Cours_7: Web Scraping

When performing data science tasks, it's common to want to use data found on the internet. You'll usually be able to access this data in *csv* format, or via an Application Programming Interface (API). However, there are times when the data you want can only be accessed as part of a web page. In cases like this, you'll want to use a technique called **web scraping** to get the data from the web page into a format you can work with in your analysis.

1) The components of a webpage

When we visit a web page, our web browser makes a request to a web server. This request is called a `GET` request, since we're getting files from the server. The server then sends back files that tell our browser how to print the page for us. The files fall into a few main types:

- HTML — contain the main content of the page.
- CSS — add styling to make the page look nicer.
- JS — Javascript files add interactivity to web pages.
- Images — image formats, such as JPG and PNG allow web pages to show pictures.

After our browser receives all the files, it prints the page to us. There's a lot that happens behind the scenes to print a page nicely, but we don't need to worry about most of it when we're web scraping. To perform web scraping, we're interested in the main content of the web page, so we look at the **HTML code.**

2) HTML

HyperText Markup Language **(HTML)** is a language that web pages are created in. HTML isn't a programming language, like Python — instead, it's a **markup language** that tells a browser how to layout content. HTML allows you to do similar things to what you do in a word processor like Microsoft Word — make text bold, create paragraphs, and so on. Because HTML isn't a programming language, it is way more simple then Python.

You can access to the HTML code of most webpages by doing *right-clic → view page source*. Go to the (very simple) webpage http://dataquestio.github.io/web-scraping-pages/simple.html and look at its HTML code.

Let's take a quick tour through HTML so we know enough to scrape effectively. HTML consists of elements called **tags** (**balise** en français). The most basic tag is the `<html>` tag. This tag tells the web browser that everything inside of it is HTML. We can make a simple HTML document just using this tag (at each step, compare with the code of the webpage you just went to):

```
<html>
*code*
</html>
```

Right inside an **html tag**, we put two other tags, the **head tag**, and the **body tag**. The main content of the web page goes into the body tag. The head tag contains data about the title of the page, and other informations that generally isn't useful in web scraping:

```
<html>
<head>
*text*
</head>
<body>
*text*
</body>
</html>
```

You may have noticed above that we put the *head* and *body* tags inside the *html* tag. In HTML, tags are nested, and can go inside other tags.

We'll now add our first content to the page, in the form of the p tag. The **p tag** defines a **paragraph**, and any text inside the tag is shown as a separate paragraph:

```
<html>
    <head>
    </head>
    <body>
          <p>
            Here's a paragraph of text!
        </p>
        <p>
            Here's a second paragraph of text!
        </p>
    </body>
</html>
```

This gives a webpage that looks like this:

Here’s a paragraph of text!

Here’s a second paragraph of text !

Tags have commonly used names that depend on their position in relation to other tags:

- `child` — a child is a tag inside another tag. So the two `p` tags above are both children of the `body` tag.
- `parent` — a parent is the tag another tag is inside. Above, the `html` tag is the parent of the `body` tag.
- `sibiling` — a sibiling is a tag that is nested inside the same parent as another tag. For example, `head` and `body` are siblings, since they’re both inside `html`. Both `p` tags are siblings, since they’re both inside `body`.

We can also add properties to HTML tags that change their behavior:

```
<html>
    <head>
    </head>
    <body>
        <p>
            Here's a paragraph of text!
            <a href="https://www.dataquest.io">Learn Data Science Online</a>
        </p>
        <p>
            Here's a second paragraph of text!
            <a href="https://www.python.org">Python</a>        </p>
    </body></html>
```

In the above example, we added two **`a` tags**. ***`a` tags*** are **links**, and tell the browser to render a link to another web page. The **`href`** property of the tag contains the address of the link.

`a` and `p` are extremely common html tags. Here are a few others:

- `div` — indicates a division, or area, of the page.
- `b` — bolds any text inside.
- `i` — italicizes any text inside.
- `table` — creates a table.
- `form` — creates an input form.

For a full list of tags, look here (https://developer.mozilla.org/en-US/docs/Web/HTML/Element).

