

7

Web Scraping

Lab Objective: *Web Scraping is the process of gathering data from websites on the internet. Since almost everything rendered by an internet browser as a web page uses HTML, the first step in web scraping is being able to extract information from HTML. In this lab, we introduce the requests library for scraping web pages, and BeautifulSoup, Python's canonical tool for efficiently and cleanly navigating and parsing HTML.*

HTTP and Requests

HTTP stands for Hypertext Transfer Protocol, which is an application layer networking protocol. It is a higher level protocol than TCP, which we used to build a server in the Web Technologies lab, but uses TCP protocols to manage connections and provide network capabilities. The HTTP protocol is centered around a request and response paradigm, in which a client makes a request to a server and the server replies with a response. There are several methods, or *requests*, defined for HTTP servers, the three most common of which are GET, POST, and PUT. A GET request asks for information from the server, a POST request modifies the state of the server, and a PUT request adds new pieces of data to the server.

The standard way to get the source code of a website using Python is via the `requests` library.¹ Calling `requests.get()` sends an HTTP GET request to a specified website. The website returns a response code, which indicates whether or not the request was received, understood, and accepted. If the response code is good, typically 200², then the response will also include the website source code as an HTML file.

```
>>> import requests

# Make a request and check the result. A status code of 200 is good.
>>> response = requests.get("http://www.byu.edu")
>>> print(response.status_code, response.ok, response.reason)
200 True OK
```

¹Though `requests` is not part of the standard library, it is recognized as a standard tool in the data science community. See <http://docs.python-requests.org/>.

²See https://en.wikipedia.org/wiki/List_of_HTTP_status_codes for explanation of specific response codes.

```
# The HTML of the website is stored in the 'text' attribute.
>>> print(response.text)
<!DOCTYPE html>
<html lang="en" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/↵
  content/ dc: http://purl.org/dc/terms/ foaf: http://xmlns.com/foaf/0.1/ ↵
  og: http://ogp.me/ns# rdfs: http://www.w3.org/2000/01/rdf-schema# schema:↵
  http://schema.org/ sioc: http://rdfs.org/sioc/ns# sioc: http://rdfs.org↵
  /sioc/types# skos: http://www.w3.org/2004/02/skos/core# xsd: http://www.↵
  w3.org/2001/XMLSchema# " class=" ">

<head>
  <meta charset="utf-8" />
# ...
```

Note that some websites aren't built to handle large amounts of traffic or many repeated requests. Most are built to identify web scrapers or crawlers that initiate many consecutive GET requests without pauses, and retaliate or block them. When web scraping, always make sure to store the data that you receive in a file and include error checks to prevent retrieving the same data unnecessarily. We won't spend much time on that in this lab, but it's especially important in larger applications.

Problem 1. Use the `requests` library to get the HTML source for the website `http://www.example.com`. Save the source as a file called `example.html`. If the file already exists, make sure not to scrape the website or overwrite the file. You will use this file later in the lab.

ACHTUNG!

Scraping copyrighted information without the consent of the copyright owner can have severe legal consequences. Many websites, in their terms and conditions, prohibit scraping parts or all of the site. Websites that do allow scraping usually have a file called `robots.txt` (for example, `www.google.com/robots.txt`) that specifies which parts of the website are off-limits and how often requests can be made according to the *robots exclusion standard*.^a

Be careful and considerate when doing any sort of scraping, and take care when writing and testing code to avoid unintended behavior. It is up to the programmer to create a scraper that respects the rules found in the terms and conditions and in `robots.txt`.^b

We will cover this more in the next lab.

^aSee www.robotstxt.org/orig.html and en.wikipedia.org/wiki/Robots_exclusion_standard.

^bPython provides a parsing library called `urllib.robotparser` for reading `robot.txt` files. For more information, see <https://docs.python.org/3/library/urllib.robotparser.html>.

HTML

Hyper Text Markup Language, or *HTML*, is the standard *markup language*—a language designed for the processing, definition, and presentation of text—for creating webpages. It structures a document

using pairs of *tags* that surround and define content. Opening tags have a tag name surrounded by angle brackets (`<tag-name>`). The companion closing tag looks the same, but with a forward slash before the tag name (`</tag-name>`). A list of all current HTML tags can be found at <http://htmldog.com/reference/htmltags>.

Most tags can be combined with *attributes* to include more data about the content, help identify individual tags, and make navigating the document much simpler. In the following example, the `<a>` tag has `id` and `href` attributes.

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

In HTML, `href` stands for *hypertext reference*, a link to another website. Thus the above example would be rendered by a browser as a single line of text, with `here` being a clickable link to <http://www.example.com>:

Click here for more information.

