Web Scraping Lab

Estimated time needed: **30** minutes

Objectives

After completing this lab you will be able to:

Table of Contents

Estimated time needed: **25 min**

For this lab, we are going to be using Python and several Python libraries. Some of these libraries might be installed in your lab environment or in SN Labs. Others may need to be installed by you. The cells below will install these libraries when executed.

In [ ]:
```
!mamba install bs4==4.10.0 -y
!pip install lxml==4.6.4
!mamba install html5lib==1.1 -y
# !pip install requests==2.26.0
```

Import the required modules and functions

In [ ]:
```
from bs4 import BeautifulSoup # this module helps in web scrapping.
import requests  # this module helps us to download a web page
```

Beautiful Soup Objects

Beautiful Soup is a Python library for pulling data out of HTML and XML files, we will focus on HTML files. This is accomplished by representing the HTML as a set of objects with methods used to parse the HTML. We can navigate the HTML as a tree and/or filter out what we are looking for.

Consider the following HTML:

In [ ]:
```
%%html
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>
<h3><b id='boldest'>Lebron James</b></h3>
<p> Salary: $ 92,000,000 </p>
<h3> Stephen Curry</h3>
<p> Salary: $85,000, 000 </p>
```

```
<h3> Kevin Durant </h3>
<p> Salary: $73,200, 000</p>
</body>
</html>
```

We can store it as a string in the variable HTML:

In [ ]:
```
html="<!DOCTYPE html><html><head><title>Page Title</title></head><body><h3><b id='boldest'>Lebron James</b></h3><p> Salary: $
```

To parse a document, pass it into the `BeautifulSoup` constructor, the `BeautifulSoup` object, which represents the document as a nested data structure:

In [ ]:
```
soup = BeautifulSoup(html, "html.parser")
```

First, the document is converted to Unicode, (similar to ASCII), and HTML entities are converted to Unicode characters. Beautiful Soup transforms a complex HTML document into a complex tree of Python objects. The `BeautifulSoup` object can create other types of objects. In this lab, we will cover `BeautifulSoup` and `Tag` objects that for the purposes of this lab are identical, and `NavigableString` objects.

We can use the method `prettify()` to display the HTML in the nested structure:

In [ ]:
```
print(soup.prettify())
```

Tags

Let's say we want the title of the page and the name of the top paid player we can use the `Tag`. The `Tag` object corresponds to an HTML tag in the original document, for example, the tag title.

In [ ]:
```
tag_object=soup.title
print("tag object:",tag_object)
```

we can see the tag type `bs4.element.Tag`

In [ ]:
```
print("tag object type:",type(tag_object))
```

If there is more than one `Tag` with the same name, the first element with that `Tag` name is called, this corresponds to the most paid player:

In [ ]:
```
tag_object=soup.h3
tag_object
```

Enclosed in the bold attribute `b`, it helps to use the tree representation. We can navigate down the tree using the child attribute to get the name.

Children, Parents, and Siblings

As stated above the `Tag` object is a tree of objects we can access the child of the tag or navigate down the branch as follows:

In [ ]:
```
tag_child =tag_object.b
tag_child
```

You can access the parent with the `parent`

In [ ]:
```
parent_tag=tag_child.parent
parent_tag
```

this is identical to

In [ ]:
```
tag_object
```

`tag_object` parent is the `body` element.

In [ ]:
```
tag_object.parent
```

`tag_object` sibling is the `paragraph` element

In [ ]:
```
sibling_1=tag_object.next_sibling
sibling_1
```

`sibling_2` is the `header` element which is also a sibling of both `sibling_1` and `tag_object`

In [ ]:
```
sibling_2=sibling_1.next_sibling
sibling_2
```

Exercise: `next_sibling`

Using the object `sibling_2` and the property `next_sibling` to find the salary of Stephen Curry:

In [ ]:
```

```

▶ Click here for the solution

HTML Attributes

If the tag has attributes, the tag `id="boldest"` has an attribute `id` whose value is `boldest`. You can access a tag's attributes by treating the tag like a dictionary:

In [ ]:
```
tag_child['id']
```

You can access that dictionary directly as `attrs`:

In [ ]:
```
tag_child.attrs
```

You can also work with Multi-valued attribute check out [1] for more.

