Webscraping

- Harshal Chaudhari

Webscraping

Web scraping (web harvesting or web data extraction) is data scraping used for extracting data from websites. Web scraping software may access the World WideWeb directly using the Hypertext Transfer Protocol, or through a web browser.

- **Web-scraping is difficult sometimes, but most times annoying (not particularly intellectually challenging) coding.**
- **Web-pages change their source codes frequently, breaking your web-scraping codes.**
- **Source codes are not logical or consistent (most browsers do remarkably good job at parsing them, but your Python code most probably would not!).**
- **Websites go to insane lengths to prevent web scraping.**

  But,

- **I believe, if you are allowed to access website data in a browser window, why should some web developer dictate that you cannot access the same data from your Python code?**

Ethical issues

- **‘Webscraping’ maybe against the terms and conditions of some websites. Laws are yet to catch up with technology, so lawyers like to interpret ‘wget’ or ‘cURL’ commands as ‘hacking’.**

- **Irresponsible scraping codes can lead to D-DoS attacks!**

- **Constant scraping pings can affect the actual service provided by the website. For example, constantly pinging Uber API via your code can cause surge spikes for actual users.**

- **Harvesting data for personal use falls in the gray category legally, but using scraped data for commercial purposes is a strict NO!**

  Bottomline, don’t mess with the laws. Law recognises that the biggest tech company, Google, is in fact a webscraping company, and responsible webscraping and its uses are recognised by law. Some websites may impose additional restrictions in their terms and conditions.

What is webscraping?

- **Retrieve HTTP webpages**
  - The protocol for transferring HTML and other data over the internet.
  - A request-response protocol.
  - Both requests and responses have ‘headers’

- **Parse and extract information from the HTML webpage source codes.**
  - HTML / Regex parsers
  - JSON parsers

- **Modern websites use AJAX and JavaScript for client-side processing, thereby rendering above technique useless.**
  - Use XHR (XML HTTP Request) to decipher unofficial APIs.
  - Use PhantomJS or NodeJS (Python cannot execute Javascript code)
  - Use web-browser automation (Selenium)

Retrieve webpages

- **wget**
  - **‘web get’ allows fetching files and webpages over HTTP, HTTPS and FTP protocols.**
  - **Supports proxies, authentication, cookies, even has a ‘--spider’ option !**
  - **Respects ‘robots.txt’. Web developers always make sure to block it. :(**

- **cURL**
  - **‘See URL’ supports everything that wget supports, with additional protocols and robust features.**
  - **‘libcurl’ provides command line interface, but can also be used in Python via PyCurl package.**

- **Python web access libraries**
  - **urllib, Requests, etc. Sometimes buggy, especially while using with SOCKS proxies along with authentication.**

My favorite: Use PyCurl or simply use Python subprocess package to use cURL. Use --options to mimic a real world browser. Closer your request is to a standard HTTP request, lesser the odds of being detected.

Parsing webpages

- **For short very easily distinguishable patterns, use regular expressions (RegEx package in Python) to parse the source code.**
  - **Very simple and fast !**

- **For more complicated and reader friendly parsing, use BeautifulSoup package in Python.**
  - **Converts a webpage into DOM (Document Object Model, w3c standard)**
  - **Traverses DOM to parse the required elements using HTML tag attributes as identifiers.**
  - **Store data in JSON format.**

Since we have already used BeautifulSoup in first homework, I wouldn't go into the details. Let's now look into some practical issues and tips for web-scraping with examples.

Sitemaps

- **Consider example of scraping entire Amazon catalogue.**
  - **Creating a spider that traverses through pagination on entire Amazon website is complicated, time-consuming, probably impossible to make. Also, Amazon will easily detect and block you by IP address!**
  - **Amazon lets Google index their catalogue show that they show up in Google Search!**
  - **A sitemap is a hierarchy of XML files (or .gz files) containing list of web pages on website accessible to crawlers and users. Easy random uptodate access! Code would never break because sitemap protocol is designed by Google, very rarely updated.**

    Amazon sitemap: http://www.amazon.com/sitemaps.f3053414d236e84.SitemapIndex_0.xml.gz

Typically, we can find sitemap links in robots.txt. Let's check out sitemaps for following sites - TripAdvisor, Apple product reviews, Change.org and see some trickery involved.

Scraping AJAX websites

- **AJAX webpages make Javascript requests to the server**
  - **Fetch and process data on the client side**
  - **The HTML webpage source code you get in Python does not contain the data as JS scripts don't get executed. Example, https://jobs.apple.com/us/search**
  - **You could use PhantomJS or NodeJS.**

- **Solution:**
  - **XHR requests (if possible)**
  - **Automated browser instances**

Let's look at examples in both cases.

Finding AJAX request

- **Go to XHR, under the ‘Networks’ tab in the developer tools on Chrome. Firefox also has similar capabilities.**
- **When you reload the webpage, you should see a POST request being made. Let’s take a look at the request and response headers of this POST request, and the form-data sent with the request.**
- **Form-data contains a JSON string with two fields,**
  - **SearchRequestJson (contains fields like page numbers, sorting order, etc.)**
  - **clientOffset (seems to be fixed to a particular value)**
- **The response is in XML format.**

Now, our job is simplified. Instead of querying the jobs website URL, we simply send a POST request (using requests in Python or cURL) to the link from request header, with appropriate payload. This is effectively an unofficial API that provides ‘clean’ data.

Browser automation

- **Some websites use cookies to track users, if you cannot attach those cookies to every request, they won't serve you webpages!**
  - **You can use CookieJar in Python to automate cookie collection and attachment. Messy!**

- **Enter, browser automation.**
  - **Literally fires up an instance of a browser of your choice and opens the website in there.**
  - **There is no way the server can distinguish between an automated browser and a real user, barring captchas.**
  - **Handles cookies, authentication like a real browser. Retrieve webpage from this automated browser.**

Let us look at an example, where we scrape first page of videos of a particular search string from YouTube. However, some intermediate steps to install required packages.

Browser automation

- **Selenium:** A browser automation library in Python, Java, etc.

- **Chromedriver:**
  - Google provides chromedriver here: https://sites.google.com/a/chromium.org/chromedriver/home
  - Download appropriate chromedriver zip file, extract it.
  - Add the location to your PATH environmental variable.

- Now, let's get to it. Let's search for 'Formula One' videos on YouTube.

Helpful tips

- **Proxies:**
  - **If possible, always use proxies.**
  - **Allows multi-processing.**
  - **Most free proxies leak original IP.**

- **Web Scraping architecture:**
  - **Database (or some storage facility)**
  - **Multi-processing capable scraping code, independent of website to be scraped.**
  - **Website specific parsing code (will contain BeautifulSoup rules).**
  - **Driver code. Always use logging mechanism.**

Happy ethical responsible web scraping!