Automation and Webscraping With Python

Charles Clayton

January 22, 2017

Course Overview

1 Scraping with `Selenium`

Welcome to the first volume in our course Getting Started with Python Web Scraping. This volume is called Scraping with Selenium.

I assure you we will cover selenium, but before we do so we will first discuss circumstances in which you might want to web scrape and why it's a valuable skill.

We will spend a good amount of time first covering how websites work and what they're made of. Without knowing this, it's impossible to write code to interact with them for you. After that we will install a webdriver, the tool needed to webscrape with selenium.

We will then walk though a basic example of webscraping, and I will discuss the technical aspects and the thought process as we go.

Finally we will conclude by discussing the ethics and legality of webscraping. And your responsibilities as a programmer and as a person.

1.1 When to Use Web Scraping

Let's say we need some data. Maybe it's sports statistics, maybe it's job listings, maybe you want to compare prices across certain competitors, the main take away from this course is that there's usually a more efficient way to get this data than doing it manually.

We might either want to write a program to repeatedly retrieve the data in the case that it's being updated, or it might just be way too tedious for us to copy and paste each relevant piece of data by hand into an excel spreadsheet or what have you. This is where webscraping comes in. This tool will allow us to automate the process of gathering data from the internet and be more productive in our day-to-day personal and professional lives.

After familiarizing yourself with webscraping, it also shapes how you view websites and the internet in general. You can see it as an easier source of data, which means you're more likely to be interested in accumulating the information and doing some of your own comparisons, analysis, or visualizations of it. This leads to making more informed and numerical decisions in the long run.

So, it's a really cool tool and a valuable skill to learn regardless of your field of work.

1.1.1 Before you Begin

Before we begin, this course assumes that you already have some foundation in Python programming, or even just programming in general. We also assume that you have Python installed, and that you know how to install Python modules using pip.

I will be using Python 3.4 for this section, and I recommend you use the same version. I will also be using a Windows 10 PC, but the process will be the same except for maybe some syntax in the events that we might be saving or loading files.

The first method of webscraping we will go over is using the Selenium module. Selenium is the best framework to start with because it deals immediately with the browser which everyone is familiar with, and the interactions happen tangibly in front of you.

1.2 What Makes Up A Website

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.

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.

1.2.1 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`.

1.2.2 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.

1.2.3 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/Microsoft/TypeScript`.

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.

1.2.4 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.

1.2.5 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`

1.2.6 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]")
```

1.3 Using the Selenium Module

Okay, I know that seems like a whole lot of fuss about nothing relevant so far. This is supposed to be about Python webscraping and so far we haven't touched Python, it's all been this web-development stuff. But now we'll start to use Python and you'll see the exact same techniques will apply, and that knowing how websites work and how they are built will help us immensely with our scraping.

1.3.1 Setting Up A WebDriver

So first we'll install our WebDriver. A WebDriver is commonly used as a tool for testing your web-applications. If you build an app, instead of clicking every single button and making sure it all works correctly, you can write code to simulate all sorts of clicks and entries into the browser and make sure it doesn't break.

For us, this is a handy tool that lets us write Python code to tell a browser what to do, and more importantly, grab the contents of a website.

For the WebDriver in this tutorial, we'll be using ChromeDriver which uses the Chrome browser and can be downloaded here: `https://sites.google.com/a/chromium.org/chromedriver/`.

Download the `.exe` and we'll put it in our project directory. Let's also create a Python file there, and we're going to be using Python 3.4.

We can now use our Python code to take hold of the browser with the `selenium` module like so:

```
import selenium
webdriver.Chrome("chromedriver.exe")
```

Now let's assign this to a variable called `browser`. We'll use this to start scraping in just a moment, but let's scroll through the autocomplete options to get an idea of what we can do with our browser now.

We can navigate to a website like so:

```
browser.get("https://google.com")
```

We can also insert any JavaScript code we want to run:

```
browser.execute_script("alert('hello')")
```

We can modify cookies, go back or forward or refresh, or even take screenshots of the current page if we want:

```
browser.save_screenshot("test.png")
```

And most importantly, we have those element finder queries we discussed before. Using these we can select elements and mess around with them. Let's select the google input textbox.

```
input = browser.find_element_by_css_selector("#lst-ib")
```

And again using autocomplete, I just want to show you what we can do with an element. Here we can use `input.click()` and you can see the cursor focuses there. We can also child elements, that is tags within the selected tag. We can also simulate text input with the `input.send_keys("example text")` method. Then clear the textbox with `input.clear()`.

So, we have quite a bit of power to tell the browser what to do. Now let's tell it to get some data for us!

1.3.2 Automating the Browser

Let's start by going to `https://reddit.com`. Perhaps I want to start by getting a list of all the posts on the front page. If I inspect element, can see that it has this class `title`, so maybe we start by using that.

