Automation and Webscraping With Python

Charles Clayton

January 3, 2017

Course Overview

Course Overview **i**

1 Scraping with `Selenium` **1**
- 1.1 What Makes Up A Website . . . 1
  - 1.1.1 Using the Chrome Developer tools . . . 2
  - 1.1.2 HTML Tags . . . 2
  - 1.1.3 CSS Classes . . . 3
  - 1.1.4 JavaScript Queries . . . 3
  - 1.1.5 CSS Selectors . . . 4
  - 1.1.6 XPaths . . . 5
- 1.2 Using the Selenium Module . . . 7
  - 1.2.1 Setting Up A WebDriver . . . 7
  - 1.2.2 Automating the Browser . . . 7
    - 1.2.2.1 Debugging with the Console . . . 7
  - 1.2.3 Downloading Files . . . 7
  - 1.2.4 Using a Headless Browser (PhantomJS) . . . 7

2 Parsing with `BeautifulSoup 4` **7**
- 2.1 Requesting HTML . . . 7
  - 2.1.1 `urllib2` preview . . . 7
  - 2.1.2 `Requests` . . . 7
  - 2.1.3 `wget` . . . 7
- 2.2 `BeautifulSoup` Objects . . . 7
- 2.3 `BeautifulSoup` Queries . . . 7
  - 2.3.1 Comparison To JS/Selenium Queries . . . 7
  - 2.3.2 Nested Selectors . . . 7
  - 2.3.3 Regular Expressions Basics . . . 7
- 2.4 Inevitable Roadblocks . . . 7
  - 2.4.1 Server-side vs. Client-side Websites . . . 7
  - 2.4.2 UTF-8 Encoding . . . 7
  - 2.4.3 Login Pages . . . 7

3 Fetching with `urlib2` **7**
- 3.1 Using the Developer Tools Network Tab . . . 7
- 3.2 Bypassing the Browser . . . 7
  - 3.2.1 URL Query Strings . . . 7
  - 3.2.2 HTTP Requests . . . 7
    - 3.2.2.1 `GET` . . . 7
    - 3.2.2.2 `POST` . . . 7
  - 3.2.3 Retrieving Files . . . 7
  - 3.2.4 Demo with WireShark . . . 7
- 3.3 Authentication . . . 7
- 3.4 Refused/Timed out Connections . . . 7

4 Beginning New Projects **7**
4.1 Defining the Goal 7
4.2 Identifying the Obstacles 7
4.3 Selecting the Right Tool 7
4.4 Avoiding Pitfalls 7

5 Example Project Demo **7**
5.1 Accomplishing our Goal With `Selenium` 7
5.1.1 Walkthrough of Process 7
5.1.2 Strengths and Limitations 7
5.2 Accomplishing our Goal With `BeautifulSoup` 7
5.2.1 Walkthrough of Process 7
5.2.2 Strengths and Limitations 7
5.3 Accomplishing our Goal With `urllib2` 7
5.3.1 Walkthrough of Process 7
5.3.2 Strengths and Limitations 7

1 Scraping with `Selenium`

In order to scrape data from a website, we have to first understand what makes up a website and how data on a website is organized. Once we know this, we can tell our script the location of the data we're interested in so that it can grab it for us.

1.1 What Makes Up A Website

A web page is made out of three core elements: **HTML**, **CSS**, and **JavaScript**. HTML is mostly in charge of the content and structure of the website – like all the text and images, and the different parts of a page.

```
<h1>Hello, World!</h1>
```

CSS is in charge of the style of the website and layout of the website – such as the colours, fonts, and what content is next to or overtop of what.

```
 h1 {
  color:red;
  text-align:center;
}
```

JavaScript is in charge of the interactivity of a webpage – popups, timeouts, reactions to clicks and mouse movements, and so on.

```
document.onclick = () =>
  alert("You clicked!");
```

This is a good overview, but in reality the functionality of these different languages can overlap somewhat. For instance both CSS and JavaScript can be used to create animations; both CSS and HTML can be used to enforce page layout; and interactivity can be achieved with different kinds of languages called server-side languages, like C#, PHP, and Python, which we will discuss later.

1.1.1 Using the Chrome Developer tools

To explore these elements, you can use your browser's developer tools. In this course, we will be using the Chrome browser – but all modern browsers will have analogous functionality.

To open developer tools, right-click the page and choose the **Inspect** context-menu item. Alternatively, you can press **F12**. Let's poke around a website and see some examples of HTML, CSS, and JavaScript in action.

`https://github.com/crclayton/invoicing/commits/master`.

1.1.2 HTML Tags

So here we can see that HTML is structured into **tags**, that look like so:

`<tag>content</tag>`.

Where `<tag>` begins the content, and `</tag>` with the forward-slash ends the content within.