Unlike Python, HTML does not enforce indentation (or any whitespace rules), though indentation generally makes HTML more readable. The previous example can be written in a single line.

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

Special tags, which don't contain any text or other tags, are written without a closing tag and in a single pair of brackets. A forward slash is included between the name and the closing bracket. Examples of these include `<hr/>`, which describes a horizontal line, and ``, the tag for representing an image.

NOTE

You can open `.html` files using a text editor or any web browser. In a browser, you can inspect the source code associated with specific elements. Right click the element and select **Inspect**. If you are using Safari, you may first need to enable “Show Develop menu” in “Preferences” under the “Advanced” tab.

BeautifulSoup

BeautifulSoup (`bs4`) is a package³ that makes it simple to navigate and extract data from HTML documents. See <http://www.crummy.com/software/BeautifulSoup/bs4/doc/index.html> for the full documentation.

³BeautifulSoup is not part of the standard library; install it with `conda install beautifulsoup4` or with `pip install beautifulsoup4`.

The `bs4.BeautifulSoup` class accepts two parameters to its constructor: a string of HTML code and an HTML parser to use under the hood. The HTML parser is technically a keyword argument, but the constructor prints a warning if one is not specified. The standard choice for the parser is `"html.parser"`, which means the object uses the standard library's `html.parser` module as the engine behind the scenes.

NOTE

Depending on project demands, a parser other than `"html.parser"` may be useful. A couple of other options are `"lxml"`, an extremely fast parser written in C, and `"html5lib"`, a slower parser that treats HTML in much the same way a web browser does, allowing for irregularities. Both must be installed independently; see <https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser> for more information.

A `BeautifulSoup` object represents an HTML document as a tree. In the tree, each tag is a *node* with nested tags and strings as its *children*. The `prettify()` method returns a string that can be printed to represent the `BeautifulSoup` object in a readable format that reflects the tree structure.

```
>>> from bs4 import BeautifulSoup

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

>>> small_soup = BeautifulSoup(small_example_html, 'html.parser')
>>> print(small_soup.prettify())
<html>
  <body>
    <p>
      Click
      <a href="http://www.example.com" id="info">
        here
      </a>
      for more information.
    </p>
  </body>
</html>
```

Each tag in a `BeautifulSoup` object's HTML code is stored as a `bs4.element.Tag` object, with actual text stored as a `bs4.element.NavigableString` object. Tags are accessible directly through the `BeautifulSoup` object.

```
# Get the <p> tag (and everything inside of it).
>>> small_soup.p
```

```

<p>
  Click <a href="http://www.example.com" id="info">here</a>
  for more information.
</p>

# Get the <a> sub-tag of the <p> tag.
>>> a_tag = small_soup.p.a
>>> print(a_tag, type(a_tag), sep='\n')
<a href="http://www.example.com" id="info">here</a>
<class 'bs4.element.Tag'>

# Get just the name, attributes, and text of the <a> tag.
>>> print(a_tag.name, a_tag.attrs, a_tag.string, sep="\n")
a
{'id': 'info', 'href': 'http://www.example.com'}
here

```

Attribute	Description
<code>name</code>	The name of the tag
<code>attrs</code>	A dictionary of the attributes
<code>string</code>	The single string contained in the tag
<code>strings</code>	Generator for strings of children tags
<code>stripped_strings</code>	Generator for strings of children tags, stripping whitespace
<code>text</code>	Concatenation of strings from all children tags

Table 7.1: Data attributes of the `bs4.element.Tag` class.

Problem 2. The `BeautifulSoup` class has a `find_all()` method that, when called with `True` as the only argument, returns a list of all tags in the HTML source code.

Write a function that accepts a string of HTML code as an argument. Use `BeautifulSoup` to return a list of the **names** of the tags in the code.

Navigating the Tree Structure

Not all tags are easily accessible from a `BeautifulSoup` object. Consider the following example.

```

>>> pig_html = """
<html><head><title>Three Little Pigs</title></head>
<body>
<p class="title"><b>The Three Little Pigs</b></p>
<p class="story">Once upon a time, there were three little pigs named
<a href="http://example.com/larry" class="pig" id="link1">Larry,</a>
<a href="http://example.com/mo" class="pig" id="link2">Mo</a>, and
<a href="http://example.com/curly" class="pig" id="link3">Curly.</a>
<p>The three pigs had an odd fascination with experimental construction.</p>

```

```

<p>...</p>
</body></html>
"""

>>> pig_soup = BeautifulSoup(pig_html, "html.parser")
>>> pig_soup.p
<p class="title"><b>The Three Little Pigs</b></p>

>>> pig_soup.a
<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>

```

Since the HTML in this example has several `<p>` and `<a>` tags, only the **first** tag of each name is accessible directly from `pig_soup`. The other tags can be accessed by manually navigating through the HTML tree.

