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. GET requests request information from the server, POST requests modify the state of the server, and PUT requests add 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. 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 200, 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
```

1 Though 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/.
The HTML of the website is stored in the 'text' attribute.

```python
>>> print(response.text)
<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:
/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. This is 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](https://www.robotstxt.org/orig.html). 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`.

---

**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>
  </body>
</html> <!-- Closing tags -->
```

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 <img />, the tag for representing an image.

Problem 2. Using the output from Problem 1, examine the HTML source code for http://www.example.com. What tags are used? What is the value of the type attribute associated with the style tag?

Write a function that returns the set of names of tags used in the website, and the value of the type attribute of the style tag (as a string).
(Hint: there are ten unique tag names.)

BeautifulSoup

BeautifulSoup (bs4) is a package 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.

2BeautifulSoup 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.

```python
>>> 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.

```python
# Get the <p> tag (and everything inside of it).
>>> small_soup.p
```
Click `<a href="http://www.example.com" id="info">here</a>` for more information.

# Get the `<a>` sub-tag of the `<p>` tag.
```python
>>> a_tag = small_soup.p.a
>>> print(a_tag, type(a_tag), sep='
')
<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.
```python
>>> print(a_tag.name, a_tag.attrs, a_tag.string, sep="\n")
a {'id': 'info', 'href': 'http://www.example.com'}
here
```

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

Table 1.1: Data attributes of the `bs4.element.Tag` class.

**Problem 3.** 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. Use your function and the source code from `http://www.example.com` (use the output from Problem 1) to check your answers from **Problem 2**.

Navigating the Tree Structure

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

```python
>>> pig_html = ""
<html><head><title>Three Little Pigs</title></head>
<body>
<p class="title">The Three Little Pigs</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>
</body>
```

```c
<html><head><title>Three Little Pigs</title></head>
<body>
<p class="title">The Three Little Pigs</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>.
```
The three pigs had an odd fascination with experimental construction.

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

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 `siblings` 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.

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

Table 1.2: Navigation attributes of the `bs4.element.Tag` class.

```python
# 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>
```

```python
# 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.
```

```python
# Get the next siblings of <a>.
>>> a_tag.next_sibling
'\n'
```

```python
>>> a_tag.next_sibling.next_sibling  # The first sibling is just text.
```

```python
>>> a_tag.next_sibling.next_sibling  # The second sibling is a tag.
```

```python
<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.

```
# Get to the <p> tag that has class="story".
>>> 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:
...     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.

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

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

# Case 3: the `<body>` tag has several children.
```python
>>> 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 4.
Using the output from Problem 1, write a function that reads the file and loads
the code into BeautifulSoup. Find the only `<a>` tag with a hyperlink, and return its text.

---

**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.

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

# Find all tags with a class attribute of 'pig'.
```python
# 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.
```python
>>> pig_soup.find(attributes=dict(class_="pig", href="http://example.com/mo"))
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>
```
Problem 5. The file `san_diego_weather.html` contains the HTML source for an old page from Weather Underground. Write a function that reads the file and loads it into Beautiful-Soup. 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.

This HTML tree is significantly larger than the previous examples. To get started, consider opening the file in a web browser. Find the element that you are searching for on the page, right click it, and select Inspect. This opens the HTML source at the element that the mouse clicked on.

Advanced Search Techniques

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

```python
>>> 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.

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

# Find all tags with text containing 'Three'.
```
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.

```python
# 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>]

# Final 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']
```

**Problem 6.** The file `large_banks_index.html` is an index of data about large banks, as
recorded by the Federal Reserve. 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.

---

**Problem 7.** The file `large_banks_data.html` is one of the pages from the index in Problem
6. 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.

---