```
titles = browser.find_elements_by_css_selector(".title")
```

Let's take a look at the text in all those elements using Python

```
[t.text for t in titles]
```

Hmm, that's not exactly what we want. It looks like other elements have the class `title` so that's messing up our query. Let's try to be a bit more precise.

1.3.2.1 Debugging with the Console

At this point, it's easier to switch to the developer console to debug so let's try that.

Recall that Python's `browser.find_elements_by_css_selector` and JavaScript's `document.querySelectorAll` do the exact same thing. So let's try

```
document.querySelectorAll(".title")
```

And see what we get. It looks like this element at the top here has that class, as well as these elements at the bottom. Furthermore, it looks like there is both a paragraph and an anchor tag with that class for each link, so we're getting doubles. We can refine our query by specifying we only want the links by putting an `a` in our queryselector followed by the class type.

There we go, now we only have the links, so we can get the text from those more easily. But let's take it a step further. It looks like this class `outbound` specifies whether the link goes to another website or stays on reddit.com. Let's say we only want external links. We can add another class to our selector and that refines our list even more.

```
document.querySelectorAll("a.title.outbound")
```

Now you might be thinking, well, if we can do this all in the browser, why even use Python. But the problem with this JavaScript, is we can't continue to run the code across page changes. So let's say we want to keep getting the text from these posts across to the next page, we can do that with our Python script.

Let's move our Python code into a working script now:

```
import selenium

browser = selenium.webdriver.Chrome("chromedriver.exe")
browser.get("https://reddit.com")
titles = browser.find_elements_by_css_selector("a.title.outbound")
titlesText = [t.text for t in titles]
```

Let's say we want to get the text for the first three pages. We'll now tell the browser to press the next page button and repeat our code.

Let's look at the next button, it has the class `next-button` and a quick test shows us that no other elements have that class. So instead of doing `find_elements`, which returns a list, let's use `find_element` which will return the first element it finds. In this case it'll be the only one.

And we'll call the `click` method on that element.

```
next = browser.find_element_by_css_selector(".next-button")
next.click()
```

Now let's run the full script, getting the posts of the first three pages.

```
from selenium import webdriver

browser = webdriver.Chrome("chromedriver.exe")
browser.get("https://reddit.com")

postTitles = []

for i in range(3):
    titles = browser.find_elements_by_css_selector("a.title.outbound")
    postTitles +=  [t.text for t in titles]

    next = browser.find_element_by_css_selector(".next-button")
    next.click()

for post in postTitles:
    print(post)
```

One problem you might get depending on your console is that it could stuggle with some unicode characters. If that's the case, don't worry because Python 3 is handling it and getting the data, it's just that your console doesn't support the characters. For instance, my visual studio console here doesn't have a problem printing it, but the default console doesn't know how to decode the characters.

If you just want to print, you can try using this code:

```
import sys
...
print(post.encode(sys.stdout.encoding, errors='replace'))
```

We'll discuss encoding in more detail in later videos.

1.4 Ethics

This leads us to a point where we should discuss the ethics of webscraping. It's important to remember that people run these websites, and that it costs them money to service requests. We only went through five pages, but reddit was forced to service far more data faster than it expects from a typical user. Imagine if we had

decided to run our script across the first 100 pages – this would an aggressive scraper and we would be imposing an irresponsibly large workload on reddit, disrupting their business.

First and foremost, this is unkind. Case closed.

Beyond that, some scraping is against the terms and conditions of certain websites. If you're found to be scraping aggressively, a website can easily detect this and you are likely to find yourself blocked from certain websites. The simplest way to prevent this, is to impose delays in your script. Sure, it'll take longer, but run it at night and you won't notice any difference and you won't be hogging the throughput of the website. Moreover, you won't find yourself blocked and your scraper broken.

Finally, you aren't going to find yourself coding around captchas.

The law revolving around webscraping is complicated, varies by country, is somewhat subjective, and challenging to enforce. The best advice is to be considerate, imagine you were the one running the website, and use common sense.

Where possible, using an API – an application program interface – is a wise alternative to traversing the HTML. These are mechanisms by which a website will allow you to make queries directly to the server, without using the middle-man as a browser.

We will examine APIs in a later section.

1.4.1 Using a Headless Browser (PhantomJS)

At this point, some programmers may feel uncomfortable with the browser popping up on the screen when they run their script. It would be nice if this could be done as an invisible, background task instead of hogging the computer while you could be doing something else.

I recommend writing your script using Chrome to debug, but once you have a working script. You may consider using a tool such as PhantomJS. PhantomJS is a browser that we can control with our webdriver, however it's headless – that is, invisible.