Introduction to Python for Social Science

Lecture 7 - Mining the Web

Musashi Harukawa, DPIR

7th Week Hilary 2020

This Week

Mining the Web

- This week we learn about automating data collection from the Internet.
- This is a powerful tool in your research arsenal, and a frequently sought-after skill by social science researchers.
- This also lays bare a broader goal that computational methods seek to accomplish: *automation*.

Roadmap

- How does the Internet work? (Short version)
- What kinds of data can we collect from the Internet?
- How can Python assist us in conducting this collection on a large scale?
- When is it (in)appropriate to scrape?
- Coding Tutorial

Final Note

The aim of this lecture/tutorial is twofold:

- To learn the *logic* behind a web scraper: understanding how data is structured on the web.
- To learn the *mechanics* of a web scraper: the tools for searching, selecting and filtering the data within these structures.

Writing a web scraper takes a lot of time and effort. Mastery of the tools will enable you to build these more efficiently, and focus on the *logic* instead of the *mechanics*.

How does the Internet work? (Abridged)

Our Experience

We are all familiar with how to access information on the Internet. We:

- Open our preferred browser.
- EITHER:
  - Type in the `URL` of the website we want to visit, OR
  - Type a query into our preferred search engine.
- Navigate the webpage with scrolling and clicks until we find what we want.

To understand how to automate this process, we need to understand which portions of this process can be *automated*.

To understand this, let's look under the hood to see what actually happens.

STEP 1: **REQUEST**

When you type a website into your browser's `URL` bar and hit `enter`:

- Your computer sends a hypertext transfer protocol (secure) (`HTTPS`) `GET` request to the specified `URL`.

Let's break this down:

- Protocol: `HTTP(S)`
- Action: `GET`
- Destination: `URL`

Protocol

- `HTTP` is an application-level data transfer protocol used on the Internet.
- `HTTPS` is a secured version of this protocol.
- Other protocols include `FTP` (file transfer protocol), `SSH` (secure shell), etc.

Action

- A `GET` request is an `HTTP` request for retrieving information from the target webserver.
  - Keep this in mind: accessing a webpage is similar to requesting a book or document from a library.

Destination: URL

A URL (uniform resource locator) is a reference to a web resource. They have the generic syntax[1]:

```
URI = scheme:[//authority]path[?query][#fragment]
authority = [userinfo@]host[:port]

userinfo       host      port

https://john.doe@www.example.com:123/forum/questions/?tag=n

scheme          authority                  path
```

[1] This example is taken directly from https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#Generic_syntax

URLs Explained

```
                host

https://muhark.github.io/dpir-intro-python/Week7/lecture.ht

scheme     authority                          path
```

- **scheme**: Protocol, as mentioned in the slide above.
- **authority**: Constructed of three *subcomponents*.
  - Two of them are optional (`userinfo@` and `:port`), and not seen here.
  - `host` is essentially the name of the webserver.
- **path**: A path in the internal filesystem of the webserver.
- **fragment**: Optional, "scrolls" to a specific sub-element within the webpage.

Aside: IP Addresses, DNS

(Heavy oversimplification on this slide)

- The destination webserver's "location" within the Internet is stored not as a `URL`, but usually as an IP address, e.g. `216.58.210.206`
- You can think of an IP address as acting exactly like a physical address for mail.
- To make sure that your request gets sent to the right desintation the host in the `URL` must be converted to an IP address. This conversion is done by `DNS`.
  - DNS resolvers are essentially the address books of the web. They maintain records of urls and IP addresses.
  - These resolvers are within the browser, our OS, and also maintained by third parties such as our ISPs, Google or Cloudflare.
- This usually does not matter too heavily for web scraping.

STEP 2: **RESPONSE**

Your request now having been routed to the correct address, the webserver:

- Reads the header, and accepts or declines the request.
- Reads the contents of the request (i.e. the path, query, fragment)
- Returns the data to the IP address given in the request packet header.
- Usually, this information will be in the formatted as a mixture of an `html`, `css` and `javascript`.
  - Exceptions: requesting `pdf` documents directly from webpages, interacting with a `php` server.

STEP 3: **RENDER**

A web browser is actually a specialised piece of software that can render and display all kinds of document formats, and allow you to interact with them via a graphical interface[2].

Many websites you deal with will be a mixture of `html`, `css` and `javascript`.

- `html` is more appropriately described as a data structure than a language. It provides the "skeleton" of the webpage and the textual elements.
- `css` is also a data structure, but provides *styling* information that informs much of the aesthetics of the webpage.
- `javascript` is a programming language that runs programs on the *client side* (i.e. in your computer) and creates interactive elements on webpages.

[2]There are CLI web browsers, such as `lynx`. These are fun to play around with if you want to explore the Internet without any graphical elements.

`html`

Hypertext Markup Language, or `html`, is the core of all webpages.

- Consists of *elements*, beginning and ending with a *tag*.
- Nested structure (like a dictionary).

html tags

- Tags define the type of element contained between them:
  - `<head>...</head>`: Defines the header block
  - `<h1>...</h1>`: Section header level 1
  - `<p>...</p>`: Paragraph
  - `<div>...</div>`: Defines a section in a document
  - For a full reference see here
- The front tag can contain additional attributes:
  - `<section id="title-slide">...</section>`
  - The `class` attribute allows for style inheritance from `css`.

html example

```
<section id="title-slide">
  <h1 class="title">Introduction to Python for Social Scien
  <p class="subtitle">Lecture 7 - Mining the Web</p>
  <p class="author">Musashi Harukawa, DPIR</p>
  <p class="date">7th Week Hilary 2020</p>
</section>
```

Inspecting Source

Most browsers allow you to inspect the source of the webpage that you are viewing. I recommend that you use Chrome/Chromium/Firefox.