Some common tags are `p` for paragraph, `div` for page division, `h1, h2, etc.` for headers, `a` for links, and `li` for list items. Even without any styling, a browser will look at these tags and make educated guesses for how to display the content within them.

Here's a comprehensive list of the tags a website may have: `http://www.w3schools.com/tags/default.asp`.

Tags also have attributes, also called properties. These just specify more information about the tag other than its content. For instance, a `input` tag has an attribute which can specify whether it will input a date, a password, plain-text, just a check-mark, and so on.

`https://codepen.io/pen/`.

```
<input type="text" value="Input"\>
```

Different tags have different attributes, but the most common attributes are the **id** and the **class** attributes. An id is unique to a tag, and allows it to be directly identified.

```
.demo {
```

```
  background:lime;
  font-size:large;
}
```

A class isn't unique, and is usually used to assign a common CSS style to all tags of that class.

1.1.3 CSS Classes

Tags can have multiple classes. The helpful thing about classes when webscraping isn't that they identify common styles, it's that we can use them to locate the data we want. If there are repeating tags containing the data we want, we can be fairly confident they will have a common class.

For instance, take a look at this GitHub commit log.

`https://github.com/crclayton/invoicing/commits/master`.

If I was interested in downloading a record of all the commit messages and who posted them, I could see that all messages all within an `a` tag and all authors are stored in a `span` tag. However, there are lots of other tags that are links and spans on this page that aren't commit messages, so we have to be more precise.

But since all the commit messages look the same, and all the authors look the same, we know that they probably have the same class.

And sure enough, each link with a commit message has a class `message`, and each commit author has a class `commit-author`.

Now we can use this to go through the page and gather the data in each tag with those classes.

1.1.4 JavaScript Queries

We can locate the tags we want using JavaScript queries and CSS selectors.

I'm going to use JavaScript in the developer tools console to demonstrate some queries.

Although we won't be using JavaScript when we do our scraping, the way we identify the tags we want is the same in both Python and JavaScript, and it is faster to debug to make sure we are getting exactly what it is we want within the browser.

Using JavaScript, we can call the method `document.getElementsByClassName()`. and this will return a list of all those tags. You can see the other types of queries using the ID or another attribute called the Name.

If you expand out the list, you can hover over the items and see their corresponding location on the page.

1.1.5 CSS Selectors

A more powerful technique for these queries is using something called CSS selectors.

If you want to find something by the tag, just enter the tag. If you want to find something by a class name, enter the class preceded with a period. If you want to find something by the id, enter the id preceded by a pound-sign.

For instance

```
document.querySelector("#fork-destination-box")
```

and

```
document.getElementById("fork-destination-box")
```

Perform the same function, however, with CSS selectors we can nest queries more easily, specifying tags of a certain class within a tag with a certain ID, with an attribute that equals a certain value.

Consider the following example. Here we identify the tag by the id, then identify a tag within that tag, one of its descendants, by the class, then identify a tag within that tag by the tag type.

```
document.querySelector("#fork-destination-box .fork-select-fragment img")
```

Here's a comprehensive list of CSS selectors. The more adept you are at using these queries to precisely identify the data you want, the more streamlined and elegant your scraping can be.

`http://www.w3schools.com/cssref/css_selectors.asp`

1.1.6 XPaths

When all else fails. If there are no ids, no names, no classes, no attributes we can use. You may be able to use an XPath, which identifies a tag by its position in the HTML hierarchy. These aren't ideal because if a website makes a small change like adding or removing an element, this query may be broken. Whereas style and tag selectors are more robust.

For now, that will be all the web-development we will cover. However, I cannot stress enough the importance of familiarizing yourself with these queries as it can greatly simplify your code from code that looks something like this:

```
images = []
for e in browser.find_elements_by_class_name("example"):
    for i in e.find_elements_by_tag_name("img"):
        if "foo" in i.get_attribute("alt"):
            images.append(i)
```

To code that looks something like this:

```
images = browser.find_element_by_css_selector(".example > img[alt~=foo]")
```

1.2 Using the Selenium Module

1.2.1 Setting Up A WebDriver

1.2.2 Automating the Browser

1.2.2.1 Debugging with the Console

1.2.3 Downloading Files

1.2.4 Using a Headless Browser (PhantomJS)

2 Parsing with `BeautifulSoup 4`

2.1 Requesting HTML

2.1.1 `urllib2` preview

2.1.2 `Requests`

2.1.3 `wget`

2.2 `BeautifulSoup` Objects

2.3 `BeautifulSoup` Queries

2.3.1 Comparison To JS/Selenium Queries

2.3.2 Nested Selectors

2.3.3 Regular Expressions Basics

2.4 Inevitable Roadblocks

2.4.1 Server-side vs. Client-side Websites

2.4.2 UTF-8 Encoding

2.4.3 Login Pages

3 Fetching with `urlib2`

3.1 Using the Developer Tools Network Tab

3.2 Bypassing the Browser