# Big Data and Automated Content Analysis (12EC)
# Week 10: »Web Scraping«
# Wednesday

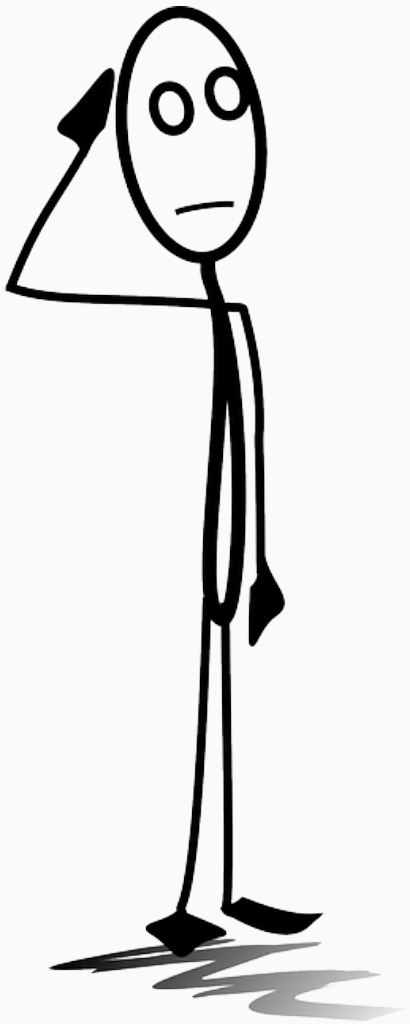
Felicia Loecherbach
f.loecherbach@uva.nl

April 19, 2023

UvA RM Communication Science

# Today

- The idea behind web scraping
  - APIs vs. web scraping
  - From HTML to structured data
- XPATHs and CSS Selectors
  - HTML documents as trees
- Scaling up
  - Avanced techniques
- Next steps



*Everything clear from last week?*

# Main points from last week

**I assume that by now, everybody knows:**

- how to work with textual data;
- and in particular (for this week) the concept of regular expressions

**This week, we will learn how to gather non-structured online data.**

# The idea behind web scraping

# The idea behind web scraping

## APIs vs. web scraping

# Ways to collect online data

1. Download existing datasets (that's trivial. . . )
2. Use an API (relatively easy)
3. Web scraping (anything between easy and extremely difficult)

# APIs vs web scraping

## APIs

- + structured (or at least semi-structured) data (JSON)
- + little programming effort needed
- − not always available; or restrictions apply
- − no guarantee it looks like what a human would see

## web scraping

- − unstructured data (HTML text)
- − (potentially) much programming effort needed
- + in principle always* possible
- + see what a human user would see*

* Don't get too excited. With dynamic websites that use a lot of stuff like JavaScript, Cookie Walls, etc., this can be difficult to unfeasible.

# APIs vs web scraping

## APIs

- + structured (or at least semi-structured) data (JSON)
- + little programming effort needed
- − not always available; or restrictions apply
- − no guarantee it looks like what a human would see

## web scraping

- − unstructured data (HTML text)
- − (potentially) much programming effort needed
- + in principle always* possible
- + see what a human user would see*

* Don't get too excited. With dynamic websites that use a lot of stuff like JavaScript, Cookie Walls, etc., this can be difficult to unfeasible.

If there's an API, you want to use the API (either directly with the `requests` package or via a so-called wrapper (a package that makes using a specific API even easier).

Otherwise, consider using web scraping.

# The idea behind web scraping

## From HTML to structured data

Let's have a look at some websites and understand the underlying structure.

**Websites change constantly!**

The examples are meant to illustrate the principles and approaches and are *not* meant as a practical guide for scraping specific websites. Websites change their structure quite regularly, and you cannot assume that scraping code written once keeps working in the future.

