# Chapter 4: Web Scraping

Prev: Chapter 3 - Cryptography | Next: Chapter 5 - GUIs

## Summary:

Big data. Big deal. In this chapter, we're going to take a gander at a subject known as "Web Scraping." What's that you ask? Is that when you crawl the Internet for days trying to escape the dark web? No, that's called poor judgement. Web scraping is when a script or program retrieves information from a website for processing, analyzing, or organizing.

So why did I mention big data? Well, since the computer is retrieving the data from the website(s), it can get that data and crunch through it pretty quickly. So, naturally, web scraping is popular among those seeking the usage and analytics of "big data." You could grab the numbers and statistics from a large collection of websites, store it in a neat data structure, and produce whatever your heart desires–graphs, charts, tables, etc. manufactured from the data.

Maybe handling a sea of data seems daunting to you though. Perhaps we could study the more common applications of web scraping? Sure! Web scraping can be used for myriad tasks, such as: - Sports statistics - Social media trends - Online product comparisons - Stock data aggregation - Crime statistics

So, in this chapter we'll get to look at the basics of scraping the web using a pretty sweet third-party Python package called *BeautifulSoup*.

---

## Content:

Alright, so first we need to install *BeautifulSoup* for Python. The newest version, as of the writing of this book, is v4.4.1, which supports both Python 2.6+ and Python 3. You can install the package with whatever package manager you prefer, such as *pip* or *easy_install*, like so:

```
pip install beautifulsoup4
```

```
easy_install beautifulsoup4
```

Or, if you don't like that method, you can always visit the website for the package at https://www.crummy.com/software/BeautifulSoup/bs4/download/.

To test that you installed the package successfully, you can simply start the Python interpreter, and run the following import command:

```
from bs4 import BeautifulSoup
```

If that import command completed without error, you're setup and ready to scrape!

---

Now then, before we actually start using the *bs4* package, let's first take a look at the general process of web scraping, as follows: 1. Study the format of target webpage 2. Retrieve content of webpage (an HTTP request) 3. Search the content for specific data based off the content format 4. Do stuff with the data found in the search

Okay, let's put this into practice. This first example is so simple, we don't need to use *bs4*.

```
# Example 4-1 (scrape0.py):
import requests
page = requests.get('http://github.com/zach-king/')
print(page.content)
```

Here, we're using the builtin *requests* package from Python (2 and 3). After importing the package, we use the `get()` function to retrieve the HTML document that the server sends back for my GitHub profile, at http://github.com/zach-king.

After storing that in a variable called `page` we can print its contents out and verify the same HTML is received this way as when we view it in a web browser. Keep in mind that you can view the source code of a webpage in your browser by right clicking and selecting either *inspect element* or *view source* (I recommend using *inspect element* since you can more accurately tell what elements you are looking at).

Now that we've seen the basic retrieval of HTML from a webpage, we can actually begin *scraping* it for our target information. The two most popular methods of doing this task in Python are using either *regular expressions* or *bs4*. However, regular expressions–which will be covered in chapter 9– are typically not suggested in comparison with bs4. Let's take a look at an example using the `BeautifulSoup` class in bs4, and see the basics it has to offer:

```
# Example 4-2 (scrape1.py)
from bs4 import BeautifulSoup
from urllib.request import urlopen

url = "http://www.github.com/zach-king/CoolPython"
content = urlopen(url).read()
print(content)

soup = BeautifulSoup(content)

print(soup.prettify())
print(soup.title)
print(soup.title.string)
print(soup.p)
print(soup.a)
```

First off, note that this time we use the *urllib.request* package and its `urlopen()` function for retrieving the HTML; this is simply to showcase another way of accomplishing the same task. As you can see, we just store the HTML content in the content variable and print it out (ugly output, that is). Then, we create the mystical `BeautifulSoup` object with the HTML content passed in. After creating that, we can access many things easily, such as the `<title>` element and text, the first `<p>` (paragraph) element, and the first `<a>` (link) element. Also note that the soup object has a `prettify()` method which returns a long string of a *prettier* format than the jumbled HTML originally received.

While this is cool and all, next we need to start figuring out how to really get something useful out of the webpage. The next example demonstrates the scraping of all the links found on the repository page for this book:

```
# Example 4-3 (scrape2.py)
from bs4 import BeautifulSoup
from urllib.request import urlopen

url = "http://www.github.com/zach-king/CoolPython"
content = urlopen(url).read()

soup = BeautifulSoup(content)

for link in soup.find_all('a'):
    print(link.get('href'))
```

Nearly all of the code is the same, except for the last snippet. This uses one of BeautifulSoup's most powerful methods, `find_all()`. This method can be used in a plethora of ways, with different arguments passed to it. In this case, we want all of the `<a>` tags, and more specifically we want their actual link (the `href` attribute). So if we pass in 'a' to `soup.find_all()`, we will get a list of items of type `bs4.element.Tag`. This class in bs4 then contains a nifty method called `get()` used for getting the value of attributes within the tag; so to extract the href from the links we simply call this method and pass 'href' as the argument!

After scraping all the links out of a page, you might begin to wonder how you would scrape different tags *from within other tags*. This comes up when you find yourself targeting a specific `<div>` or section of a webpage, and you need to look at its *children*.

