Python Notes - Web Scraping

Somenath Sinha

April 2018

Contents

Chapter 1

BS4 Concepts

1.1 Document Object Model (DOM)

The Document Object Model represents a web page in a hierarchical order where each element(i.e., *tag*) is represented by a node, and all the content within the tag is contained as children of that node.

Generally speaking, no two web pages are the exact same - even on the same site. Thus, when writing a script to extract data from a web site, different scripts have to be written for each web page. Web scrapers must be custom-built for a page on a website, and for a specific required data set.

HTML5 uses `<div>` and `<span>` to contain *block* and *in-line* data respectively. These are nestable, and thus the data that we require might reside several layers deep in `<div>`s.

1.1.1 CSS, Classes and IDs

While it's function is to control the rendering of a webpage by the browser, the tools used by CSS - namely *classes* and *IDs* are invaluable for web-scraping, since they help us to zero-in on the exact element that we want to extract data from. While an element can belong to multiple classes, and a class naturally has multiple elements, the relationship between ID and an element is mapped 1:1, i.e., IDs help uniquely identify an element.

1.2 Finding the required data

Most often, manual investigation is required before we build a web-scraper. This is because, the particular elements of interest must be identified and then leveraged to extract the data. First, we use the *Developer Tools* in web browsers such as *Mozilla Firefox* and *Google Chrome* to identify the attributes of the parent container of our data. The classes and IDs help us do this.

Dynamic content such as those generated by Javascript need to be parsed differently, and a tool such as BS4 might not be suited to the task. In such cases, the **Selenium** WebDriver can be useful.

The most ideal way to gather more information about the element containing our data is to find it in the DOM and take a look at its and its parent's attributes. We can do this is firefox by right clicking the element and choosing *inspect element*.

1.2.1 Gathering data via web-scraping

Some sites have a robots.txt file instructing scrapers about which parts they may access and how fast they can go. It's important to pay attention to these files, since otherwise we may get blocked. When this information is not present, a good strategy is to mimic *human speed*.

1.3 Introduction to BS4

BeautifulSoup4 is a web-scraper that can extract data contained within DOM elements, and even filter the DOM elements to only have to worry about the relevant ones. For this, HTTP(S) requests have to be made to web-servers, which is achieved through the *requests* package.

1.3.1 Requests Package

There is a certain procedure that must be followed while making requests to the web servers. The steps are:

- Create a session object to manage the requests.
- Control cookies and headers with requests. This step can be especially helpful if we're interesting in *mimicing human behavior* to stay undetected.
- Use the session's `get()` method to fetch the contents of the web-page.

1.3.2 Imports

To utilize the functionality of the Requests and BS4 module, we import the requests module to make HTTP(S) requests and the BeautifulSoup class for web-scraping functionality:

```python
import requests
from bs4 import BeautifulSoup
```

1.3.3 Setting the headers

Providing some data for the headers is useful to seem more *human* in cases where web-scrapers have to obtain data meant for human consumption. Before we create the header, though, we need to create a session:

```python
session = requests.Session()
```

The data requried to create the header can be obtained by visiting a site such as WhatIsMy-Browser.com which shows us all the information a site can gather from our requests through our browser. This information is generally sent as JSON data:

```
header = {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate, br",
    "Accept-Language": "en-US,en;q=0.5",
    "Cache-Control": "max-age=0",
    "Connection": "keep-alive",
    "Cookie": "WMF-Last-Access=06-Apr-2018; WMF-Last-Access-Global=06-Apr-2018; CP=H2",
    "DNT": 1,
    "Host": "en.wikipedia.org",
    "Referer": "https://www.google.co.in/",
    "Upgrade-Insecure-Requests": 1,
    "User-Agent": "Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:59.0) Gecko/20100101
     ↪  Firefox/59.0"
}
```

1.3.4 Making the request

To make the request, we need to provide the URL of the page we're going to scrape and then initiate the request:

```
url = "https://en.wikipedia.org/wiki/List_of_Nobel_laureates"
page = session.get(url, headers=header).text
```

Now when we make our request, the contents of the HTML document is sent back and stored in the `page` variable!

1.3.5 How BS4 works

The BeautifulSoup object that we create will mimic the DOM on the web-page and allow us to filter out the irrelevant stuff to only focus on what we need. We can find the required tags using the `find()` and `findAll()` methods.

The **find()** method returns the *first* object that meets the criteria, but the **findAll()** method returns a python list of all the elements that meet the criteria. For both of these methods, the first argument is the name of the tag that we want to locate. It's possible to filter out the data using a second, optional argument - a *Dictionary object* that tells the method which attributes for the target object should have what values.

1.3.6 Extracting content

Let us consider we want the data present between the starting and closing tags, such as the content of a certain paragraph:

```
<p id="test">TEXT that we're trying to extract</p>
```

The data `contents()` method. If however, the data is contained within an attribute, such as the link contained within the *href* attribute of an *a tag*, then we have to use `attrs()` method.

1.4 Working with BS4

The first step is to create the BeautifulSoup object that we'll use to parse the data:

```
nobelList = BeautifulSoup(page)
```

With the above line, the contents of the *page* variable is converted to a traversable DOM-like object - a reference to which is stored in the *nobelList* object.

1.4.1 Finding elements

To find a single link (the first one in the page), we use:

```
nobelList.find("a")
```

To find *all* links on the page:

```
nobelList.findAll("a")
```

To filter out only those elements with a specific class called *internal*, we use:

```
nobelList.findAll("a", {"class": "internal"})
```

Now, to directly get the URL of the location of the second link with the class *internal*, we use:

```
nobelList.findAll("a", {"class": "internal"})[0].attrs("href")
```

To get the contents of a certain paragraph with BS4, we use:

```
nobelList.find("p").contents
```

Finally, to match more than one class, we use a JSON array:

```
nobelList.find("a", {"class": ["wikitable", "sortable"]})
```

1.5 DOM Families

The three main relationships between nodes are:

Terms	Description
Parents	Nodes in the DOM which are above a node. They can be located using the `parent()` method.
Children	Nodes in the DOM which are sub-nodes of the current node. They can be located using the `children()` method.
Siblings	Siblings are two or more nodes that have the same parent.

A shorthand to get a list of all tags that belong to a certain category, we could use `tag('filter')`. To find all the children of a table (i.e., all the rows) we can use:

```python
trows = nobelList.table.children

# To refer to the first row:
fRow = nobelList.table.tr
```