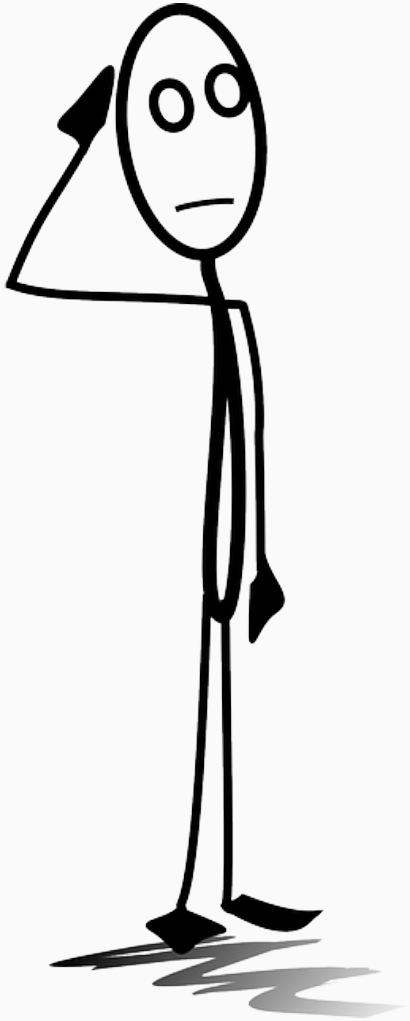
Except, of course, the simplified example at https://cssbook.net/d/eat/ – that one will be kept unchanged ;-)

Let's have a look at some websites and understand the underlying structure.

**Websites change constantly!**

The examples are meant to illustrate the principles and approaches and are *not* meant as a practical guide for scraping specific websites. Websites change their structure quite regularly, and you cannot assume that scraping code written once keeps working in the future.

Except, of course, the simplified example at https://cssbook.net/d/eat/ – that one will be kept unchanged ;-)



*Do you know some HTML?*

An example website

https://cssbook.net/d/eat/

# The best restaurants in town

On this site, you find reviews of the best restaurants in town

## Italian cuisine

### Pizzeria Roma

Here you can get ... ...

Read the full review here

### Trattoria Napoli

Another restaurant ... ...

Read the full review here

## Indian cuisine

### Curry King

Some description.

Read the full review here

---

This extremely ugly website was brought to you for illustrating some principles of web scraping.

If you view the underlying source code (depending on your browser, sth like “View Source”, “View Page Source”, or `CTRL-U`)...

An example website | https://cssbook.net/d/eat/

view-source:https://cssbook.net/d/eat/

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="mystyle.css">
    <title>An example website</title>
  </head>
  <body>
    <h1>The best restaurants in town</h1>
    <p> On this site, you find reviews of the best restaurants in town</p>
    <h2>Italian cuisine</h2>
    <div class="restaurant"> <h3> Pizzeria Roma </h3>
      <p> Here you can get ... ... </p>
      <p> Read the full review <a href="review0001.html">here</a></p>
    </div>
    <div class="restaurant"> <h3> Trattoria Napoli </h3>
      <p> Another restaurant ... ... </p>
      <p> Read the full review <a href="review0002.html">here</a></p>
    </div>

    <h2>Indian cuisine</h2>
    <div class="restaurant"> <h3> Curry King </h3>
      <p> Some description. </p>
      <p> Read the full review <a href="review0003.html">here</a></p>
    </div>

    <hr>
    <div class="footer">
      This extremely ugly website was brought to you for illustrating some principles of web scraping.
    </div>

  </body>
</html>
```

You can get an even more comfortable view of the source code using the “Inspect element” function:

An example website

https://cssbook.net/d/eat/

# The best restaurants in town

On this site, you find reviews of the best restaurants in town

## Italian cuisine

### Pizzeria R

Here you ca

Read the fu

### Trattoria

Another res

Read the fu

Copy
Select All
Print Selection
Take Screenshot
Search Google for “Pizzeria Roma”
View Selection Source
Inspect Accessibility Properties
Inspect (Q)
DownThemAll!
Bitwarden

## Indian cui

### Curry King

Some description.

Read the full review here

This extremely ugly website was brought to you for illustrating some principles of web scraping.

Developer Tools — An example w

Inspector | Console | Debugger | Style

Search HTML