We can also obtain the content if the attribute of the `tag` using the Python `get()` method.

In [ ]:
```
tag_child.get('id')
```

Navigable String

A string corresponds to a bit of text or content within a tag. Beautiful Soup uses the `NavigableString` class to contain this text. In our HTML we can obtain the name of the first player by extracting the sting of the `Tag` object `tag_child` as follows:

In [ ]:
```
tag_string=tag_child.string
tag_string
```

we can verify the type is Navigable String

In [ ]:
```
type(tag_string)
```

A NavigableString is just like a Python string or Unicode string, to be more precise. The main difference is that it also supports some `BeautifulSoup` features. We can covert it to sting object in Python:

In [ ]:
```
unicode_string = str(tag_string)
unicode_string
```

Filter

Filters allow you to find complex patterns, the simplest filter is a string. In this section we will pass a string to a different filter method and Beautiful Soup will perform a match against that exact string. Consider the following HTML of rocket launchs:

In [ ]:
```
%%html
<table>
  <tr>
    <td id='flight' >Flight No</td>
    <td>Launch site</td>
    <td>Payload mass</td>
   </tr>
  <tr>
    <td>1</td>
    <td><a href='https://en.wikipedia.org/wiki/Florida'>Florida</a></td>
    <td>300 kg</td>
  </tr>
  <tr>
    <td>2</td>
    <td><a href='https://en.wikipedia.org/wiki/Texas'>Texas</a></td>
    <td>94 kg</td>
  </tr>
  <tr>
```

```
    <td>3</td>
    <td><a href='https://en.wikipedia.org/wiki/Florida'>Florida</a> </td>
    <td>80 kg</td>
  </tr>
</table>
```

We can store it as a string in the variable `table`:

In [ ]:
```
table="<table><tr><td id='flight' >Flight No</td><td>Launch site</td><td>Payload mass</td></tr><tr><td>1</td><td><a href='http
```

In [ ]:
```
table_bs = BeautifulSoup(table, "html.parser")
```

find All

The `find_all()` method looks through a tag's descendants and retrieves all descendants that match your filters.

The Method signature for `find_all(name, attrs, recursive, string, limit, **kwargs)`

Name

When we set the `name` parameter to a tag name, the method will extract all the tags with that name and its children.

In [ ]:
```
table_rows=table_bs.find_all('tr')
table_rows
```

The result is a Python Iterable just like a list, each element is a `tag` object:

In [ ]:
```
first_row =table_rows[0]
first_row
```

The type is `tag`

In [ ]:
```
print(type(first_row))
```

we can obtain the child

In [ ]:
```
first_row.td
```

If we iterate through the list, each element corresponds to a row in the table:

In [ ]:
```
for i,row in enumerate(table_rows):
    print("row",i,"is",row)
```

As `row` is a `cell` object, we can apply the method `find_all` to it and extract table cells in the object `cells` using the tag `td`, this is all the children with the name `td`. The result is a list, each element corresponds to a cell and is a `Tag` object, we can iterate through this list as well. We can extract the content using the `string` attribute.

In [ ]:
```
for i,row in enumerate(table_rows):
    print("row",i)
    cells=row.find_all('td')
    for j,cell in enumerate(cells):
        print('colunm',j,"cell",cell)
```

If we use a list we can match against any item in that list.

In [ ]:
```
list_input=table_bs .find_all(name=["tr", "td"])
list_input
```

Attributes

If the argument is not recognized it will be turned into a filter on the tag's attributes. For example the `id` argument, Beautiful Soup will filter against each tag's `id` attribute. For example, the first `td` elements have a value of `id` of `flight`, therefore we can filter based on that `id` value.

