Web Scraping Reference: A Simple Cheat Sheet for Web Scraping with Python

October 24, 2018

Once you've put together enough web scrapers, you start to feel like you can do it in your sleep. I've probably built hundreds of scrapers over the years for my own projects, as well as for clients and students in my web scraping course.

Occasionally though, I find myself referencing documentation or re-reading old code looking for snippets I can reuse. One of the students in my course suggested I put together a "cheat sheet" of commonly used code snippets and patterns for easy reference.

I decided to publish it publicly as well – as an organized set of easy-to-reference notes – in case they're helpful to others.

While it's written primarily for people who are new to programming, I also hope that it'll be helpful to those who already have a background in software or python, but who are looking to learn some web scraping fundamentals and concepts.

Table of Contents:

Useful Libraries

For the most part, a scraping program deals with making HTTP requests and parsing HTML responses.

I always make sure I have `requests` and `BeautifulSoup` installed before I begin a new scraping project. From the command line:

```
pip install requests
pip install beautifulsoup4
```

Then, at the top of your `.py` file, make sure you've imported these libraries correctly.

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

Making Simple Requests

Make a simple GET request (just fetching a page)

```
r = requests.get("http://example.com/page")
```

Make a POST requests (usually used when sending information to the server like submitting a form)

```
r = requests.post("http://example.com/page", data=dict(
    email="me@domain.com",
    password="secret_value"
))
```

Pass query arguments aka URL parameters (usually used when making a search query or paging through results)

```
r = requests.get("http://example.com/page", params=dict(
    query="web scraping",
    page=2
))
```

Inspecting the Response

See what response code the server sent back (useful for detecting 4XX or 5XX errors)

```
print r.status_code
```

Access the full response as text (get the HTML of the page in a big string)

```
print r.text
```

Look for a specific substring of text within the response

```
if "blocked" in r.text:
    print "we've been blocked"
```

Check the response's Content Type (see if you got back HTML, JSON, XML, etc)

```
print r.headers.get("content-type", "unknown")
```

Extracting Content from HTML

Now that you've made your HTTP request and gotten some HTML content, it's time to parse it so that you can extract the values you're looking for.

Using Regular Expressions

Using Regular Expressions to look for HTML patterns is famously NOT recommended at all.

However, regular expressions are still useful for finding specific string patterns like prices, email addresses or phone numbers.

Run a regular expression on the response text to look for specific string patterns:

```
import re  # put this at the top of the file
...
re.findall(r'\$[0-9,.]+', r.text)
```

Using BeautifulSoup

BeautifulSoup is widely used due to its simple API and its powerful extraction capabilities. It has many different parser options that allow it to understand even the most poorly written HTML pages – and the default one works great.

Compared to libraries that offer similar functionality, it's a pleasure to use. To get started, you'll have to turn the HTML text that you got in the response into a nested, DOM-like structure that you can traverse and search

```
soup = BeautifulSoup(r.text, "html.parser")
```

Look for all anchor tags on the page (useful if you're building a crawler and need to find the next pages to visit)

```
links = soup.find_all("a")
```

Look for all tags with a specific class attribute (eg `<li class="search-result">...</li>`)

```
tags = soup.find_all("li", "search-result")
```

Look for the tag with a specific ID attribute (eg: `<div id="bar">...</div>`)

```
tag = soup.find("div", id="bar")
```

Look for nested patterns of tags (useful for finding generic elements, but only within a specific section of the page)

```
tags = soup.find("div", id="search-results").find_all("a", "external-links")
```

Look for all tags matching CSS selectors (similar query to the last one, but might be easier to write for someone who knows CSS)

```
tags = soup.select("#search-results .external-links")
```

Get a list of strings representing the inner contents of a tag (this includes both the text nodes as well as the text representation of any other nested HTML tags within)

```
inner_contents = soup.find("div", id="price").contents
```

Return only the text contents within this tag, but ignore the text representation of other HTML tags (useful for stripping our pesky `<span>`, `<strong>`, `<i>`, or other inline tags that might show up sometimes)

```
inner_text = soup.find("div", id="price").text.strip()
```