```
<!DOCTYPE html>
<html>
▶ <head> ··· </head>
▼ <body>
    <h1>The best restaurants in town</h1>
  ▶ <p> ··· </p>
    <h2>Italian cuisine</h2>
  ▼ <div class="restaurant">
      <h3>Pizzeria Roma</h3>
      <p>Here you can get ... ...</p>
    ▶ <p> ··· </p>
    </div>
  ▶ <div class="restaurant">···</div>
    <h2>Indian cuisine</h2>
  ▶ <div class="restaurant">···</div>
    <hr>
  ▶ <div class="footer">···</div>
  </body>
</html>
```

html › body › div.restaurant › h3

Filter Output | Errors

An example website

https://cssbook.net/d/eat/

# The best restaurants in t

On this site, you find reviews of the best restaurants in to

## Italian cuisine

h3 | 8

### Pizzeria Roma

Here you can get ... ...

Read the full review here

### Trattoria Napoli

Another restaurant ... ...

Read the full review here

## Indian cuisine

### Curry King

Some description.

Read the full review here

This extremely ugly website was brought to you for illustrating some principles of

# Let's make a plan!

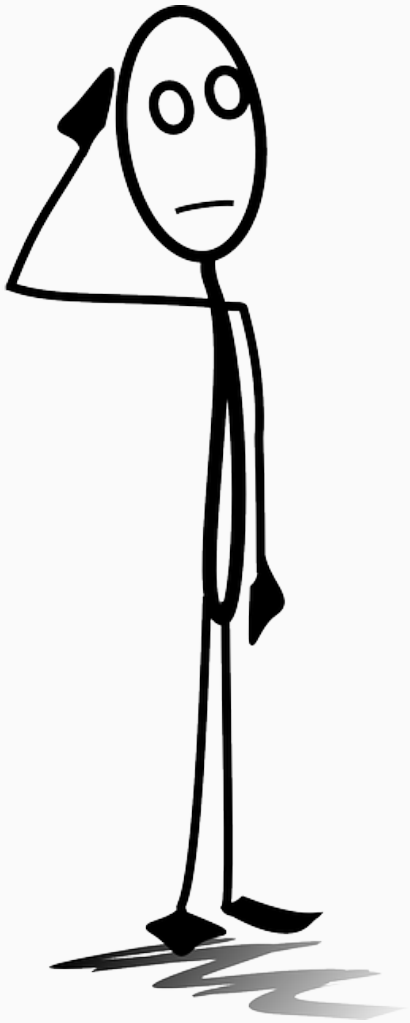
## Which elements from the page do we need?

- What do they mean?
- How are they represented in the source code?

## How should our output look like?

- What *lists* do we want?
- ...

And how can we achieve this?



*Go to https://cssbook.net/d/eat and explore which elements we may need!*

# 1. Retrieve the web page

```
1  import requests
2  r = requests.get("https://cssbook.net/d/eat/")
3  htmlsource = r.text   # that's all!
4
5  # Additional verification if needed:
6  # (a) this should print exactly the source code in the browser
7  print(htmlsource)
8
9  # (b) opening test.html in your browser should show the same page
10 # as you would have gotten with a "File/Save As" in the browser
11 with open("test.html", mode="w") as f:
12     f.write(r.text)
```

You see that it's exactly the same as with retrieving data from a JSON-based API – just that we use `.text` instead of `.json` in line 3.

# 2. Parse the HTML code

We could now use regular expressions to extract the relevant information

```
import re
re.findall(r"<h3>(.*?)</h3>", htmlsource)

# returns [' Pizzeria Roma ', ' Trattoria Napoli ', ' Curry King ']
```

# 2. Parse the HTML code

We could now use regular expressions to extract the relevant information

```
import re
re.findall(r"<h3>(.*?)</h3>", htmlsource)