Mac	Windows/Linux
`Command+Option+I`	`F12` or `Control+Shift+I`

The devtools in the browser allow you to inspect the `html` and `css` files that generate the webpages. Recognising how these are structured is key to web scraping.

Web Mining for Social Scientists?

Note on Terminology

I use the following three terms to refer to different things:

- *Web Scraping*: The automated collection of data from web pages.
- *Data Collection via API*: The automated collection of data from the web, via a server-provided API.
- *Web Crawling*: The automated traversing of web pages to search for information.

The difference will become clearer as I discuss them.

Social Science Use Cases

- Collecting parliamentary transcripts.
- Collecting pdfs of referendum texts (in Switzerland).
- Collecting government statistics (on education, for example).
- Collecting press releases.
- Gathering news articles stored online.
- Recording user interactions on Reddit.
- Gathering tweets.
- Analysing precinct-level crime data.
- ...

Trace Data vs Embedded Structured Data

It's useful to distinguish between *trace* data, and structured data embedded in websites.

- Trace data is incidental; it is generated by usage of online resources. Examples include Facebook or Reddit posts, website visit counts, Tweets, and so on.
  - Much of "big data" refers to massive volumes of trace data generated on a secondly basis by users on these websites.
  - Using trace data requires you to think more carefully about what exists, how that connects to the activity you are trying to measure. Missingness is less clear; it could be erased/censored entries, or offline activity.
- Embedded structured data (my term) refers to online archives of structured data. This could take the form of statistics stored in spreadsheets, transcripts of debates, and so on.
  - The objective here is entirely different; it may make sense to collect large portions of the archive and leverage other libraries to parse it afterwards.
- This is not a dichotomy; many things fall somewhere in between, such as press releases.

APIs

Some websites provide an *application programming interface* (API), which is an interface specifically designed for automated querying/retrieval.

- If an API exists, *use it*. APIs are more resource efficient than webpages, and exist in part to prevent unstructured scraping.
- Check for an API at the bottom of the webpage, sometimes in a section "for developers".
- There will usually be documentation for an API, and sometimes even a specific Python library (e.g. `tweepy` for Twitter).
- Because APIs are so specific to websites, I will not go over them in this class.

Using Python for Web Mining

Libraries

The following libraries are key for building web scrapers:

- `requests`: For making generic http(s) requests.
- `beautifulsoup`: "Cleans up" and provides powerful interface for navigating html.
- `re`: Regular expressions library for powerful string matching.

Some glaring omissions from this list:

- `scrapy`: For building deployable web crawlers
- `selenium`: Web testing library, can handle `javascript` and be programmed to behave much more like a human than a bot.

Retrieving Web Pages

Here's a function I wrote/adapted from *Web Scraping with Python, 2nd Ed.*

```
def safe_get(url, parser='html.parser'):
    try:
        session = requests.Session()
        headers = {"Accept": "text/html,application/xhtml+x
                   "User-Agent": "Mozilla/5.0 (X11; Fedora
                   }
        req = session.get(url, headers=headers)
    except requests.exceptions.RequestsWarning:
        return None
    return BeautifulSoup(req.text, parser)
```

try/except

```
try:
    session = requests.Session()
    [...]
    req = session.get(url, headers=headers)
except requests.exceptions.RequestsWarning:
    return None
```

`try/except` is a control flow structure that runs the code in the `try` block until an *exception* occurs, and if the *exception* is of the kind defined after `except`, then it executes the `except` block.

- In this case the function first runs the code from `session = ... to ... headers=headers)`
- If in the execution of this code, a `requests.exceptions.RequestsWarning` is *raised*, then the code `return None` is executed.
- If a different kind of exception occurs, then it is not *handled* by this except statement, and is raised normally (resulting in an error).

`requests`

There are just three things the `requests` library is being used for:

- Initiating a session.
- Sending a `GET` command to the provided url with a customized header.
  - This header is copied from a standard browser, to make the request appear as a regular user (and not a bot).
- Web-request specific errors/exceptions.

BeautifulSoup

`re`

A regular expression (RegEx) is a sequence of characters that defines a *search pattern*. These patterns are used to systematically and flexibly search through strings.

Python provides its own implementation of regular expressions, along with the built-in library `re`.

Understanding Regular Expressions

Regular expressions are constructed of:

- *regular characters*, which are the literal characters themselves
- *metacharacters*, which have special meanings

For instance, the regular expression `a.` contains:

- `a`: the regular character lowercase 'a'
- `.`: the metacharacter matching any character except a newline.

Metacharacters for Character Sets

- `.`: Any character other than `newline` (the character denoting that the subsequent character should be on a new line)
- `[]`: Defines a character set.

Metacharacters Defining Repetition

- `*`: Matches 0 or more consecutive instances of the previous regular expression. Matches as many as possible.
- `+`: Matches 1 or more consecutive instances of the previous regular expression. Matches as many as possible.
- `?`: Matches 0 or 1 instances of the previous regular expression.
- `{m}`: Matches exactly `m` instances of the previous regular expression.
- `{m, n}`: Matches between `m` and `n` instances of the previous regular expression.

Other Special Characters

- `^`: Matches the null character at the beginning of a string.
- `$`: Matches the null character at the end of a string.
- `\`: Converts the subsequent metacharacter to a literal.

This following pattern will match strings beginning with "comput":

```
comput.*
```

- The letters `comput` match EXACTLY those letters.
- The `.` symbol denotes ANY non-whitespace character.