Convert the text that are extracting from unicode to ascii if you're having issues printing it to the console or writing it to files

```
inner_text = soup.find("div", id="price").text.strip().encode("utf-8")
```

Get the attribute of a tag (useful for grabbing the `src` attribute of an `<img>` tag or the `href` attribute of an `<a>` tag)

```
anchor_href = soup.find("a")["href"]
```

Putting several of these concepts together, here's a common idiom: iterating over a bunch of container tags and pull out content from each of them

```
for product in soup.find_all("div", "products"):
    product_title = product.find("h3").text
    product_price = product.find("span", "price").text
    product_url = product.find("a")["href"]
    print "{} is selling for {} at {}".format(product_title, product_price, produ
```

Using XPath Selectors

BeautifulSoup doesn't currently support XPath selectors, and I've found them to be really terse and more of a pain than they're worth. I haven't found a pattern I couldn't parse using the above methods.

If you're really dedicated to using them for some reason, you can use the lxml library instead of BeautifulSoup, as described here.

Storing Your Data

Now that you've extracted your data from the page, it's time to save it somewhere.

Note: The implication in these examples is that the scraper went out and collected all of the items, and then waited until the very end to iterate over all of them and write them to a spreadsheet or database.

I did this to simplify the code examples. In practice, you'd want to store the values you extract from each page as you go, so that you don't lose all of your progress if you hit an exception towards the end of your scrape and have to go back and re-scrape every page.

Writing to a CSV

Probably the most basic thing you can do is write your extracted items to a CSV file. By default, each row that is passed to the `csv.writer` object to be written has to be a python `list`.

In order for the spreadsheet to make sense and have consistent columns, you need to make sure all of the items that you've extracted have their properties in the same order. This isn't usually a problem if the lists are created consistently.

```
import csv
...
with open("~/Desktop/output.csv", "w") as f:
    writer = csv.writer(f)

    # collected_items = [
    #   ["Product #1", "$10", "http://example.com/product-1"],
    #   ["Product #2", "$25", "http://example.com/product-2"],
    #   ...
    # ]

    for item_property_list in collected_items:
        writer.writerow(item_property_list)
```

If you're extracting lots of properties about each item, sometimes it's more useful to store the item as a python `dict` instead of having to remember the order of columns within a row. The `csv` module has a handy `DictWriter` that keeps track of which column is for writing which dict key.

```
import csv
...
field_names = ["Product Name", "Price", "Detail URL"]
with open("~/Desktop/output.csv", "w") as f:
    writer = csv.DictWriter(f, field_names)

    # collected_items = [
    #   {
    #       "Product Name": "Product #1",
    #       "Price": "$10",
    #       "Detail URL": "http://example.com/product-1"
    #   },
    #   ...
    # ]

    # Write a header row
    writer.writerow({x: x for x in field_names})
    for item_property_dict in collected_items:
        writer.writerow(item_property_dict)
```

Writing to a SQLite Database

You can also use a simple SQL insert if you'd prefer to store your data in a database for later querying and retrieval.

```
import sqlite3

conn = sqlite3.connect("/tmp/output.sqlite")
cur = conn.cursor()
...
for item in collected_items:
    cur.execute("INSERT INTO scraped_data (title, price, url) values (?, ?, ?)",
        (item["title"], item["price"], item["url"])
    )
```

More Advanced Topics

These aren't really things you'll need if you're building a simple, small scale scraper for 90% of websites. But they're useful tricks to keep up your sleeve.

Javascript Heavy Websites

Contrary to popular belief, you do not need any special tools to scrape websites that load their content via Javascript. In order for the information to get from their server and show up on a page in your browser, that information *had* to have been returned in an HTTP response *somewhere*.

It usually means that you won't be making an HTTP request to the page's URL that you see at the top of your browser window, but instead you'll need to find the URL of the AJAX request that's going on in the background to fetch the data from the server and load it into the page.

There's not really an easy code snippet I can show here, but if you open the Chrome or Firefox Developer Tools, you can load the page, go to the "Network" tab and then look through the all of the requests that are being sent in the background to find the one that's returning the data you're looking for. Start by filtering the requests to only `XHR` or `JS` to make this easier.