# returns [' Pizzeria Roma ', ' Trattoria Napoli ', ' Curry King ']
```

# 2. Parse the HTML code

But:

- difficult for more complex pages
- error-prone
- hard to consider all edge cases (what about tags in tags? linebreaks? . . . )

**Others have written these regular expressions for you!**

Very few edge cases aside (broken pages, for instance), you do not write these (low-level) regular expressions yourself but use existing packages that let you describe the position of some content within a HTML file with an easier (high-level) syntax, so-called CSS Selectors and/or XPATHs (two new languages next to regexp, yeah![1])

[1] I promise they are easier!

## 2. Parse the HTML code

But:

- difficult for more complex pages
- error-prone
- hard to consider all edge cases (what about tags in tags? linebreaks? . . . )

**Others have written these regular expressions for you!**

Very few edge cases aside (broken pages, for instance), you do not write these (low-level) regular expressions yourself but use existing packages that let you describe the position of some content within a HTML file with an easier (high-level) syntax, so-called CSS Selectors and/or XPATHs (two new languages next to regexp, yeah![1])

[1] I promise they are easier!

# XPATHs and CSS Selectors

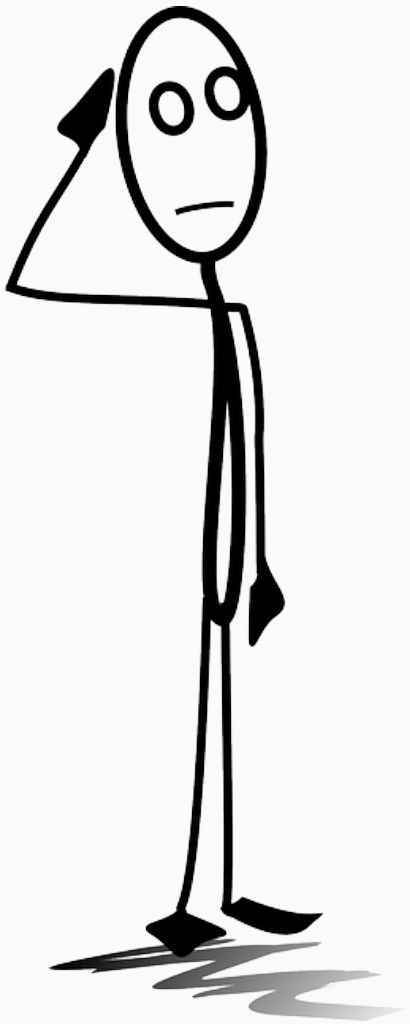
# XPATHs and CSS Selectors

## HTML documents as trees

# HTMl documents are hierarchical

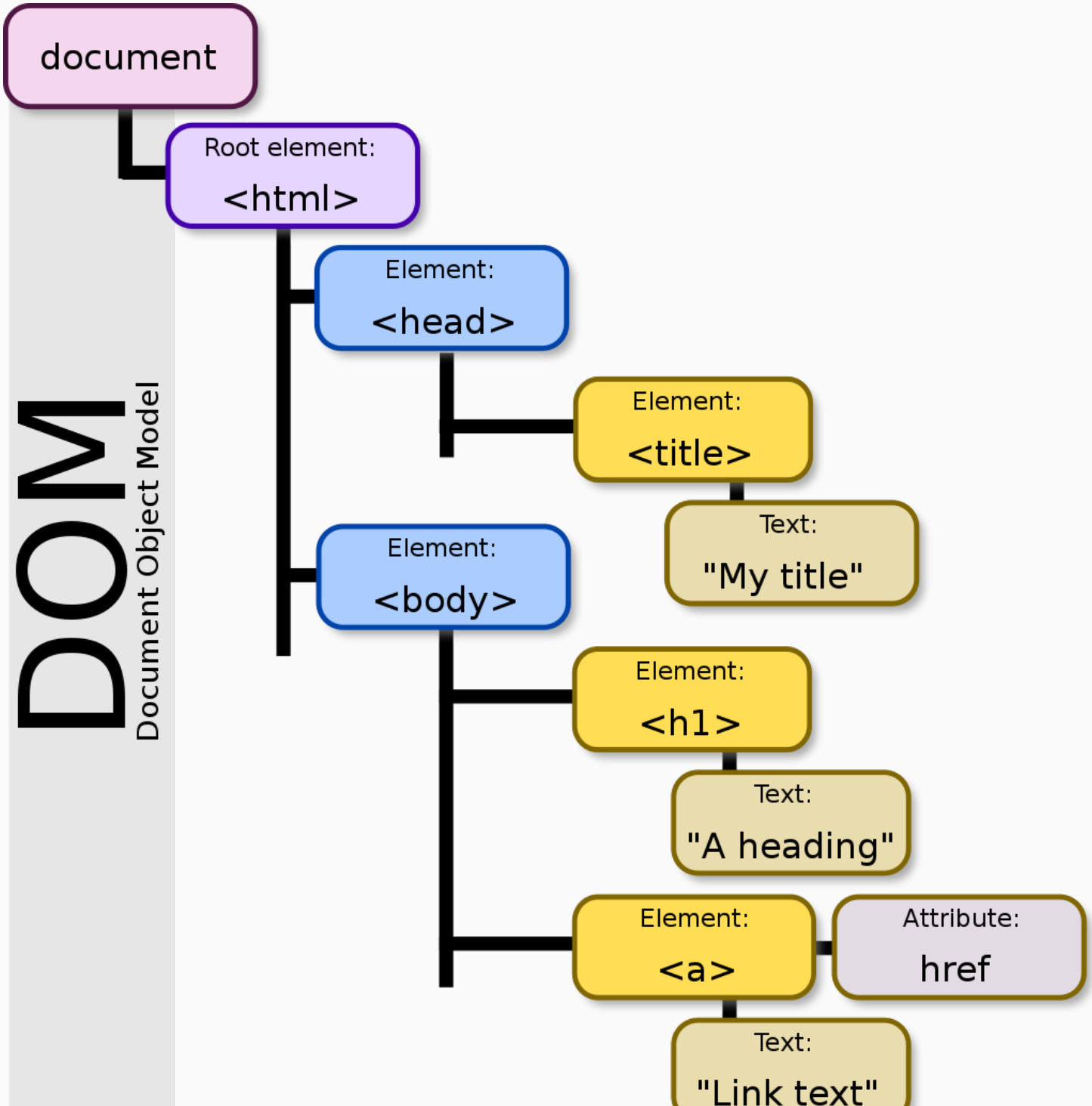
- tags are opened () and closed ()
- tags are nested
- hence, we can represent them as a tree

It's also called a DOM-tree (Document Object Model)



*Go back to your browser window inspecting our example page. Can you see this back?*

https://en.wikipedia.org/wiki/Document_Object_Model



We can use an XPATH to denote a position in the tree ("how to traverse the tree, starting from the root").

Alternatively, we can use CSS Selectors to select specific tags and/or attributes

# Back to our example

We now have much better tools!