Before we move into actual web scraping, let’s learn about the **`class`** and **`id`** properties. These special properties give HTML elements names, and make them **easier to interact with when we’re scraping**. One element can have multiple classes, and a class can be shared between elements. Each element can only have one id, and an id can only be used once on a page. Classes and ids are optional, and not all elements will have them.

We can add classes and ids to our example:

```
<html>
    <head>
    </head>
    <body>
        <p class="bold-paragraph">
            Here's a paragraph of text!
            <a href="https://www.dataquest.io" id="learn-link">Learn Data
Science Online</a>
        </p>
        <p class="bold-paragraph extra-large">
            Here's a second paragraph of text!
            <a href="https://www.python.org" class="extra-large">Python</a>
        </p>
    </body>
</html>
```

Adding classes and ids **does not change** how the webpage looks like, but will be useful for scrapping. We'll go back to class and id later.

3) The requests library

The first thing we'll need to do to scrape a web page is **to download the page**. We can download pages using the Python **requests library** and a **GET** request to a web server, which will download the HTML contents of a given web page for us.

Let's try downloading the simple website we've seen earlier using the ***requests.get()*** method

```
import requests
page = requests.get("http://dataquestio.github.io/web-scraping-pages/simple.html")
print page
```

<Response [200]> is what you get. This is a *Response object*. This object has a *status_code property*, which indicates if the page was downloaded successfully. A status_code of 200 means that the page downloaded successfully. We won't fully dive into status codes here, but a status code starting with a 2 generally indicates success, and a code starting with a 4 or a 5 indicates an error.

We can print out the HTML content of the page using the content property:

```
import requests
page = requests.get("http://dataquestio.github.io/web-scraping-pages/simple.html")
print page.content
```

It prints the html code source of the page, which is quite messy. This is the reason why **parsing** has been developped.

4) Parsing a page with BeautifulSoup

Parsing means *analyse syntaxique*. We use the **BeautifulSoup** library to parse the html code, and extract the text from it. We have to import the library, and create an instance of the *BeautifulSoup* class to parse our document:

```
import requests
from bs4 import BeautifulSoup
page = requests.get("http://dataquestio.github.io/web-scraping-pages/simple.html")
soup = BeautifulSoup(page.content, 'html.parser')
```

We can now print out the HTML content of the page, formatted nicely, using the **prettify method** on the BeautifulSoup object with:

```
print soup.prettify()
```

Using tags, we can move through the html code one level at a time. We can first select all the elements at the top level of the page using the **children** method **of soup**. Note that children returns a list generator, so we need to call the list function on it:

```
L = list(soup.children)
print L
```

It prints a list with three elements:
[u'html', u'\n', <html>\n<head>\n<title>A simple example page</title>\n</head>\n<body>\n Here is some simple content for this page. \n</body>\n</html>]

It tells us that there are two tags at the top level of the page — the initial `<!DOCTYPE html>` tag, and the `<html>` tag. There is a newline character (`\n`) in the list as well. Let's see what the type of each element in the list is by adding to the script:

```
print [type(item) for item in list(soup.children)]
```

It prints:
[bs4.element.Doctype, bs4.element.NavigableString, bs4.element.Tag]

As you can see, all of the items are *BeautifulSoup* objects. The first is a *Doctype* object, which contains information about the type of the document. The second is a *NavigableString*, which represents text found in the HTML document. The final item is a **Tag object**, which contains other nested tags. The most important object type, and the one we'll deal with most often, is the *Tag object*. It allows us to navigate through an HTML document, and extract other tags and text.

