(soup.div.string)

An example div

# Navigating by Tags

To navigate through a tree, we can call the tag names themselves. Imagine we have an HTML page that looks like this:

<h1>World's Best Chocolate Chip Cookies</h1>

<div class="banner">

<h1>Ingredients</h1>

</div>

<ul>

<li> 1 cup flour </li>

<li> 1/2 cup sugar </li>

<li> 2 tbsp oil </li>

<li> 1/2 tsp baking soda </li>

<li> ½ cup chocolate chips </li>

<li> 1/2 tsp vanilla <li>

<li> 2 tbsp milk </li>

</ul>

If we made a soup object out of this HTML page, we have seen that we can get the first h1 element by calling:

print(soup.h1)

<h1>World's Best Chocolate Chip Cookies</h1>

We can get the children of a tag by accessing the .children attribute:

for child in soup.ul.children:

print(child)

<li> 1 cup flour </li>

<li> 1/2 cup sugar </li>

<li> 2 tbsp oil </li>

<li> 1/2 tsp baking soda </li>

<li> ½ cup chocolate chips </li>

<li> 1/2 tsp vanilla <li>

<li> 2 tbsp milk </li>

We can also navigate up the tree of a tag by accessing the .parents attribute:

for parent in soup.li.parents:

print(parent)

This loop will first print:

<ul>

<li> 1 cup flour </li>

<li> 1/2 cup sugar </li>

<li> 2 tbsp oil </li>

<li> 1/2 tsp baking soda </li>

<li> ½ cup chocolate chips </li>

<li> 1/2 tsp vanilla </li>

<li> 2 tbsp milk </li>

</ul>

Then, it will print the tag that contains the ul (so, the body tag of the document). Then, it will print the tag that contains the body tag (so, the html tag of the document).

# Find All

If we want to find all of the occurrences of a tag, instead of just the first one, we can use .find\_all().

This function can take in just the name of a tag and returns a list of all occurrences of that tag.

print(soup.find\_all("h1"))

['<h1>World's Best Chocolate Chip Cookies</h1>', '<h1>Ingredients</h1>']

.find\_all() is far more flexible than just accessing elements directly through the soup object. With .find\_all(), we can use regexes, attributes, or even functions to select HTML elements more intelligently.

#### Using Regex

What if we want every <ol> and every <ul> that the page contains? We can select both of these types of elements with a regex in our .find\_all():

import re

soup.find\_all(re.compile("[ou]l"))

What if we want all of the h1 - h9 tags that the page contains? Regex to the rescue again!

import re

soup.find\_all(re.compile("h[1-9]"))

#### Using Lists

We can also just specify all of the elements we want to find by supplying the function with a list of the tag names we are looking for:

soup.find\_all(['h1', 'a', 'p'])

#### Using Attributes

We can also try to match the elements with relevant attributes. We can pass a dictionary to the attrs parameter of find\_all with the desired attributes of the elements we’re looking for. If we want to find all of the elements with the "banner" class, for example, we could use the command:

soup.find\_all(attrs={'class':'banner'})

Or, we can specify multiple different attributes! What if we wanted a tag with a "banner" class and the id "jumbotron"?

soup.find\_all(attrs={'class':'banner', 'id':'jumbotron'})

#### Using A Function

If our selection starts to get really complicated, we can separate out all of the logic that we’re using to choose a tag into its own function. Then, we can pass that function into .find\_all()!

def has\_banner\_class\_and\_hello\_world(tag):

return tag.attr('class') == "banner" and tag.string == "Hello world"

soup.find\_all(has\_banner\_class\_and\_hello\_world)

This command would find an element that looks like this:

<div class="banner">Hello world</div>

but not an element that looks like this:

<div>Hello world</div>

Or this:

<div class="banner">What's up, world!</div>

# Select for CSS Selectors

Another way to capture your desired elements with the soup object is to use CSS selectors. The .select() method will take in all of the CSS selectors you normally use in a .css file!

<h1 class='results'>Search Results for: <span class='searchTerm'>Funfetti</span></h1>

<div class='recipeLink'><a href="spaghetti.html">Funfetti Spaghetti</a></div>

<div class='recipeLink' id="selected"><a href="lasagna.html">Lasagna de Funfetti</a></div>

<div class='recipeLink'><a href="cupcakes.html">Funfetti Cupcakes</a></div>

<div class='recipeLink'><a href="pie.html">Pecan Funfetti Pie</a></div>

If we wanted to select all of the elements that have the class 'recipeLink', we could use the command:

soup.select(".recipeLink")

If we wanted to select the element that has the id 'selected', we could use the command:

soup.select("#selected")

Let’s say we wanted to loop through all of the links to these funfetti recipes that we found from our search.

for link in soup.select(".recipeLink > a"):

webpage = requests.get(link)

new\_soup = BeautifulSoup(webpage)

This loop will go through each link in each .recipeLink div and create a soup object out of the webpage it links to. So, it would first make soup out of <a href="spaghetti.html">Funfetti Spaghetti</a>, then <a href="lasagna.html">Lasagna de Funfetti</a>, and so on.

# Reading Text

When we use BeautifulSoup to select HTML elements, we often want to grab the text inside of the element, so that we can analyze it. We can use .get\_text() to retrieve the text inside of whatever tag we want to call it on.

<h1 class="results">Search Results for: <span class='searchTerm'>Funfetti</span></h1>

If this is the HTML that has been used to create the soup object, we can make the call:

soup.get\_text()

Which will return:

'Search Results for: Funfetti'

Notice that this combined the text inside of the outer h1 tag with the text contained in the span tag inside of it! Using get\_text(), it looks like both of these strings are part of just one longer string. If we wanted to separate out the texts from different tags, we could specify a separator character. This command would use a . character to separate:

soup.get\_text('|')

Now, the command returns:

'Search Results for: |Funfetti'