```
from lxml.html import fromstring

tree = fromstring(htmlsource)

# instead of re.findall(r"<h3>(.*?)</h3>", htmlsource)
# we now have two easier options:
print([e.text_content() for e in tree.xpath("//h3")])
print([e.text_content() for e in tree.cssselect("h3")])

# Note that e is an element with many methods and properties.
# Try out grabbing the first one and use TAB completion:
test = tree.cssselect("h3")[0]
test.#press TAB
```

Let's look at an overview of the syntax: https://cssbook.net/chapter12.html#tab:cssselect

# CSS Selector vs XPATH

Two equivalent examples:

```
# we extract all relevant elements using their XPATH
elements = tree.xpath('//div[@class="restaurant"]')

# alternatively, we can use their CSS selector:
elements2 =  tree.cssselect("div.restaurants")

assert elements==elements2
```

If you want to use CSS selectors, you may need to `pip install cssselect` first

# CSS Selector vs XPATH

- partly a matter of personal preferences
- Table 12.1 in the book shows both
- CSS selectors are often easier to write (and more modern)
- XPATHs are more straight-forward for describing the hierarchical position of an object
- there are some cases that cannot be described as CSS selector (in particular, arbiraty attributes)

$\Rightarrow$ Many people us CSS selectors by default and restort to XPATHs if necessary

# Scaling up

# But this was on *one* page only, right?

Next step: Repeat for each relevant page.

**Possibility 1: Based on url schemes**

If the url of one review page is
https://www.hostelworld.com/hosteldetails.php/ClinkNOORD/Amsterdam/93919/reviews?page=2

...then the next one is probably?

⇒ you can construct a list of all possible URLs:

```
MAXPAGES = 20
baseurl = 'https://www.hostelworld.com/hosteldetails.php/ClinkNOORD/
    Amsterdam/93919/reviews?page='
allurls = [f"baseurl{i}" for i in range(1,MAXPAGES+1)]
```

# But this was on *one* page only, right?

Next step: Repeat for each relevant page.

**Possibility 1: Based on url schemes**

If the url of one review page is
https://www.hostelworld.com/hosteldetails.php/ClinkNOORD/Amsterdam/93919/reviews?page=2

...then the next one is probably?

⇒ you can construct a list of all possible URLs:

```
MAXPAGES = 20
baseurl = 'https://www.hostelworld.com/hosteldetails.php/ClinkNOORD/
    Amsterdam/93919/reviews?page='
allurls = [f"baseurl{i}" for i in range(1,MAXPAGES+1)]
```

# But this was on *one* page only, right?

Next step: Repeat for each relevant page.

**Possibility 2: Based on XPATHs or CSS Selectors**

Use XPATH to get the url of the next page (i.e., to get the link that you would click to get the next review)

# Recap

## General idea

1. Identify each element by its XPATH or CSS Selector (look it up in your browser)
2. Read the webpage into a (looooooong) string
3. Use the XPATH or CSS Selectors to extract the relevant text into a list (with a module like lxml)
4. Do something with the list (preprocess, analyze, save)
5. Repeat

# General remarks

**There is often more than one way to specify an XPATH or CSS Selector**

1. It's about finding a description that is not too general (e.g., each H3) tag but also not too specific (the second H3 tag nested in the first div nested in. . . )
2. (a bit like the precision-recall trade-off we discussed)
3. Look into the structure of the HTML code, for example with "Inspect Element" and use that information to play around with different possibilities

Let's look at the example at
https://github.com/uvacw/teaching-bdaca/blob/main/modules/scraping/scraping.ipynb

# Scaling up

## Avanced techniques

We could talk endlessly about all the following techniques, but it's probably better to just look at them once you need them. But a quick overview is always good!

See especially Chapter 12.3: Authentication, Cookies, and Sessions

# Dynamic vs static pages

- Modern pages often use techniques like JavaScript to load or refresh content – in this case, the HTML response you got via `requests` does not contain the final content
- (it cannot – as you do not use a browser that could run the JavaScript)
- Solution: Literally use a browser (via Selenium)

# Which browser am I (pretending to be)?

- When a HTTP request is made, it contains a header with meta-information
- One is the "User-Agent" (name and version of sender)
- You may want to set this specifically to pretend to be a specific browser (maybe even a mobile one!)

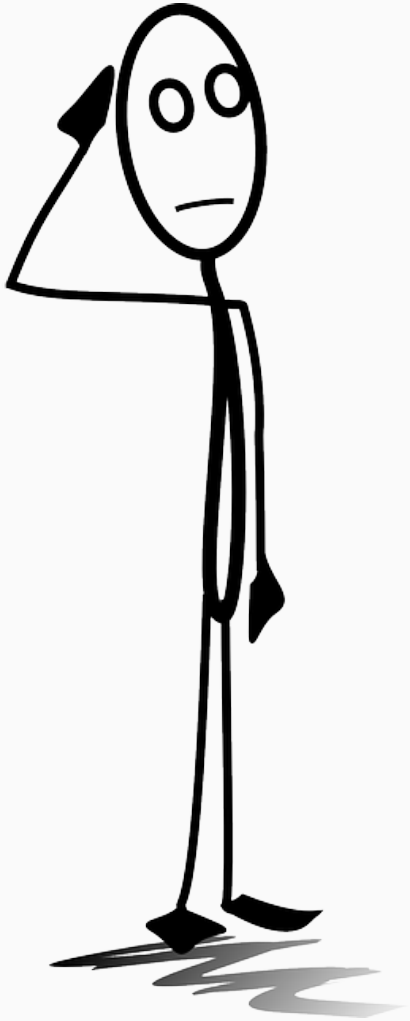
# Sessions and Cookies

- Sometimes, it may be necessary to explicitly store and/or set cookies
- You can do this with *Sessions* and *Cookie Jars* in `requests`
- Or you let your browser handle it via Selenium

# You don't always have to start from scratch!

- For some tasks, there are also packages that provide some generic scrapers – for instance, `trafilatura` for news sites.
- By definition, these cannot work perfectly for all sites – but you can start with it and then write specific scrapers for the sites were it doesn't work!

https://trafilatura.readthedocs.io/en/latest/



*Any questions?*

# Next steps

You will write your own scraper. Prepare by choosing a website that you want to scrape. It is advisable to select a site without things like cookiewalls or logins.

https://github.com/uvacw/teaching-bdaca/blob/main/12ec-course/week10/exercises/