# 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.

- Open our preferred browser.
- Type in the `URL` of the website we want to visit.
- 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*.

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 web server.
  - 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=

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 web server.
- **path**: A path in the internal filesystem of the web server.
- **fragment**: Optional, “scrolls” to a specific sub-element within the webpage.

## Aside: IP Addresses, DNS

*(Heavy oversimplification on this slide)*

- The destination web server's "location" within the Internet is stored not as a `URL`, but usually as an IP address, e.g. `216.58.210.206`
- An IP address is similar to a physical address for mail.
- Note: 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.

## STEP 2: **RESPONSE**

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

- 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.

## 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

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 Structued Resources

- 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.

- 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.

## Parsing `html`

## `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 Scie
  <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>
```

## `BeautifulSoup`

- `html` does not actually need to be "correct" to function; most parsers can deal with issues such as missing tags, etc.
- `BeautifulSoup` parses and "cleans" `html`, then represents the document as Python objects with useful methods for navigating and searching the tree, e.g.:
  - The `.text` method accesses the direct `html` of the tag.
  - `.child`: Returns the first child of the element.
  - `.parent`: Returns the immediate parent.
  - `.parents`: Iterates up the tree through each parent.

## Searching with `BeautifulSoup`

- It also provides powerful search functionality with `find()`, `find_all()`, etc. These searches can take:
  - Strings according to tag types.
  - Regular expressions (next section)
  - Keyword arguments: `id`, `class_`, and so on.
- Full documentation can be found online

## Regular Expressions

## `re`

A regular expression (RE) 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.
  - `[adw]` defines the set of any of `a`, `d` or `w`.
  - `[a-z]` defines the set of the 26 lowercase latin letters.
  - `[A-z0-9]` defines the set of all latin letters and the numbers 0-9.
  - `^` at the beginning of the set negates the following; `[^eng]` is all characters other than `e`, `n` or `g`

## More Metacharacter Character Sets

- `\w` matches all Unicode word characters, including numbers and underscore.
- `\W` matches all Unicode non-word characters, i.e. `[^\w]`
- `\s`/`\S` match respectively all whitespace and non-whitespace characters.

## Metacharacters Defining Repetition

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

## Combining Sets and Repetition

- `[a-z]*` matches 0 or more instances of lowercase latin letters.
- `[^0-9]?` matches 0 or 1 instances a non-number character.
- `[abc]{4, }[0-9]` will match 4 or more occurrences of `a`, `b` or `c` followed by a single number:
    - [x] `aaaaabac1`
    - [ ] `abcabc`
    - [ ] `abc0`
    - [x] `abcb01` -> will match the substring `abcb0`

## 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.
- `()`: Defines subgroups within the regular expression that can be extracted individually.

## Combined Workflow

## Scraping Data Files from an Archive

Say you have a webpage containing a large number of links, each leading to a csv file that you want to download.

1. Use `requests` to retrieve the `html` of the target webpage.
2. Parse the `html` and create a searchable object with `BeautifulSoup`
3. Identify the `<div>` in the webpage containing the download links.
4. Use a for-loop to go through the links in this section, appending them to a list of the hypertext matches the regex string `.*\.pdf`
5. Verify output, then use `requests` to recursively download the objects and write them to disk.

## Searching Politicians' Websites for Twitter Handles

1. Find a website listing all politicians' websites (usually exists).
2. Generate a list of all websites.
3. For each website:

- Retrieve/parse website with `requests` and `BeautifulSoup`
- Search `<body>` for a link or menu matching the regex `[Tt]witter`.
- If link, check whether the link is to twitter: `^(https?://)?twitter.com/.*` or internal `^/.*`.
  - If to `twitter.com`, then append to output.
  - If internal, pull page and repeat search for link that leads to Twitter account, but **not** Twitter's homepage.

## Good Practices

- The first and most important step in searching a page is identifying which "part" of the page you want to search, i.e. the node in the `html`-tree that contains all relevant matches. This can be found using the Inspector Tool on most browsers.
- `css` classes are often a very helpful tool for finding a series of like objects. Inspect the elements that you are looking for, maybe they are all contained in objects that possess the same `class` attribute.
- Writing a good regex string is as much about identifying true positives as it is about filtering out false positives. Do the filtering in a separate step if need be.

## When is it (in)appropriate to scrape?

## A Strong Warning

Unlike the tools we have discussed so far in this course, the tools used for web scraping can easily have unintended and damaging consequences.

- Web servers are physical computers/servers providing a filesystem to the web.
- They have resource constraints, and can only handle so many requests in a given period of time.
- Even laptops have the ability to send an enormous number requests per minute.
- If the volume of requests is greater than what the web server can handle, then you can significantly slow down, or even bring down a website.
- This is known as a Denial of Service attack, *and is essentially a cyber-attack*.

## Avoiding This

For the most part, it is unlikely that you will bring down a website; most web servers have countermeasures in place which will block IP addresses in response to a sudden high volume of requests. Nevertheless, make sure to:

- **Space out your requests**. Use the `sleep` function from the `time` library to wait 5-15 seconds between requests.
- **Test small portions of the script**. Do not run the full for-loop until you are absolutely sure that you have everything right.
- **Be considerate and efficient**. Do not request pages or resources that you do not need.
- **Know what you are doing**. Make sure you understand what is happening at each stage. If possible, have someone else check over your code.

## Legality

**I am not a lawyer, and this does not constitute legal advice.**

- Done well, web scraping will not put any more strain on a website than usual traffic.
- However, even when done well, be aware that web scraping is often in a legal gray zone.
- Check the ToS of websites you intend to scrape. If they say don't do it, then **don't do it**. Contact them instead, and ask whether they can arrange a data transfer.
- Relatedly, if there is an API, **use it**!

## Reference

## Readings

- Web Scraping: Mitchell, 2018. *Web Scraping with Python, 2nd Edition*. O'Reilly Publishing. Can be accessed for free via SOLO.
- Formal Language Theory: Becerra-Bonache et al, 2018. "Mathematical Foundations: Formal Grammars and Languages", in *The Oxford Handbook of Computational Linguistics, 2nd Edition*. OUP.
- Web Text Mining: Baeza-Yates et al, 2016. "Web Text Mining", in *The Oxford Handbook of Computational Linguistics, 2nd Edition*. OUP.

## Resources

- Python Web Scraping Exercises with Solutions
- Regex Testers:
  - Not Python specific, but the most full-featured: https://regexr.com/
  - Python specific, minimal: http://www.pyregex.com/
  - Python specific, with cheatsheet: https://pythex.org/
- Scraping Practice Website