In [ ]:
```
table_bs.find_all(id="flight")
```

We can find all the elements that have links to the Florida Wikipedia page:

In [ ]:
```
list_input=table_bs.find_all(href="https://en.wikipedia.org/wiki/Florida")
list_input
```

If we set the `href` attribute to True, regardless of what the value is, the code finds all tags with `href` value:

In [ ]:
```
table_bs.find_all(href=True)
```

There are other methods for dealing with attributes and other related methods; Check out the following link

Exercise: `find_all`

Using the logic above, find all the elements without `href` value

In [ ]:
```

```

▶ Click here for the solution

Using the soup object `soup`, find the element with the `id` attribute content set to `"boldest"`.

In [ ]:
```

```

▶ Click here for the solution

string

With string you can search for strings instead of tags, where we find all the elments with Florida:

In [ ]:
```
table_bs.find_all(string="Florida")
```

find

The `find_all()` method scans the entire document looking for results, it's if you are looking for one element you can use the `find()` method to find the first element in the document. Consider the following two table:

In [ ]:
```
%%html
<h3>Rocket Launch </h3>

<p>
<table class='rocket'>
  <tr>
    <td>Flight No</td>
    <td>Launch site</td>
    <td>Payload mass</td>
  </tr>
  <tr>
    <td>1</td>
    <td>Florida</td>
    <td>300 kg</td>
  </tr>
  <tr>
    <td>2</td>
    <td>Texas</td>
    <td>94 kg</td>
  </tr>
  <tr>
    <td>3</td>
    <td>Florida </td>
    <td>80 kg</td>
  </tr>
</table>
</p>
<p>

<h3>Pizza Party  </h3>


<table class='pizza'>
  <tr>
    <td>Pizza Place</td>
    <td>Orders</td>
```

```
    <td>Slices </td>
   </tr>
  <tr>
    <td>Domino's Pizza</td>
    <td>10</td>
    <td>100</td>
  </tr>
  <tr>
    <td>Little Caesars</td>
    <td>12</td>
    <td >144 </td>
  </tr>
  <tr>
    <td>Papa John's </td>
    <td>15 </td>
    <td>165</td>
  </tr>
```

We store the HTML as a Python string and assign `two_tables` :

In [ ]:
```
two_tables="<h3>Rocket Launch </h3><p><table class='rocket'><tr><td>Flight No</td><td>Launch site</td> <td>Payload mass</td></
```

We create a `BeautifulSoup` object `two_tables_bs`

In [ ]:
```
two_tables_bs= BeautifulSoup(two_tables, 'html.parser')
```

We can find the first table using the tag name table

In [ ]:
```
two_tables_bs.find("table")
```

We can filter on the class attribute to find the second table, but because class is a keyword in Python, we add an underscore.

In [ ]:
```
two_tables_bs.find("table",class_='pizza')
```

Downloading And Scraping The Contents Of A Web Page

We Download the contents of the web page:

In [ ]:
```
url = "http://www.ibm.com"
```

We use `get` to download the contents of the webpage in text format and store in a variable called `data`:

In [ ]:
```
data  = requests.get(url).text
```

We create a `BeautifulSoup` object using the `BeautifulSoup` constructor

In [ ]:
```
soup = BeautifulSoup(data,"html.parser")  # create a soup object using the variable 'data'
```

Scrape all links

In [ ]:
```
for link in soup.find_all('a',href=True):  # in html anchor/link is represented by the tag <a>

    print(link.get('href'))
```

Scrape all images Tags

In [ ]:
```
for link in soup.find_all('img'):# in html image is represented by the tag <img>
    print(link)
    print(link.get('src'))
```

Scrape data from HTML tables

In [ ]:
```
#The below url contains an html table with data about colors and color codes.
url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-DA0321EN-SkillsNetwork/labs/datasets/HTMLColorCo
```

Before proceeding to scrape a web site, you need to examine the contents, and the way data is organized on the website. Open the above url in your browser and check how many rows and columns are there in the color table.