Every HTML tag (except for the topmost tag, which is usually `<html>`) has a *parent* tag. Each tag also has zero or more *sibling* and *children* tags or text. Following a true tree structure, every `bs4.element.Tag` in a soup has multiple attributes for accessing or iterating through parent, sibling, or child tags.

Attribute	Description
<code>parent</code>	The parent tag
<code>parents</code>	Generator for the parent tags up to the top level
<code>next_sibling</code>	The tag immediately after to the current tag
<code>next_siblings</code>	Generator for sibling tags after the current tag
<code>previous_sibling</code>	The tag immediately before the current tag
<code>previous_siblings</code>	Generator for sibling tags before the current tag
<code>contents</code>	A list of the immediate children tags
<code>children</code>	Generator for immediate children tags
<code>descendants</code>	Generator for all children tags (recursively)

Table 7.2: Navigation attributes of the `bs4.element.Tag` class.

```

# Start at the first <a> tag in the soup.
>>> a_tag = pig_soup.a
>>> a_tag
<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>

# Get the names of all of <a>'s parent tags, traveling up to the top.
# The name '[document]' means it is the top of the HTML code.
>>> [par.name for par in a_tag.parents]      # <a>'s parent is <p>, whose
['p', 'body', 'html', '[document]']        # parent is <body>, and so on.

# Get the next siblings of <a>.
>>> a_tag.next_sibling
'\n'                                         # The first sibling is just text.
>>> a_tag.next_sibling.next_sibling          # The second sibling is a tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>

```

Note carefully that newline characters are considered to be children of a parent tag. Therefore iterating through children or siblings often requires checking which entries are tags and which are just text. In the next example, we use a tag's `attrs` attribute to access specific attributes within the tag (see Table 7.1).

```
# Get to the <p> tag that has class="story" using these commands.
>>> p_tag = pig_soup.body.p.next_sibling.next_sibling
>>> p_tag.attrs["class"]           # Make sure it's the right tag.
['story']

# Iterate through the child tags of <p> and print hrefs whenever they exist.
>>> for child in p_tag.children:
...     # Skip the children that are not bs4.element.Tag objects
...     # These don't have the attribute "attrs"
...     if hasattr(child, "attrs") and "href" in child.attrs:
...         print(child.attrs["href"])
http://example.com/larry
http://example.com/mo
http://example.com/curly
```

Note that the `"class"` attribute of the `<p>` tag is a list. This is because the `"class"` attribute can take on several values at once; for example, the tag `<p class="story book">` is of class `'story'` and of class `'book'`.

The behavior of the `string` attribute of a `bs4.element.Tag` object depends on the structure of the corresponding HTML tag.

1. If the tag has a string of text and no other child elements, then `string` is just that text.
2. If the tag has exactly one child tag and the child tag has only a string of text, then the tag has the same `string` as its child tag.
3. If the tag has more than one child, then `string` is `None`. In this case, use `strings` to iterate through the child strings. Alternatively, the `get_text()` method returns all text belonging to a tag and to all of its descendants. In other words, it returns anything inside a tag that isn't another tag.

```
>>> pig_soup.head
<head><title>Three Little Pigs</title></head>

# Case 1: the <title> tag's only child is a string.
>>> pig_soup.head.title.string
'Three Little Pigs'

# Case 2: The <head> tag's only child is the <title> tag.
>>> pig_soup.head.string
'Three Little Pigs'

# Case 3: the <body> tag has several children.
>>> pig_soup.body.string is None
```

```
True
>>> print(pig_soup.body.get_text().strip())
The Three Little Pigs
Once upon a time, there were three little pigs named
Larry,
Mo, and
Curly.
The three pigs had an odd fascination with experimental construction.
...
```

Problem 3. Write a function that reads a file of the same format as the output from Problem 1 and loads it into BeautifulSoup. Find the first `<a>` tag, and return its text along with a boolean value indicating whether or not it has a hyperlink (`href` attribute).