Once you find the AJAX request that returns the data you're hoping to scrape, then you can make your scraper send requests to this URL, instead of to the parent page's URL. If you're lucky, the response will be encoded with `JSON` which is even easier to parse than HTML.

```
print r.json()  # returns a python dict, no need for BeautifulSoup
```

Content Inside Iframes

This is another topic that causes a lot of hand wringing for no reason. Sometimes the page you're trying to scrape doesn't actually contain the data in its HTML, but instead it loads the data inside an iframe.

Again, it's just a matter of making the request to the right URL to get the data back that you want. Make a request to the outer page, find the iframe, and then make another HTTP request to the iframe's `src` attribute.

```
inner_content = requests.get(soup.find("iframe")["src"])
```

Sessions and Cookies

While HTTP is stateless, sometimes you want to use cookies to identify yourself consistently across requests to the site you're scraping.

The most common example of this is needing to login to a site in order to access protected pages. Without the correct cookies sent, a request to the URL will likely be redirected to a login form or presented with an error response.

However, once you successfully login, a session cookie is set that identifies who you are to the website. As long as future requests send this cookie along, the site knows who you are and what you have access to.

```
# create a session
session = requests.Session()

# make a login POST request, using the session
session.post("http://example.com/login", data=dict(
    email="me@domain.com",
    password="secret_value"
))

# subsequent requests that use the session will automatically handle cookies
r = session.get("http://example.com/protected_page")
```

Delays and Backing Off

If you want to be polite and not overwhelm the target site you're scraping, you can introduce an intentional delay or lag in your scraper to slow it down

```
import time

for term in ["web scraping", "web crawling", "scrape this site"]:
    r = requests.get("http://example.com/search", params=dict(
        query=term
    ))
    time.sleep(5)  # wait 5 seconds before we make the next request
```

Some also recommend adding a backoff that's proportional to how long the site took to respond to your request. That way if the site gets overwhelmed and starts to slow down, your code will automatically back off.

```
import time

for term in ["web scraping", "web crawling", "scrape this site"]:
    t0 = time.time()
    r = requests.get("http://example.com/search", params=dict(
        query=term
    ))
    response_delay = time.time() - t0
    time.sleep(10*response_delay)  # wait 10x longer than it took them to respond
```

Spoofing the User Agent

By default, the `requests` library sets the `User-Agent` header on each request to something like "python-requests/2.12.4". You might want to change it to identify your web scraper, perhaps providing a contact email address so that an admin from the target website can reach out if they see you in their logs.

More commonly, this is used to make it appear that the request is coming from a normal web browser, and not a web scraping program.

```
headers = {
    "User-Agent": "my web scraping program. contact me at admin@domain.com"
}
r = requests.get("http://example.com", headers=headers)
```

Using Proxy Servers

Even if you spoof your User Agent, the site you are scraping can still see your IP address, since they have to know where to send the response.

If you'd like to obfuscate where the request is coming from, you can use a proxy server in between you and the target site. The scraped site will see the request coming from that server instead of your actual scraping machine.

```
r = requests.get("http://example.com/", proxies=dict(
    http="http://proxy_user:proxy_pass@104.255.255.255:port",
))
```

If you'd like to make your requests appear to be spread out across many IP addresses, then you'll need access to many different proxy servers. You can keep track of them in a `list` and then have your scraping program simply go down the list, picking off the next one for each new request, so that the proxy servers get even rotation.

Setting Timeouts

If you're experiencing slow connections and would prefer that your scraper moved on to something else, you can specify a timeout on your requests.

```
try:
    requests.get("http://example.com", timeout=10)  # wait up to 10 seconds
except requests.exceptions.Timeout:
    pass  # handle the timeout
```

Handling Network Errors

Just as you should never trust user input in web applications, you shouldn't trust the network to behave well on large web scraping projects. Eventually you'll hit closed connections, SSL errors or other intermittent failures.

```
try:
    requests.get("http://example.com")
except requests.exceptions.RequestException:
    pass  # handle the exception. maybe wait and try again later
```

Learn More

If you'd like to **learn more about web scraping**, I currently have an ebook and online course that I offer, as well as a free sandbox website that's designed to be easy for beginners to scrape.