So let's take a look at yet another example to learn how to do just that!

```
# Example 4-4 (scrape3.py)
from bs4 import BeautifulSoup
from urllib.request import urlopen
```

```python
url = "https://github.com/zach-king?tab=repositories"
content = urlopen(url).read()

soup = BeautifulSoup(content, "html.parser")

for repo in soup.find_all("h3", {"class":"repo-list-name"}):
    children = repo.children
    for child in children:
        print(child)
```

Once again, no surprises with the HTML and soup object. But wait! This time we used the `find_all()` method completely differently! Yes, this time we are scraping the repositories from my GitHub profile. If you first study the HTML of the repositories tab on any GitHub profile, you'll notice each repo's title is stored in an `<a>` tag *within* an `<h3>` tag that has a class attribute of 'repo-list-name.' Thus, we find these specific h3 tags using `find_all()` and pass in a dictionary of required attribute/value pairs. Then, for each matched h3 tag, we access its children with the `children` attribute, loop through them, and print them out! How easy was that?

Now, I'm sure you're thinking these examples are dull and only exist to generate more visits to my GitHub profile–boring as it is–and that's why the last 'examples' I have are actual applications of 'web scraping.'

This first application is very simple; it simply scrapes the 10 most viewed bills on congress.gov.

```python
# Example 4-5 (gov-bills-scraper.py)
import requests
from bs4 import BeautifulSoup

url = "https://www.congress.gov/resources/display/content/Most-Viewed+Bills"
r = requests.get(url)
soup = BeautifulSoup(r.content, 'html.parser')
for i, bill in enumerate(soup.find_all("td", {"class":"confluenceTd"})[:30]):
    try:
        print(bill.text)
    except:
        pass
    if (i+1) % 3 == 0:
        print("\n")
```

Let's break it down. The first step in writing an application like this is to study the HTML in the web browser. I could see that the bills' information was being stored in `<td>` (table data) tags inside a table. Please note that tables are sometimes difficult to scrape information from, depending on how standardized and well-designed the table is.

The things I wanted for each bill was its Senate or House of Representatives code, its title, and its rank in the most-viewed list. These three things are found consecutively for each bill, but each bill has a span (no relation to the span tag…) of five matched td tags.

So I know that the *top 10* bills will consist of the first 30 matched td tags. Furthermore, with every third td tag, I want to print out a newline to provide some separation of the bills in the output. To do this I use the `enumerate()` function in Python which counts the iterations for me; then I use the modulus operator to print a '\n' after every three iterations. And that's it!

Now, the very last example is a bit longer than the previous ones, and for that reason, I will not go through it line by line. It is an application which takes a Python list of 'symbols' for different companies and scrapes their current value from the Yahoo! finance page. This example particularly showcases the capabilities of 'web-scraping' since it actually 'scrapes' multiple pages, and then produces a bar graph of that data–colored and all!

The foundation of this application was of course studying how the URLs of these company stock pages are formed on Yahoo! finance. I noticed that the website uses a query with a variable `s` (probably stands for symbol) which stores the value of the symbol, or company, being looked up. So the URL for Apple on Yahoo! finance is http://finance.yahoo.com/q?&s=aapl.

From there, it was all about using bs4 to extract the desired information out of the web pages. I'll leave the example here for you to study, if you so wish, and keep in mind that all source/examples can be found at the book's GitHub page. Here's the application:

```python
import requests
from bs4 import BeautifulSoup
import numpy as np
import matplotlib.pyplot as plt

# Variable declarations
symbols = ["aapl", "goog", "msft", "yhoo", "amzn", "fb", "twtr", "nflx"]
numCompanies = len(symbols)
values = []
arrows = []

# Scrape the data
for s in symbols:
    url = "http://finance.yahoo.com/q?&s=" + s
    r = requests.get(url)
    soup = BeautifulSoup(r.content, "html.parser")
    price = soup.find_all("span", {"id": "yfs_l84_"+s})
    arrow = soup.find_all("span", {"id": "yfs_c63_"+s})
    arrows.append(arrow[0].next_element["class"])
    print(s.upper() + " share value: $" + price[0].text)
    values.append(price[0].text)
    symbols[symbols.index(s)] = symbols[symbols.index(s)].upper()

# convert values from strings to floats
for i in range(numCompanies):
    values[i] = float(values[i])

# Plot and figure
fig = plt.figure()
fig.canvas.set_window_title('Stock-Scraper')
ax = fig.add_subplot(111)

index = np.arange(numCompanies)
rects = ax.bar(index, values, 0.25, alpha=0.5)

# Show the number value for the share above the bar
for i in range(numCompanies):
    ax.annotate('$' + str(values[i]), xy=(i+0.12, values[i]), xytext=(i - 0.12, values[i] +
            arrowprops=dict(arrowstyle='-'),
            )

# Labels and titles on graph
plt.xlabel('Company')
plt.ylabel('Share Value')
plt.title('Current Share Value for Companies')
plt.xticks(index + 0.25, symbols)

# Color the bars based on the companies' stock status
for i, arrow in enumerate(arrows):
    if (arrow == [u'pos_arrow']):
        rects[i].set_color('g') # stock is going up
    elif (arrow == [u'neg_arrow']):
        rects[i].set_color('r') # stock is going down
    else:
        rects[i].set_color('b') # unknown data

# Show the graph
fig.tight_layout()
plt.show()
```

---

## Wrap Up:

Well there you have it. You now have the metaphorical 'scraper' known as Python, waiting to crawl the World Wide Web. I highly recommend reading more on the documentation for bs4 if you're interested in strengthening your scraping skills.

While you do have this new power at your fingertips though, let me stress the importance of its ethical usage. To put it bluntly, the line defining the legality of 'web-scraping' is currently quite vague; there have been hearings related to the activity and some have yielded good results, while others. . . not so much. The concept of retrieving data from a webpage is entirely legal of course, but just be sure to keep your programs away from proprietary or sensitive information. Furthermore, I would like to remind you that I take no responsibility for your actions. . .

Prev: Chapter 3 - Cryptography | Next: Chapter 5 - GUIs