Searching for Tags

Navigating the HTML tree manually can be helpful for gathering data out of lists or tables, but these kinds of structures are usually buried deep in the tree. The `find()` and `find_all()` methods of the `BeautifulSoup` class identify tags that have distinctive characteristics, making it much easier to jump straight to a desired location in the HTML code. The `find()` method only returns the **first** tag that matches a given criteria, while `find_all()` returns a list of all matching tags. Tags can be matched by name, attributes, and/or text.

```
# Find the first <b> tag in the soup.
>>> pig_soup.find(name='b')
<b>The Three Little Pigs</b>

# Find all tags with a class attribute of 'pig'.
# Since 'class' is a Python keyword, use 'class_' as the argument.
>>> pig_soup.find_all(class_='pig')
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
  <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
  <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find the first tag that matches several attributes.
>>> pig_soup.find(attrs={"class": "pig", "href": "http://example.com/mo"})
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>

# Find the first tag whose text is 'Mo'.
>>> pig_soup.find(string='Mo')
'Mo'
# The result is the actual string,
>>> pig_soup.find(string='Mo').parent # so go up one level to get the tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>
```

Problem 4. The file `san_diego_weather.html` contains the HTML source for an old page from Weather Underground.^a Write a function that reads the file and loads it into BeautifulSoup.

Return a list of the following tags:

1. The tag containing the date “Thursday, January 1, 2015”.
2. The tags which contain the **links** “Previous Day” and “Next Day.”
3. The tag which contains the number associated with the Actual Max Temperature.

^aSee 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

Advanced Search Techniques: Regular Expressions

Consider the problem of finding the tag that is a link to the URL `http://example.com/curly`.

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

This approach works, but it requires entering in the entire URL. To perform generalized searches, the `find()` and `find_all()` method also accept compiled regular expressions from the `re` module. This way, the methods locate tags whose name, attributes, and/or string matches a pattern.

```
>>> import re

# Find the first tag with an href attribute containing 'curly'.
>>> pig_soup.find(href=re.compile(r"curly"))
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>

# Find the first tag with a string that starts with 'Cu'.
>>> pig_soup.find(string=re.compile(r"~Cu")).parent
<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>

# Find all tags with text containing 'Three'.
>>> [tag.parent for tag in pig_soup.find_all(string=re.compile(r"Three"))]
[<title>Three Little Pigs</title>, <b>The Three Little Pigs</b>]
```

Finally, to find a tag that has a particular attribute, regardless of the actual value of the attribute, use `True` in place of search values.

```
# Find all tags with an 'id' attribute.
>>> pig_soup.find_all(id=True)
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]
```

```
# Find the names all tags WITHOUT an 'id' attribute.
>>> [tag.name for tag in pig_soup.find_all(id=False)]
['html', 'head', 'title', 'body', 'p', 'b', 'p', 'p', 'p']
```

Advanced Search Techniques: CSS Selectors

BeautifulSoup also supports the use of CSS selectors. CSS (Cascading Style Sheet) describes the style and layout of a webpage, and CSS selectors provide a useful way to navigate HTML code. Use the method `soup.select()` to find all elements matching an argument. The general format for an argument is `tag-name[attribute-name = 'attribute value']`. The table below lists symbols you can use to more precisely locate various elements.

Symbol	Meaning
=	Matches an attribute value exactly
*=	Partially matches an attribute value
^=	Matches the beginning of an attribute value
\$=	Matches the end of an attribute value
+	Next sibling of matching element
>	Search an element's children

Table 7.3: CSS symbols for use with Selenium

You can do many other useful things with CSS selectors. A helpful guide can be found at https://www.w3schools.com/cssref/css_selectors.asp. The code below gives an example using arguments described above.

```
# Find all <a> tags with id="link1"
>>> pig_soup.select("[id='link1']")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>]

# Find all tags with an href attribute containing 'curly'.
>>> pig_soup.select("[href*='curly']")
[<a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find all <a> tags with an href attribute
>>> pig_soup.select("a[href]")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find all <b> tags within a <p> tag with class='title'
>>> pig_soup.select("p[class='title'] b")
[<b>The Three Little Pigs</b>]

# Use a comma to find elements matching one of two arguments
>>> pig_soup.select("a[href$='mo'],[id='link3']")
[<a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]
```

Problem 5. The file `large_banks_index.html` is an index of data about large banks, as recorded by the Federal Reserve.^a Write a function that reads the file and loads the source into BeautifulSoup. Return a list of the tags containing the links to bank data from September 30, 2003 to December 31, 2014, where the dates are in reverse chronological order.

^aSee <https://www.federalreserve.gov/releases/lbr/>.

Problem 6. The file `large_banks_data.html` is one of the pages from the index in Problem 5.^a Write a function that reads the file and loads the source into BeautifulSoup. Create a single figure with two subplots:

1. A sorted bar chart of the seven banks with the most domestic branches.
2. A sorted bar chart of the seven banks with the most foreign branches.

In the case of a tie, sort the banks alphabetically by name.

^aSee <http://www.federalreserve.gov/releases/lbr/20030930/default.htm>.