We can now select the html tag and its children by taking the third item in the list:

```
html = list(soup.children)[2]
print html.prettify()
```

Each item in the list returned by the children property is also a BeautifulSoup object, so we can also call the children method on *html* to find the children inside *html*:

```
print list(html.children)
```

['\n', <head> <title>A simple example page</title> </head>, '\n', <body> Here is some simple content for this page. </body>, '\n']

As you can see above, there are two tags here, *head*, and *body*. We want to extract the text inside the *p tag*, so we'll dive into the body:

```
body = list(html.children)[3]
```

Now, we can get the p tag by finding the children of the body tag:

```
list(body.children)
```

['\n', Here is some simple content for this page. , '\n']

We can now isolate the p tag:

```
p = list(body.children)[1]
```

Once we've isolated the tag, we can use the **get_text()** method to extract all of the text inside the tag:

```
p.get_text()
print p
```

5) Finding all instances of a tag at once

What we did above was useful to understand the structure of a webpage, but it took a lot of commands to do something fairly simple. If we want to extract a single tag, we can instead use the **find_all()** method, which will find all the instances of a tag on a page.

```
import requests
from bs4 import BeautifulSoup
page = requests.get("http://dataquestio.github.io/web-scraping-pages/simple.html")
soup = BeautifulSoup(page.content, 'html.parser')
for elt in soup.find_all('p'):
    print elt.get_text()
```

Note that *find_all* returns a list, so we'll have to loop through (as above), or use list indexing:

```
soup.find_all('p')[0].get_text()
```

If you only want to find the first instance of a tag, you can use the **find()** method, which will return a single BeautifulSoup object:

```
soup.find('p')
```

6) Searching for tag by class and id (Most important section)

We introduced *classes* and *ids* earlier, but it probably wasn't clear why they were useful. Classes and ids are used by CSS to determine which HTML elements to apply certain styles to (but here we don't care about that). We can also use them when scraping to specify specific elements we want to scrape. To illustrate this principle, go to this slightly more involved webpage: http://dataquestio.github.io/web-scraping-pages/ids_and_classes.html and look at its source code. Pay particular attention to class and id.

Let's first download the page and create a BeautifulSoup object:

```
import requests
from bs4 import BeautifulSoup
page = requests.get("http://dataquestio.github.io/web-scraping-pages/ids_and_classes.html")
soup = BeautifulSoup(page.content, 'html.parser')
```

Now, we can use the *find_all* method to search for **items by class or by id**. For example, we can have access to every **p tag** that has the **class outer-text**:

```
print soup.find_all('p', class_='outer-text')
```

It prints:

```
[<p class="outer-text first-item" id="second"> <b> First outer paragraph. </b> </p>, <p class="outer-text">
<b> Second outer paragraph. </b> </p>]
```

A again you can get access to text with *get.text()*:

```
for elt in soup.find_all('p', class_='outer-text'):
    print elt.get_text()
```

We also have access to every tag that has the **class outer-text**:

```
for elt in soup.find_all(class_="outer-text"):
        print elt.get_text()
```

We can also search for elements by id:

```
for elt in soup.find_all(id="first"):
        print elt.get_text()
```

7) Use CSS selector (Also very important section)

You can also search for items using **CSS selectors**. These selectors are how the CSS language allows developers to specify style of HTML tags. Here are some examples:

- p a — finds all a tags inside of a p tag.
- body p a — finds all a tags inside of a p tag inside of a body tag.
- html body — finds all body tags inside of an html tag.
- p.outer-text — finds all p tags with a class of outer-text.
- p#first — finds all p tags with an id of first.
- body p.outer-text — finds any p tags with a class of outer-text inside of a body tag.

You can learn more about CSS selectors here (https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Selectors).

BeautifulSoup objects support searching a page via CSS selectors using the **select method**. We can use CSS selectors to find all the **p tags** in our page that are inside of a **div** like this:

```
for elt in soup.select("div p"):
        print elt.get_text()
```

Note that the select method above returns a list of BeautifulSoup objects, just like **find** and **find_all**.

Exercice Moodle
Allez sur *https://moodle.di.ens.fr/* et utilisez BeautifulSoup pour trouver tous les noms de tous les cours qui utilisent Moodle.
Hint: étudier le code source de la page pour dans quelle classe sont rangés les noms des cours.

Exercice Weather
Allez sur le site **weather.com**. Choisissez votre ville favorite et ouvrez la page contenant les prévisions météo sur 5 jours. Ecrire un script qui demande à l'utilisateur un jour (entre 1 et 5, 1 étant aujourd'hui, 2 demain, etc) et qui affiche le temps qu'il fera le jour en question.