In [ ]:
```
# get the contents of the webpage in text format and store in a variable called data
data  = requests.get(url).text
```

In [ ]:
```
soup = BeautifulSoup(data,"html.parser")
```

In [ ]:
```
#find a html table in the web page
table = soup.find('table') # in html table is represented by the tag <table>
```

In [ ]:
```
#Get all rows from the table
for row in table.find_all('tr'): # in html table row is represented by the tag <tr>
    # Get all columns in each row.
    cols = row.find_all('td') # in html a column is represented by the tag <td>
    color_name = cols[2].string # store the value in column 3 as color_name
    color_code = cols[3].string # store the value in column 4 as color_code
    print("{}--->{}".format(color_name,color_code))
```

Scrape data from HTML tables into a DataFrame using BeautifulSoup and Pandas

In [ ]:
```
import pandas as pd
```

In [ ]:
```
#The below url contains html tables with data about world population.
url = "https://en.wikipedia.org/wiki/World_population"
```

Before proceeding to scrape a web site, you need to examine the contents, and the way data is organized on the website. Open the above url in your browser and check the tables on the webpage.

In [ ]:
```
# get the contents of the webpage in text format and store in a variable called data
data  = requests.get(url).text
```

In [ ]:
```
soup = BeautifulSoup(data,"html.parser")
```

In [ ]:
```
#find all html tables in the web page
tables = soup.find_all('table') # in html table is represented by the tag <table>
```

In [ ]:
```
# we can see how many tables were found by checking the length of the tables list
len(tables)
```

Assume that we are looking for the `10 most densly populated countries` table, we can look through the tables list and find the right one we are look for based on the data in each table or we can search for the table name if it is in the table but this option might not always work.

In [ ]:
```
for index,table in enumerate(tables):
    if ("10 most densely populated countries" in str(table)):
        table_index = index
print(table_index)
```

See if you can locate the table name of the table, `10 most densly populated countries`, below.

In [ ]:
```
print(tables[table_index].prettify())
```

In [ ]:
```
population_data = pd.DataFrame(columns=["Rank", "Country", "Population", "Area", "Density"])

for row in tables[table_index].tbody.find_all("tr"):
    col = row.find_all("td")
    if (col != []):
        rank = col[0].text
        country = col[1].text
        population = col[2].text.strip()
        area = col[3].text.strip()
        density = col[4].text.strip()
        population_data = population_data.append({"Rank":rank, "Country":country, "Population":population, "Area":area, "Densi

population_data
```

Scrape data from HTML tables into a DataFrame using BeautifulSoup and read_html

Using the same `url`, `data`, `soup`, and `tables` object as in the last section we can use the `read_html` function to create a DataFrame.

Remember the table we need is located in `tables[table_index]`

We can now use the `pandas` function `read_html` and give it the string version of the table as well as the `flavor` which is the parsing engine `bs4`.

In [ ]:
```
pd.read_html(str(tables[5]), flavor='bs4')
```

The function `read_html` always returns a list of DataFrames so we must pick the one we want out of the list.

In [ ]:
```
population_data_read_html = pd.read_html(str(tables[5]), flavor='bs4')[0]

population_data_read_html
```

Scrape data from HTML tables into a DataFrame using read_html

We can also use the `read_html` function to directly get DataFrames from a `url`.

In [ ]:
```
dataframe_list = pd.read_html(url, flavor='bs4')
```

We can see there are 25 DataFrames just like when we used `find_all` on the `soup` object.

In [ ]:
```
len(dataframe_list)
```

Finally we can pick the DataFrame we need out of the list.

In [ ]:
```
dataframe_list[5]
```

We can also use the `match` parameter to select the specific table we want. If the table contains a string matching the text it will be read.

In [ ]:
```
pd.read_html(url, match="10 most densely populated countries", flavor='bs4')[0]
```

Authors

Ramesh Sannareddy

Other Contributors

Rav Ahuja

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2021-08-04	0.2	Made changes to markdown of nextsibling	
2020-10-17	0.1	Joseph Santarcangelo Created initial version of the lab	

In [ ]:

In [ ]: