Week 7: Scraping Webpages

MY472: Data for Data Scientists
https://lse-my472.github.io/

Autumn Term 2025

Ryan Hübert
Associate Professor of Methodology

Slides last updated: 11 November 2025

Unstructured data from the web

Last lecture covered ways to get *structured* data from servers

- HTTP requests to download existing files
- APIs to return data in JSON format
- Connecting to cloud databases to download data

But a lot of data is available on the web in an *unstructured* format

- Data that is not tabular, very untidy, or fragmented across multiple sources

A very common example: data that is available on **webpages**

- Note terminology: webpage, website and (web)server

What is webscraping?

Webscraping: automated extraction of useful data from webpages

- Sometimes called **screen scraping** since it entails extracting data usually displayed on a (browser) screen

Why would you want to do this?

- Goal: transform extracted data to structured dataset
- Usually tabular, but also other possibilities

Entails: making requests for webpage data and parsing that data to extract information

The rules of the game

1. Respect the hosting site's wishes
   - ▷ Check if an API exists or if data is available for download
   - ▷ Respect copyright and ethics; what are you allowed to do?
   - ▷ Keep in mind where data comes from and give credit
   - ▷ Some websites disallow scrapers via `robots.txt` file
2. Limit your bandwidth use
   - ▷ Wait some time after each request ("polite delays")
   - ▷ Scrape only what you need, and just once
3. Read documentation
   - ▷ Is there a batch download option?
   - ▷ Are there any rate limits?
   - ▷ Can you share the data?

Outline

1. Source code for websites
2. Extracting data from a webpage
3. Webscraping in R
4. Webscraping ethics

Source code for websites

A server returns a combination of text and multimedia files (images, videos, etc.) that are used to display a website

Each “webpage” is typically a plain text file coded in a combination of languages: HTML, CSS and JavaScript

These plain text files contain instructions about how the webpage should look

- We refer to these files as the **source code** for the webpage

The purpose of a web browser (Chrome, Safari, Firefox, etc.) is to render source code so that a user sees something “nice”

- Can view/analyse source code in most browsers

Source code for websites

Hypertext Markup Language (HTML) is a markup language

- It defines structure and meaning of content on webpages
- It tells a browser what elements are on the page and how they're organized
- It only provides basic, static content

Cascading Style Sheets (CSS) is a style sheet language

- It describes formatting of HTML components

JavaScript is a programming language

- Adds dynamic behavior and interactivity to webpages
- Can modify content, structure, or style after page has loaded

HTML

An **HTML document** contains **elements**, which are the objects that make up a webpage

- E.g., text, images, links, scripts

Elements are written in the source code using **HTML tags**, which are pieces of markup enclosed in angle brackets and arranged in a tree-like structure

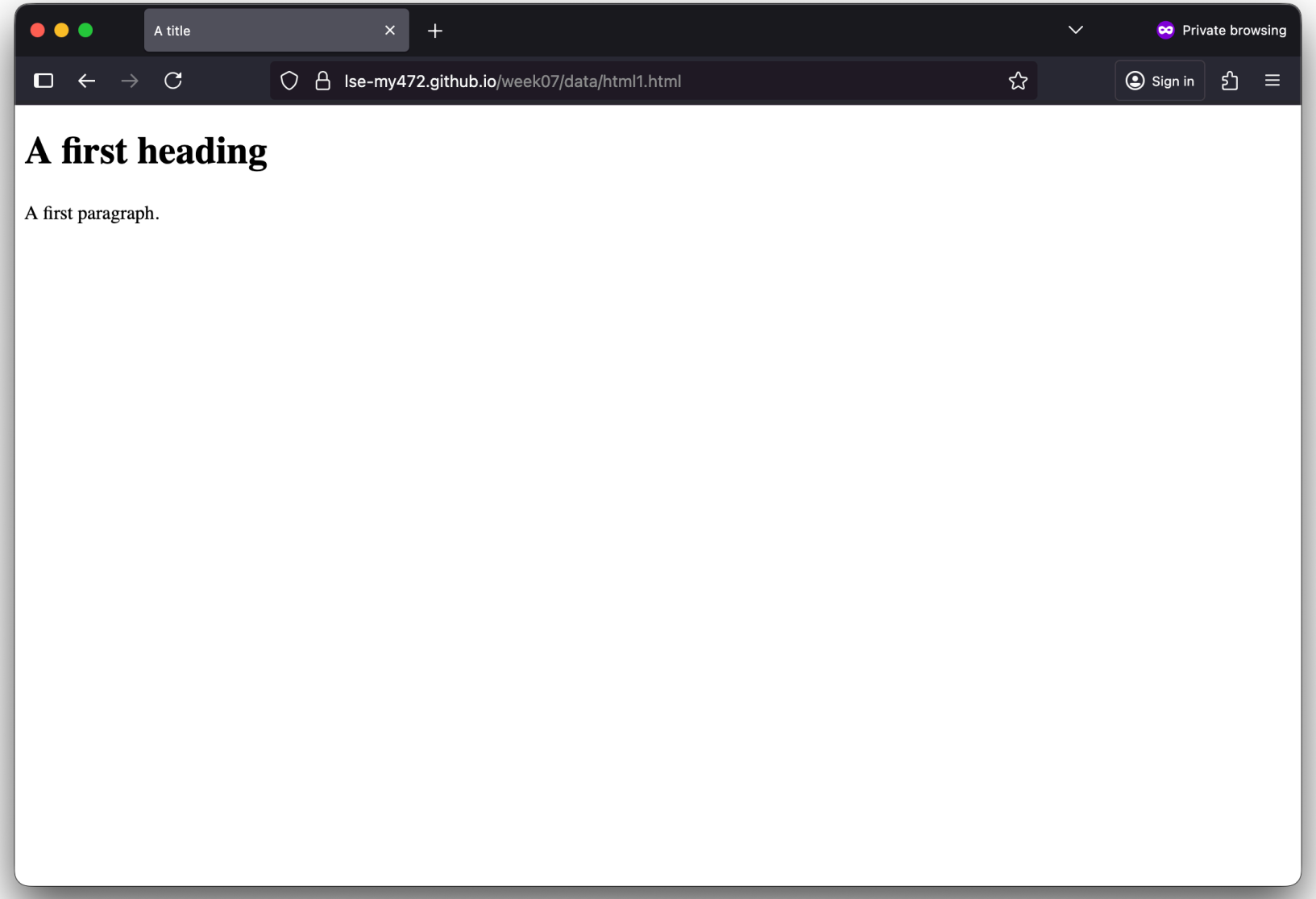
- Most elements need to be enclosed between an **opening tag** and a **closing tag**
- E.g., paragraph text like `<p>Hello world!</p>`
- **Void elements** are self-contained and do not have closing tags, such as `<img>`, `<link>` and `<br>`

A very simple HTML file

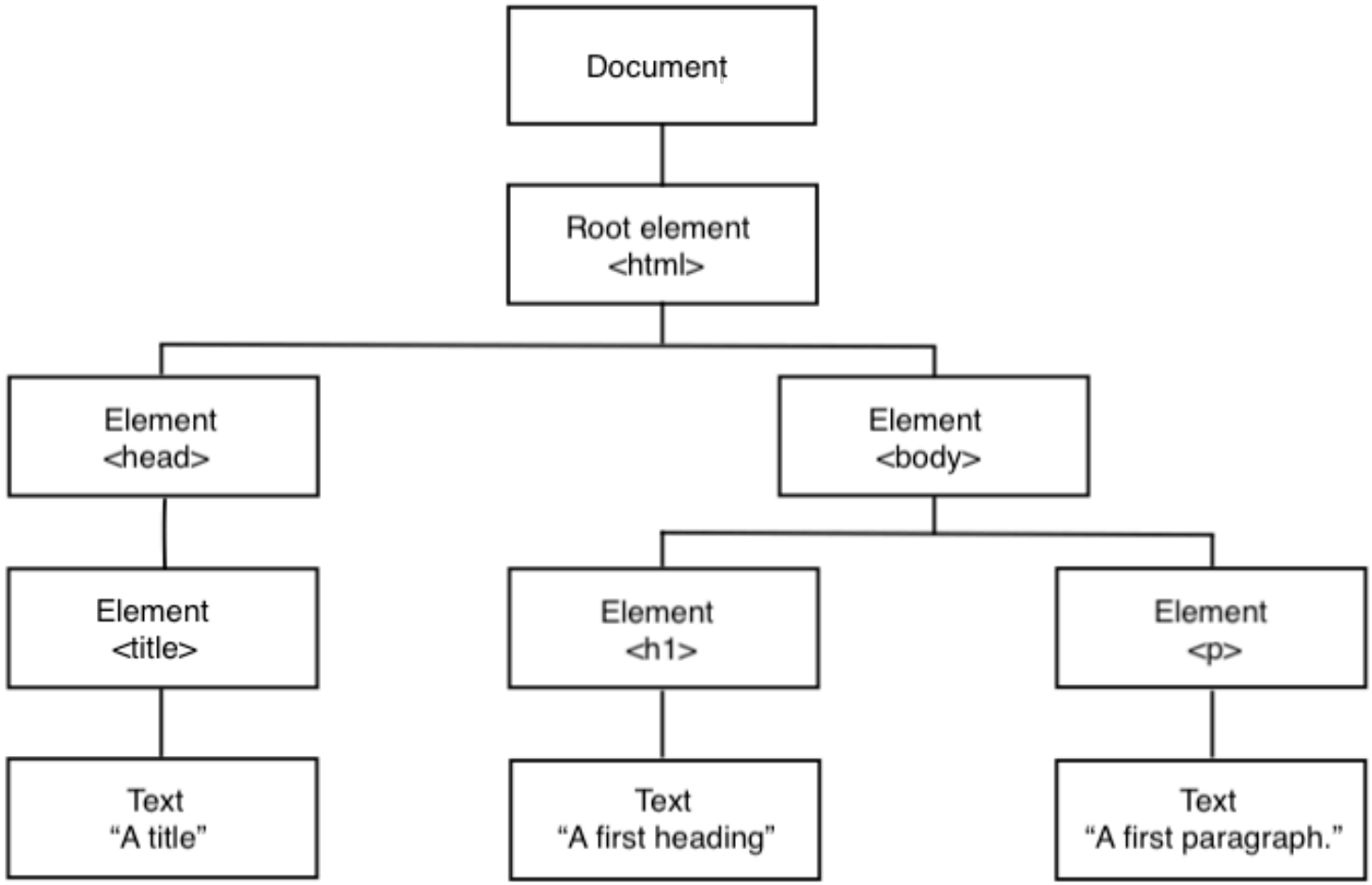
```
<!DOCTYPE html>
<html>
    <head>
        <title>A title</title>
    </head>
    <body>
        <h1>A first heading</h1>
        <p>A first paragraph.</p>
    </body>
</html>
```

Source: https://lse-my472.github.io/week07/data/html1.html

A very simple HTML file



HTML tree structure



HTML tree structure

```
<!DOCTYPE html>
<html>
    <head>
        <title>A title</title>
    </head>
    <body>
        <h1>A first heading</h1>
        <p>A first paragraph.</p>
    </body>
</html>
```

Key vocabulary (similar to file paths):

- `<html>` is *the* **root element** of this HTML document
- `<head>` is *the* **parent element** for `<title>`
- `<h1>` is *a* **child element** for `<body>`
- `<head>` and `<p>` are both **descendant elements** of `<html>`

HTML attributes

HTML tags can contain **attributes** in them

- Attributes provide additional detail about how the tag should be treated by a browser

Some common attribute examples:

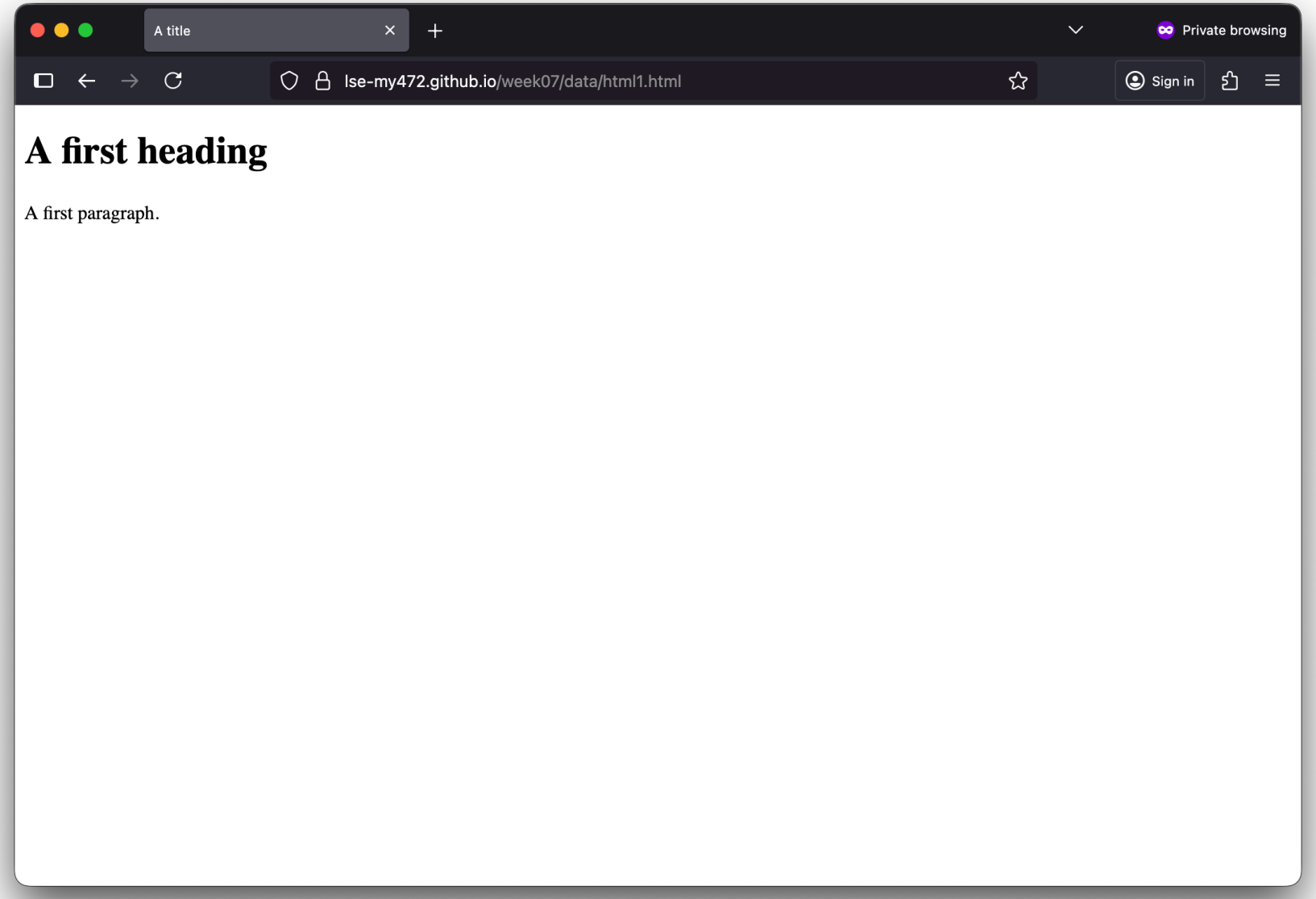
- `<a href="https://www.google.com/">Link</a>`
- `<img src="site_logo.png">`
- `<div class="my_class">Some content here.</div>`
- `<p id="firstParagraph">My first paragraph.</p>`

Slightly more features

```
<!DOCTYPE html>
<html>
    <head>
        <title>A title</title>
    </head>
    <body>
        <h1>A first heading</h1>
        <p>A first paragraph.</p>
        <p>A second paragraph with some
            <b>formatted</b> text.</p>
        <p>A third paragraph with a
            <a href="http://www.lse.ac.uk">hyperlink</a> and
            an image <img src="/assets/lse-square.png">.</p>
    </body>
</html>
```

Source: https://lse-my472.github.io/week07/data/html2.html

Slightly more features

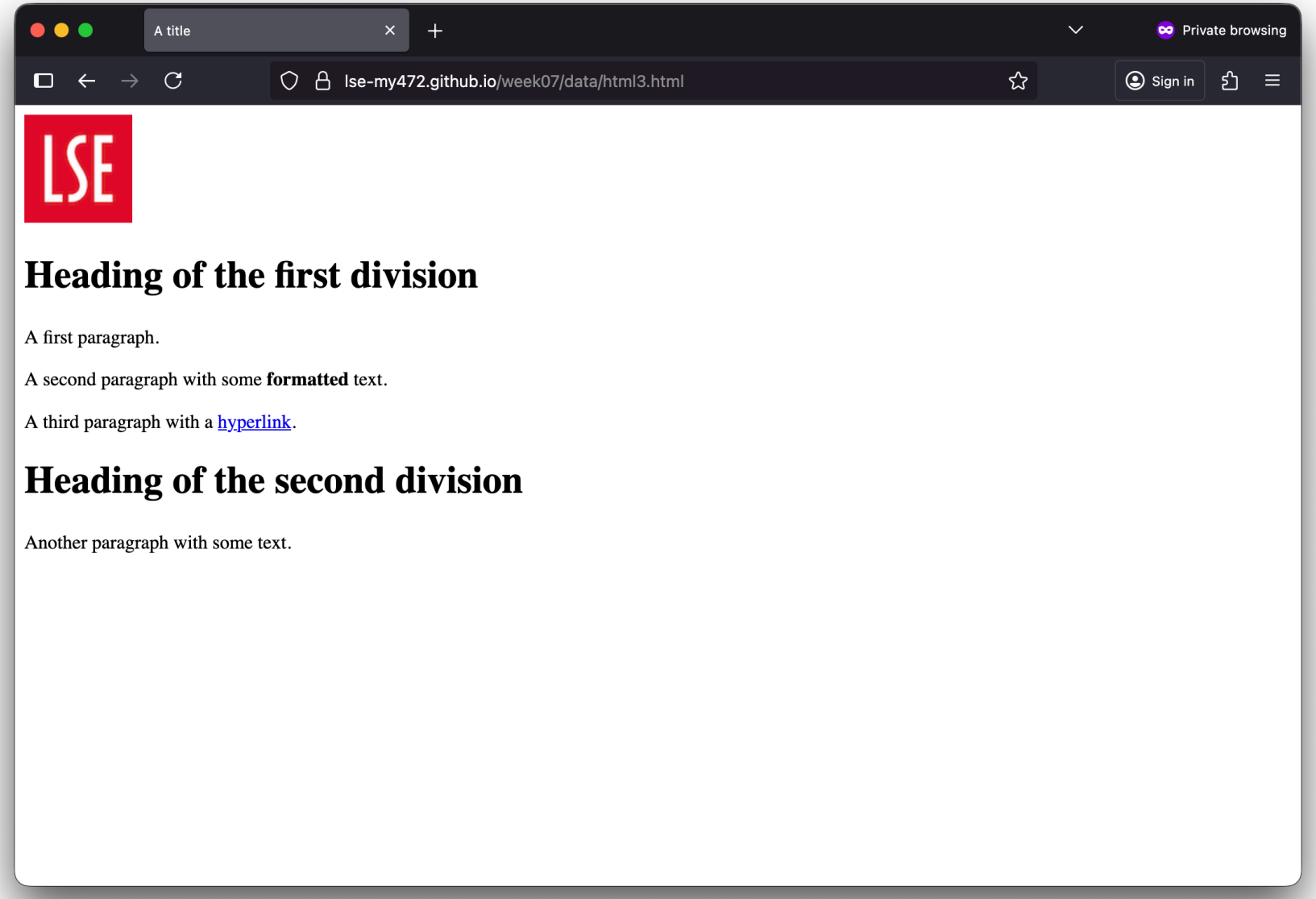


With some content divisions

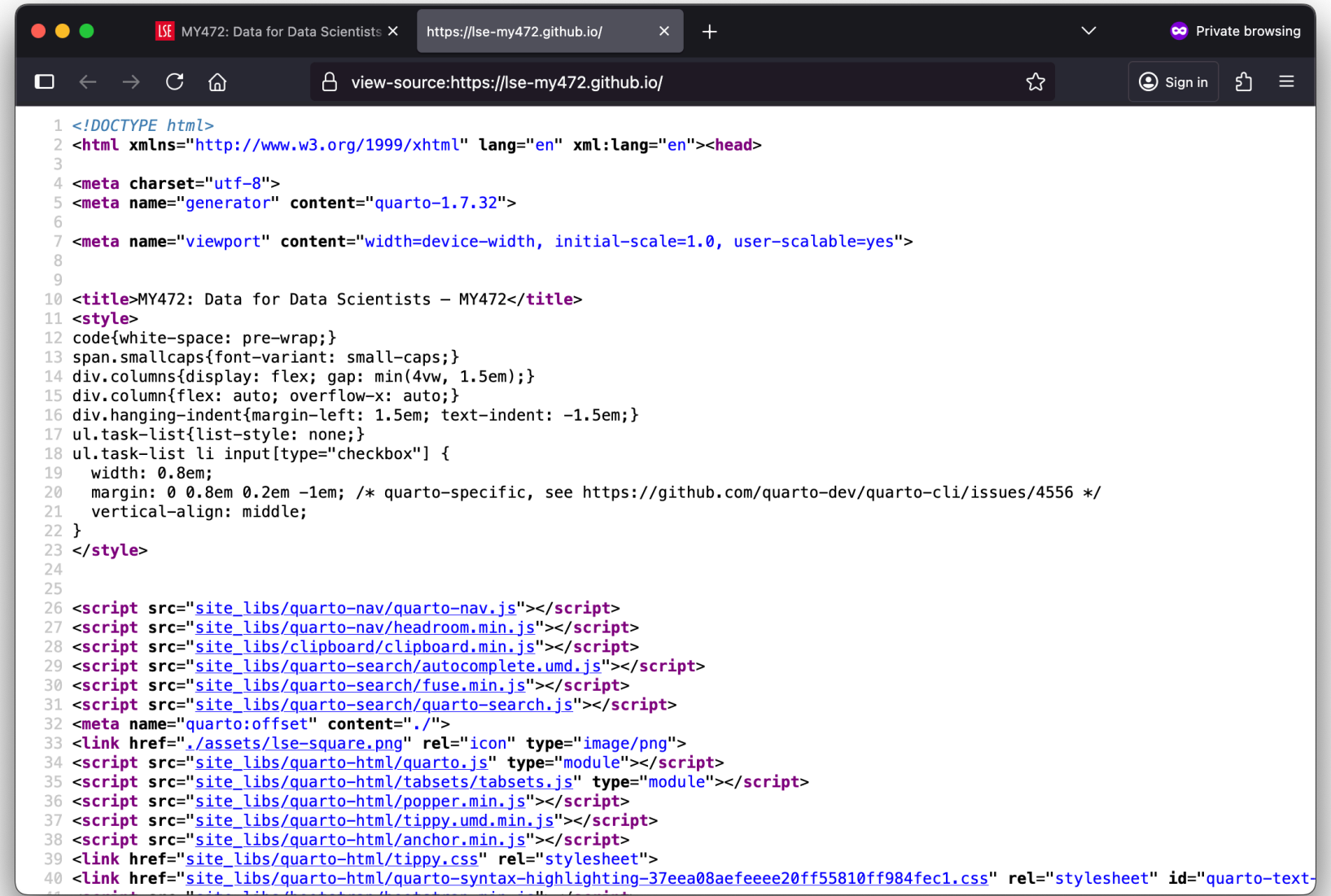
```
<!DOCTYPE html>
<html>
    <head>
        <title>A title</title>
    </head>
    <body>
        <div>
            <h1>Heading of the first division</h1>
            <p>A first paragraph.</p>
            <p>A second paragraph with some
                <b>formatted</b> text.</p>
            <p>A third paragraph with a
                <a href="http://www.lse.ac.uk">hyperlink</a>.</p>
        </div>
        <div>
            <h1>Heading of the second division</h1>
            <p>Another paragraph with some text.</p>
        </div>
    </body>
</html>
```

Source: https://lse-my472.github.io/week07/data/html3.html

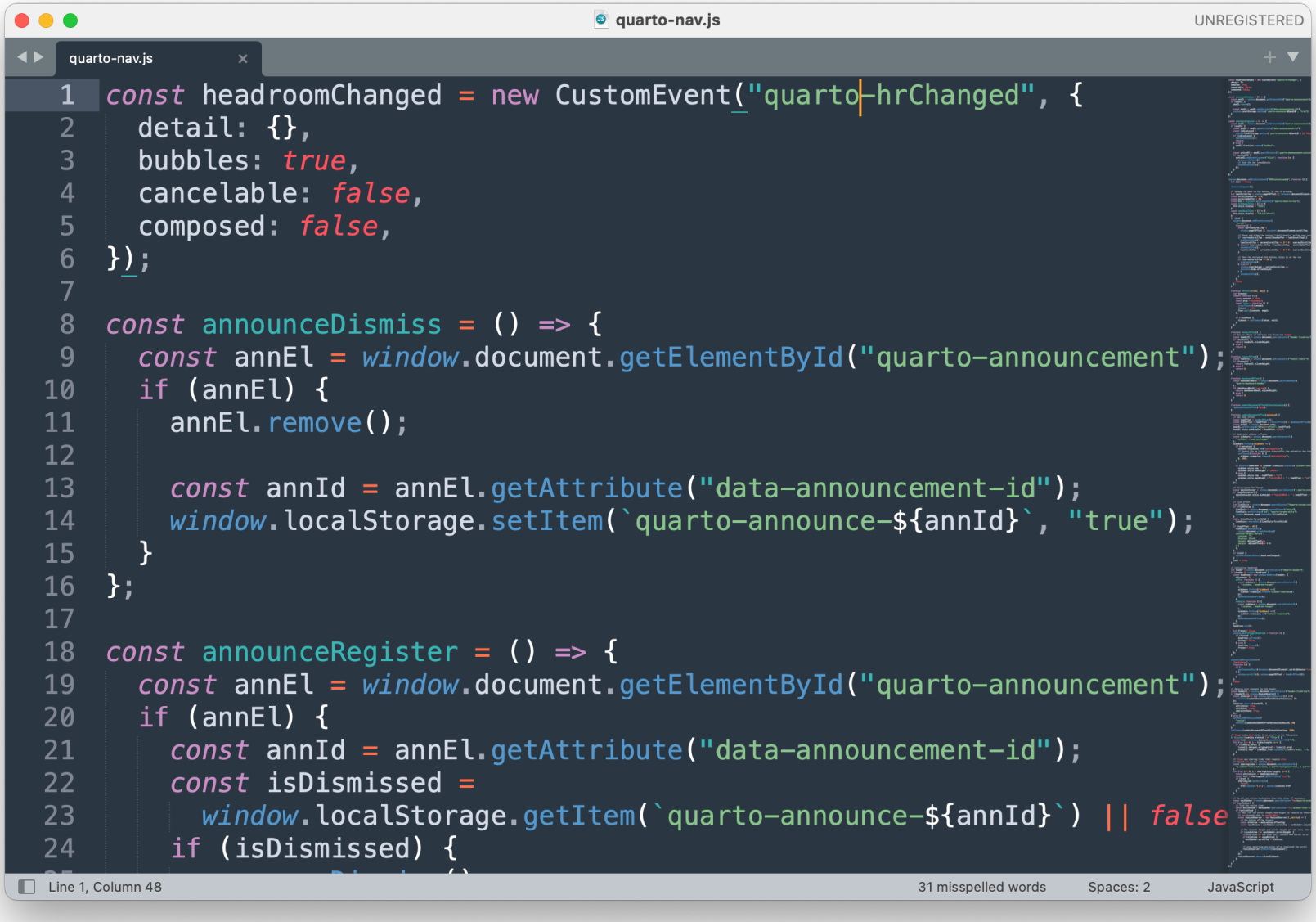
With some content divisions



What do CSS and JavaScript look like?



What do CSS and JavaScript look like?



Outline

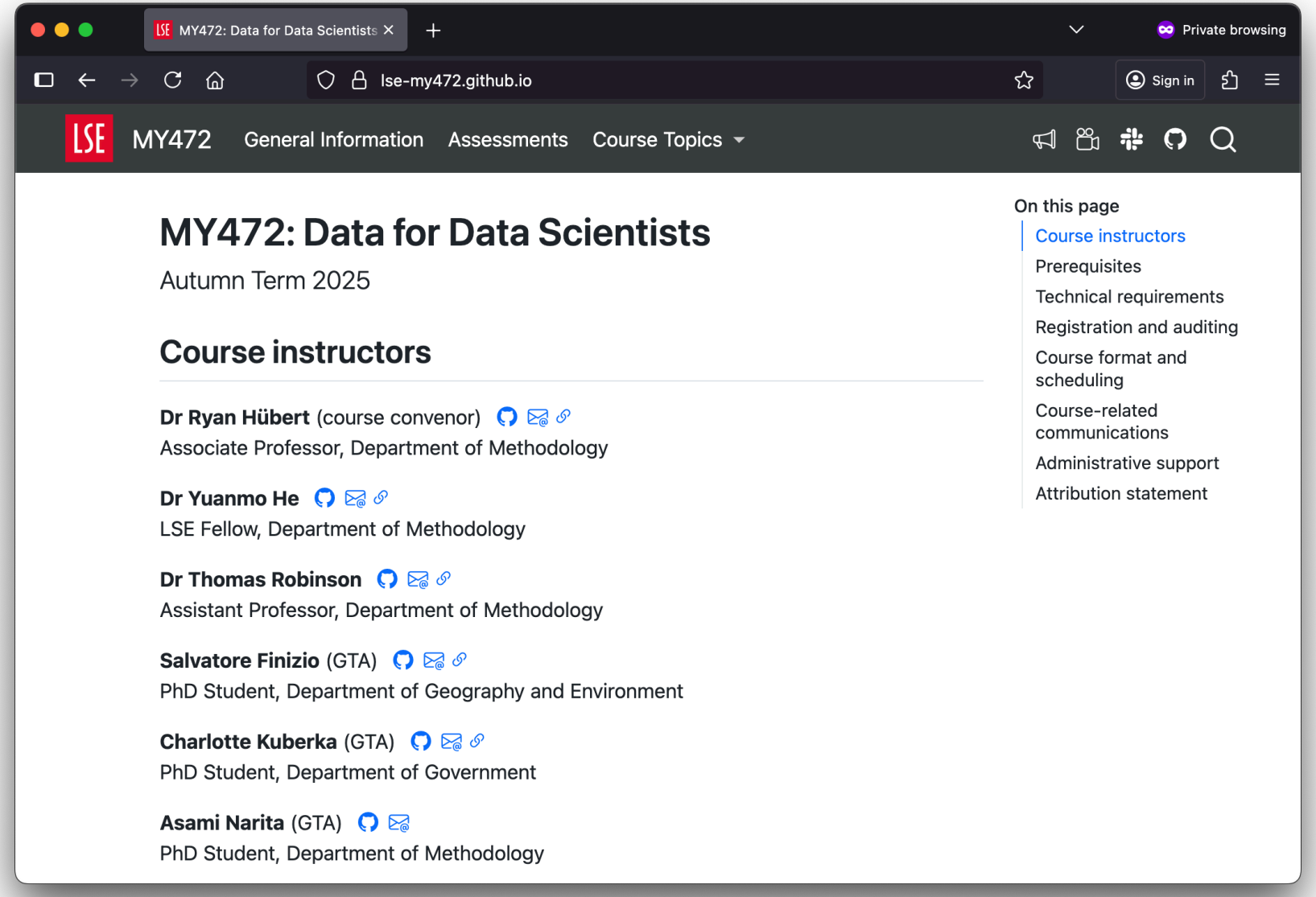
1 Source code for websites

2 Extracting data from a webpage

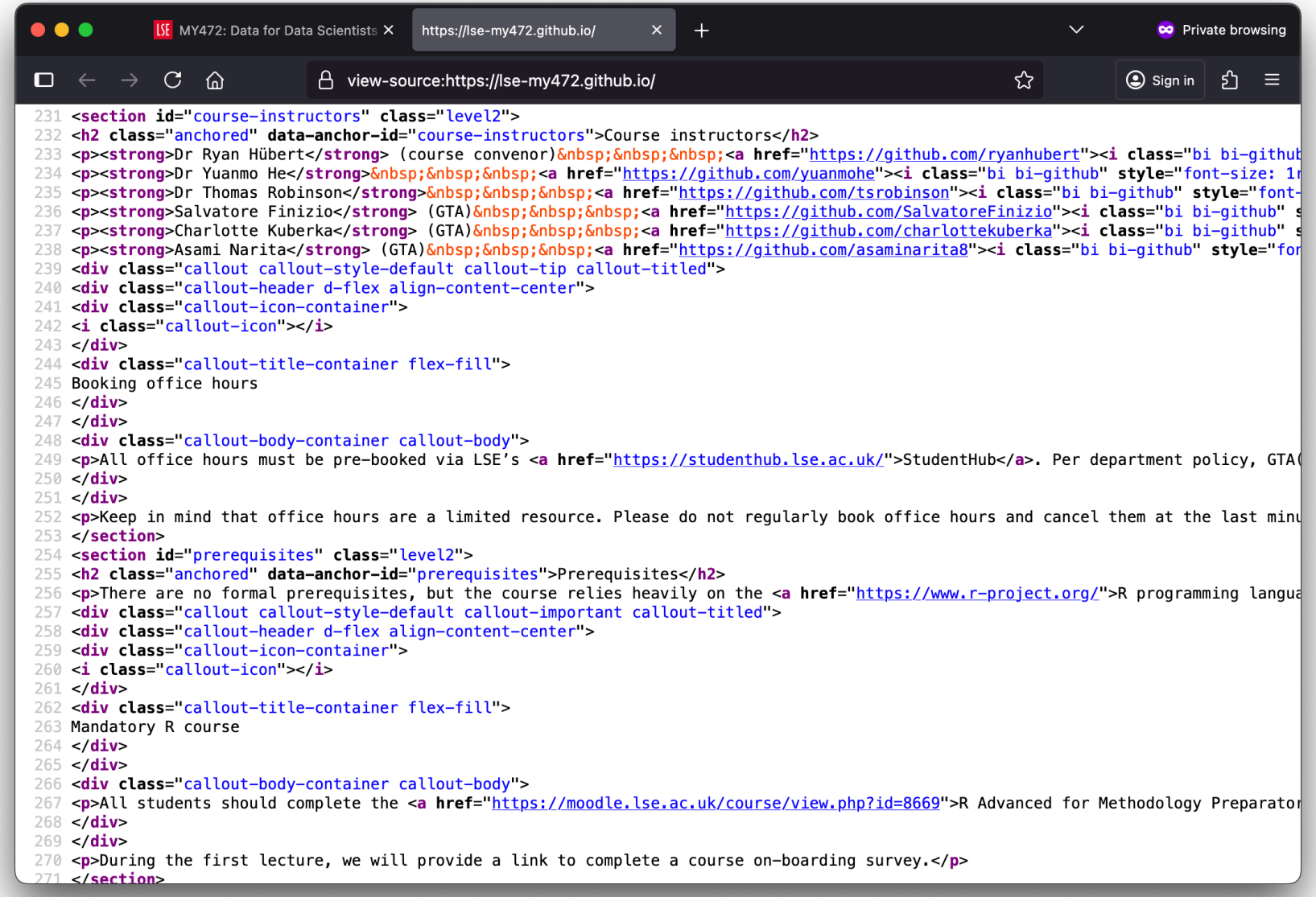
3 Webscraping in R

4 Webscraping ethics

What do you “scrape”?



Is the desired content in the source code?



Three different scenarios

Static websites

- Server sends “fixed” HTML files
- Client renders the HTML file in a browser

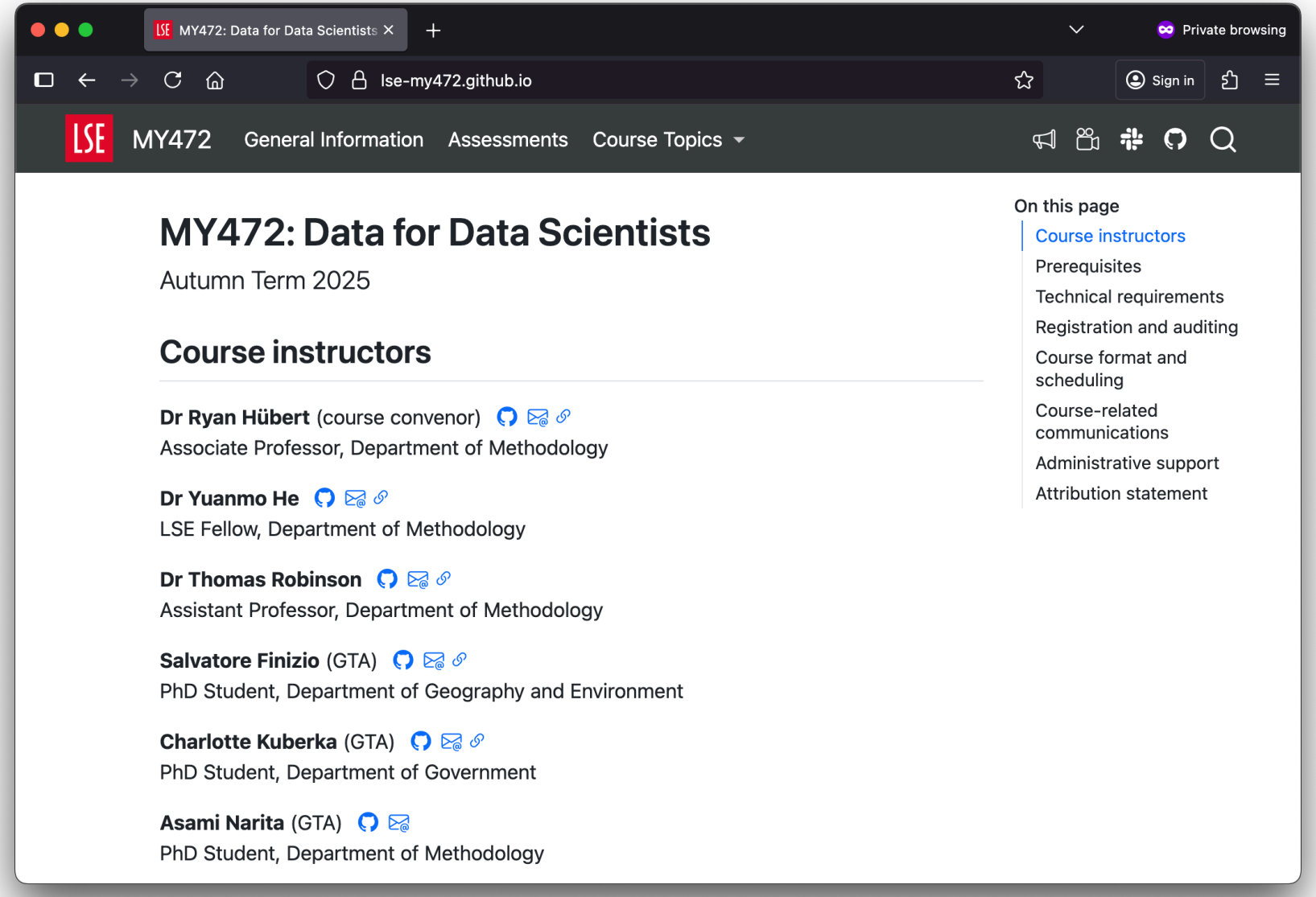
Dynamic websites

- Server or client (via JavaScript) builds the HTML on the fly
- Client browser still ends up receiving and displaying HTML
- Enables integration of real-time data (from a database or API)

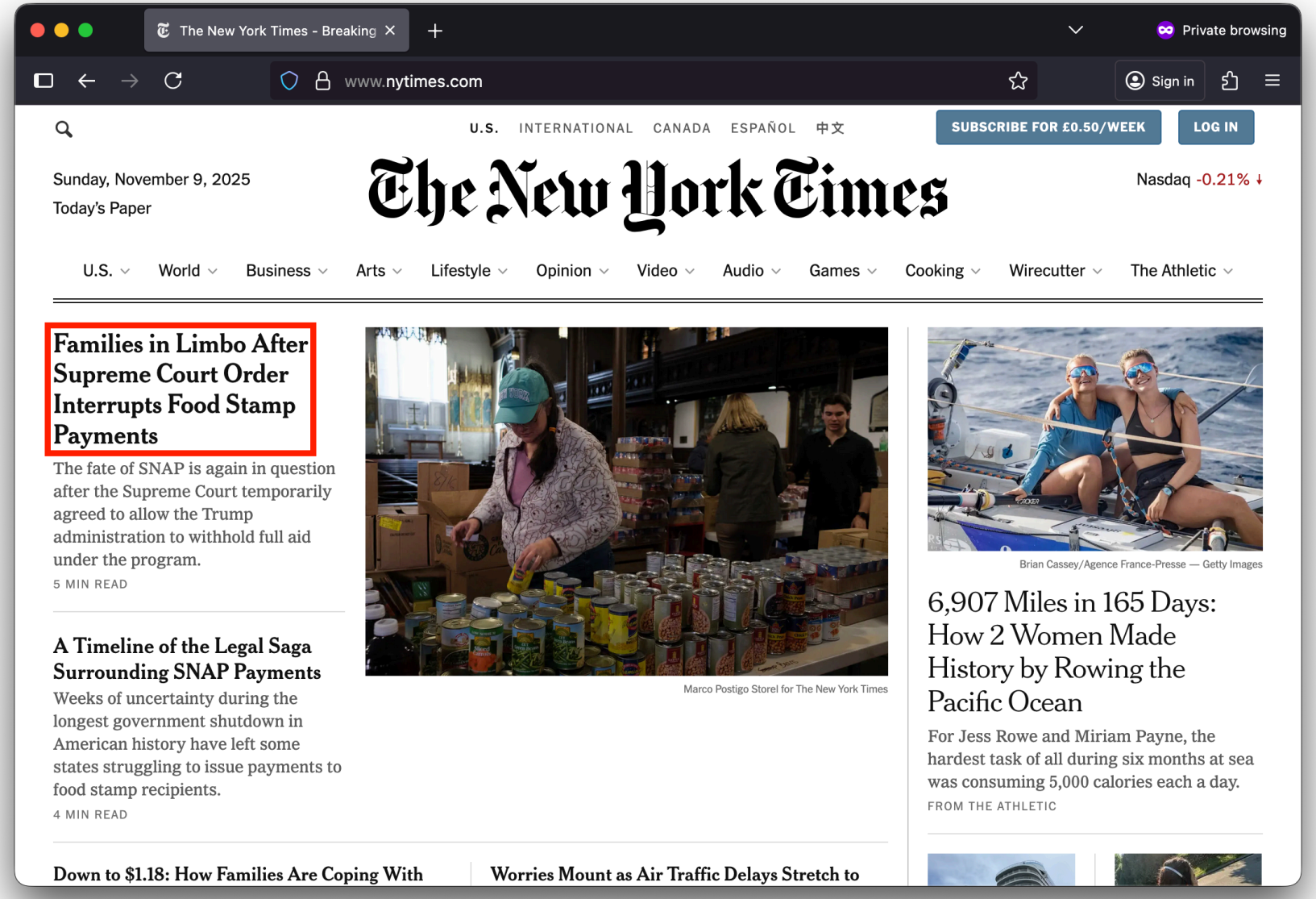
Single-page apps (or SPAs)

- Client loads a minimal HTML skeleton
- Embedded JavaScript injects more HTML dynamically

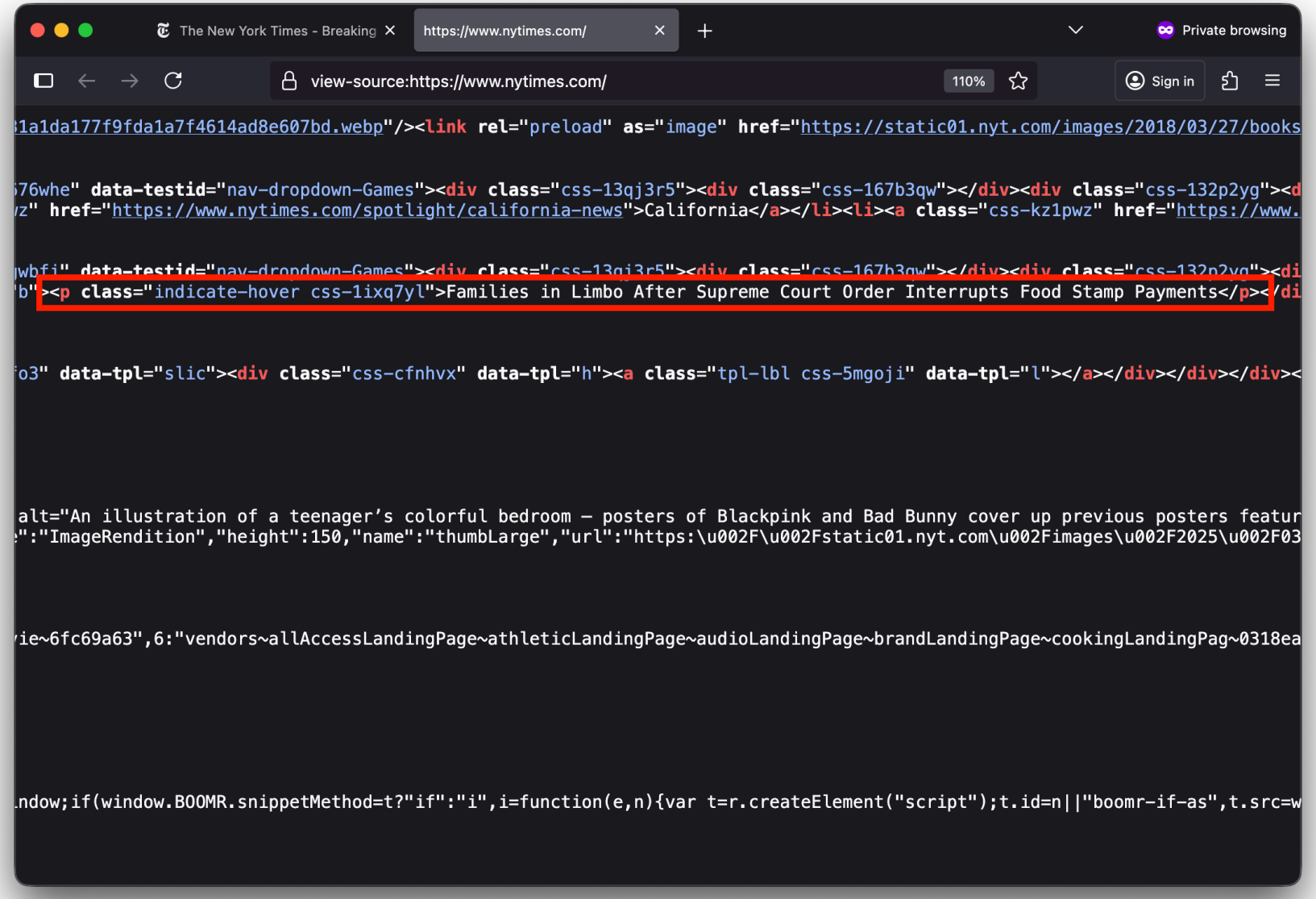
Three different scenarios



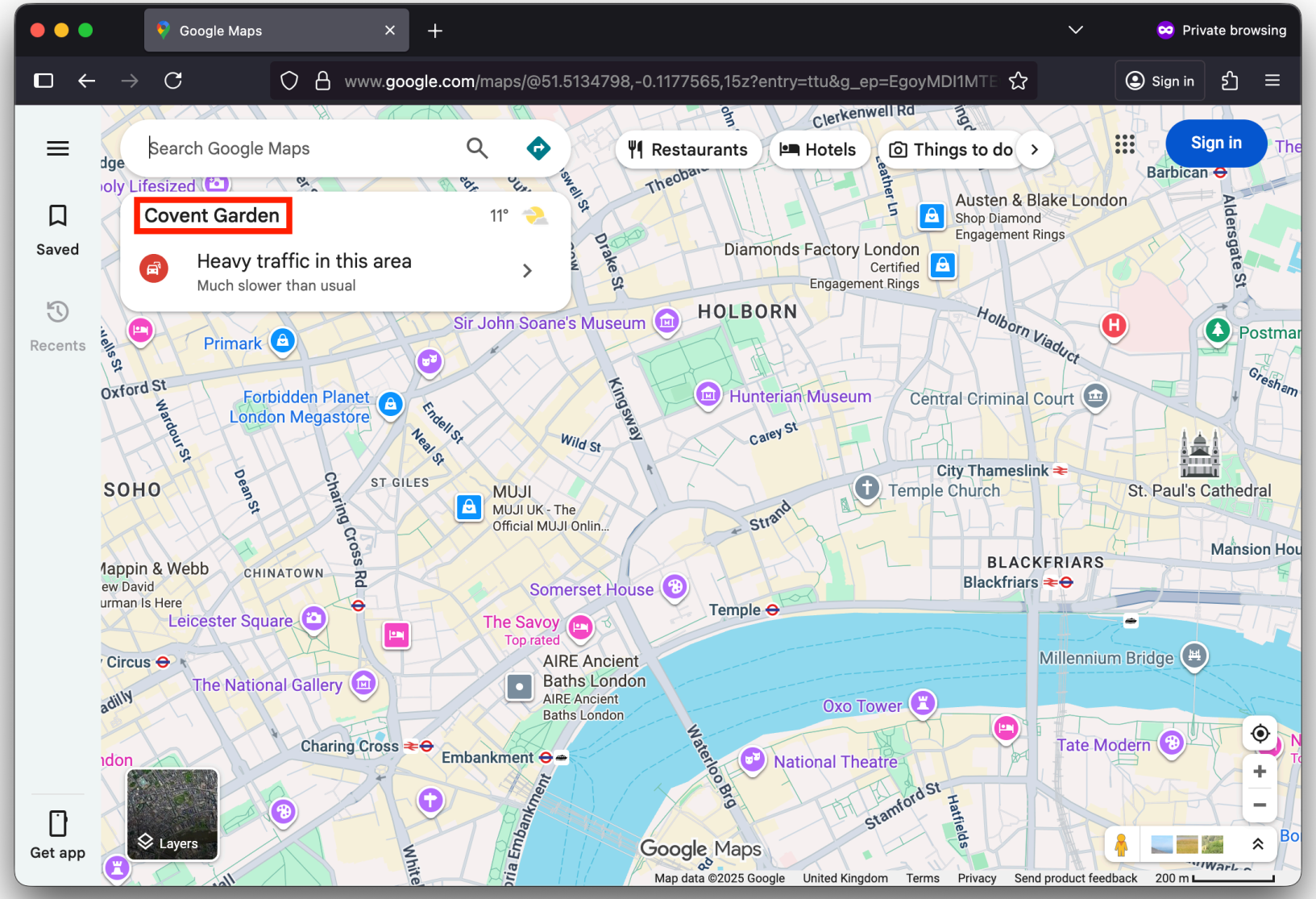
Three different scenarios



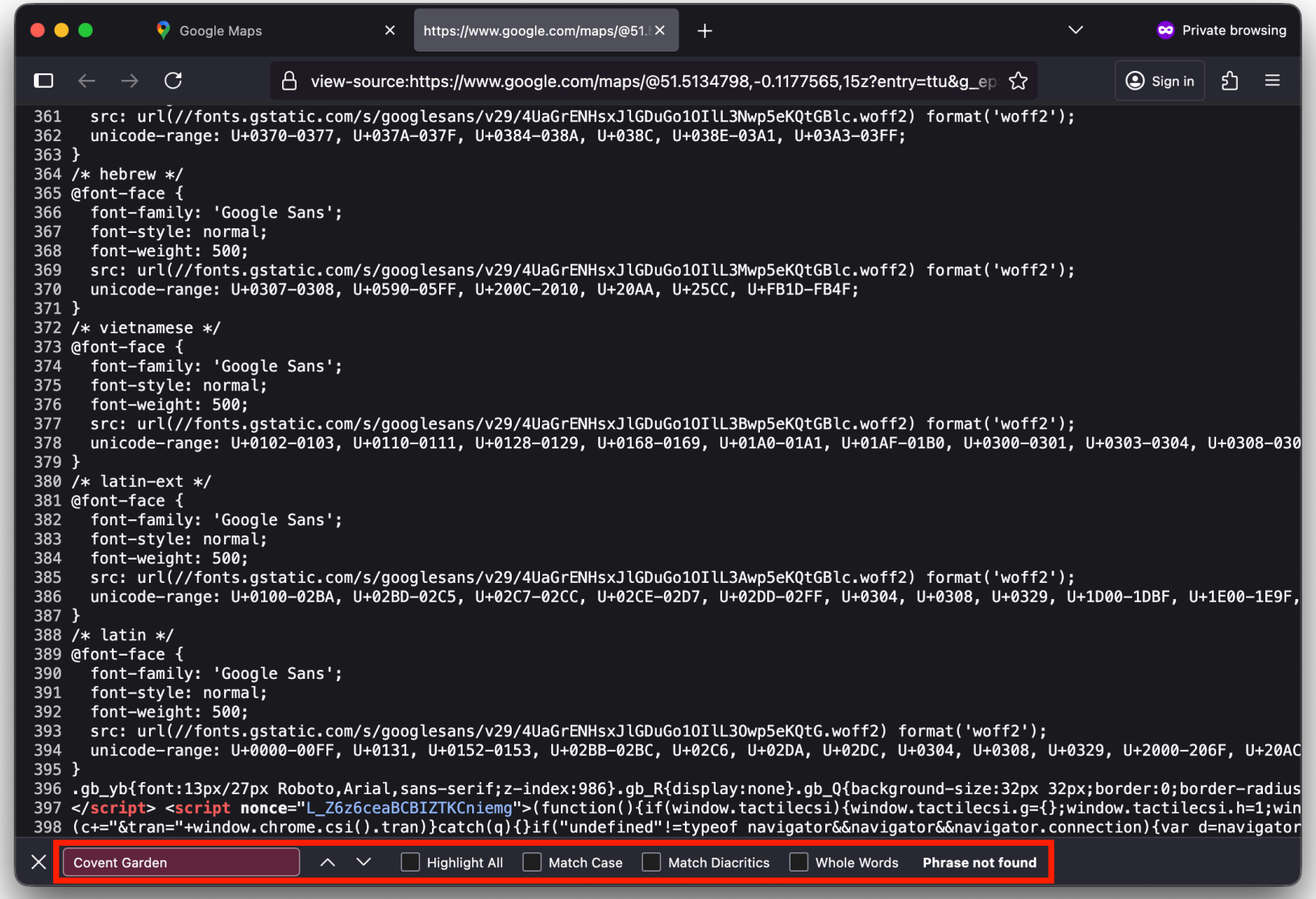
Three different scenarios



Three different scenarios



Three different scenarios



Query languages for webscraping

Typical use case for webscraping: get information from webpages

- Requires that you be able to find and extract relevant info
- Ideally: use structure of HTML to identify key elements

There are two main **query languages** used to identify elements

- **CSS selectors** (read more here)
- **XPath** (or **XML Path Language**, read more here)

Analogy between regex and query languages

For most things, you can use either CSS selectors or XPath

- But XPath is (probably) more flexible/powerful

Identifying elements by tag

```
<h3>This is the main item</h3>
```

These get this element:

- CSS Selector: `h3`
- XPath: `//h3`

This is most useful if there is only *one* `<h3>` tag

XPath resembles file paths, but note:

- An **absolute XPath** begins with one slash, e.g., `/html`
- A **relative XPath** begins with two slashes, e.g. `//h3`

Identifying elements using tree structure

```
<div>
  <p><a href="https://www.google.com">Google</a></p>
</div>
```

These get the paragraph element:

- CSS Selector: `div > p`
- XPath: `//div/p`

These get the link element:

- CSS Selector: `div a`
- XPath: `//div//a`

Key distinction: extracting *direct* child versus *any* child—so, neither `div > a` nor `//div/a` will get the link!

Identifying elements by an index

```
<body>
  <h2>Title</h2>
  <p>First paragraph.</p>
  <p>Second paragraph.</p>
</body>
```

These get the *second* paragraph element:

- CSS Selector: `body > p:nth-of-type(2)`
- XPath: `//body/p[2]`

These get the *first* paragraph element:

- CSS Selector: `body > p:nth-child(2)`
- XPath: `//body/*[2]`

Identifying elements by an attribute

```
<div id="maintitle">My main title</div>
<div class="itemdisplay">This is the main item.</div>
<p class="itemdisplay">This is a subsidiary item.</p>
```

These get the first `div` element:

- CSS Selector: `div#maintitle` or `#maintitle`
- XPath: `//div[@id='maintitle']` or `//*[@id='maintitle']`

Identifying elements by class or id attributes

```
<div id="maintitle">My main title</div>
<div class="itemdisplay">This is the main item.</div>
<p class="itemdisplay">This is a subsidiary item.</p>
```

These get the second div element:

- CSS Selector: `div.itemdisplay`
- XPath:`//div[@class='itemdisplay']`

These get both the second div and the paragraph elements:

- CSS Selector: `.itemdisplay`
- XPath: `//*[@class='itemdisplay']`

Selecting tags by other attributes

```
<link href = "css1.css" id = "some_id">
<link href = "css2.css">
```

These get both links:

- CSS Selector: `link` or `link[href]`
- XPath: `//link` or `//link[@href]`

These get the first link:

- CSS Selector: `link[id]` or `link[id="some_id"]` or `link[href="css1.css"]` or `link[id*="some"]`
- XPath: `//link[@id]` or `//link[@id="some_id"]` or `//link[@href="css1.css"]` or `//link[contains(@id, 'some')]`

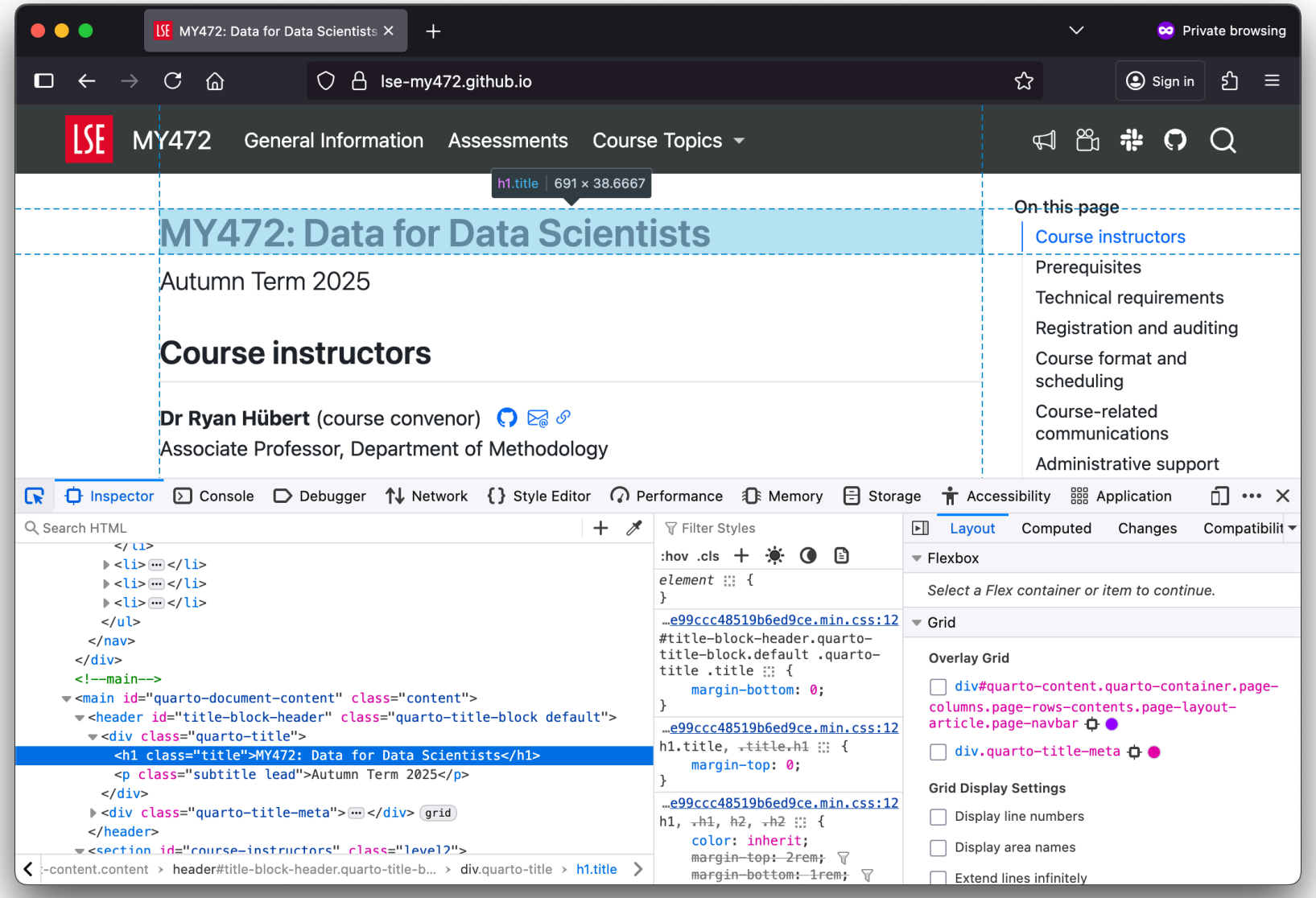
Selecting tags by displayed text

```
<div>
  <h1>Heading of the first division</h1>
</div>
<div>
  <h1>Heading of the second division</h1>
</div>
```

These get the first `h1` element:

- CSS Selector: `h1:contains("Heading of the first division")`
- XPath: `//h1[text()="Heading of the first division"]`

Figuring out your query by inspecting elements



Outline

Webscraping in R

Webscraping in R entails:

1. Making requests for webpages
   - ▷ Simple `GET` requests for webpages: `{httr}` package
   - ▷ Automated browser requests: `{RSelenium}` package
2. Parsing html using `{rvest}` package
   - ▷ Convert raw HTML to tree object using `read_html()`
   - ▷ Find elements using `html_elements()`
   - ▷ Extract text/tables: `html_text()`, `html_tables()`

Note: `{httr}` and `{rvest}` are tightly integrated

- ▷ If you supply a URL as the argument for `read_html()` it will make `GET` request then convert raw HTML from response

Two webscraping approaches

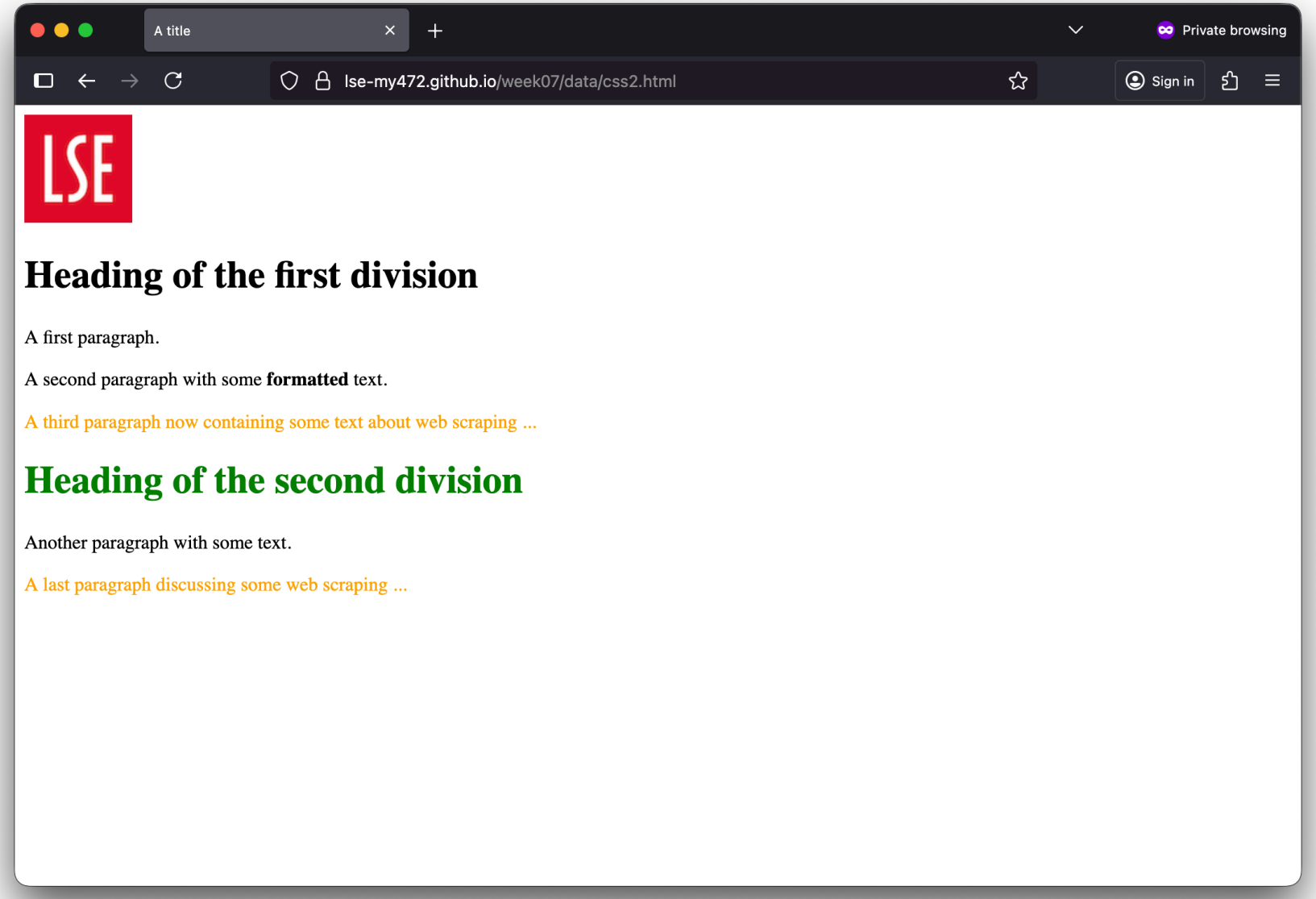
Simple `GET` requests work for:

- Static HTML pages
- Dynamic HTML pages that are rendered server-side
- Pages requiring “simple” interaction

Automated browser is needed for:

- Pages with client-side rendering (e.g., JavaScript)
- Pages with infinite scroll
- Pages requiring user login or script-based interaction
- Sites with “malfunctioning” bot detection

A simple example



A simple example

```
library("rvest")
url <- "https://lse-my472.github.io/week07/data/css2.html"

html <- read_html(url)
```

```
{html_document}
<html>
[1] <head>\n<meta http-equiv="Content-Type" content="text/h ...
[2] <body>\n        <div>\n            <img src="/assets/ls ...
```

```
html |>
  html_elements(css = "div > p:nth-of-type(3)") |>
  html_text()
```

```
[1] "A third paragraph now containing some text about web s ..."
```

A simple example

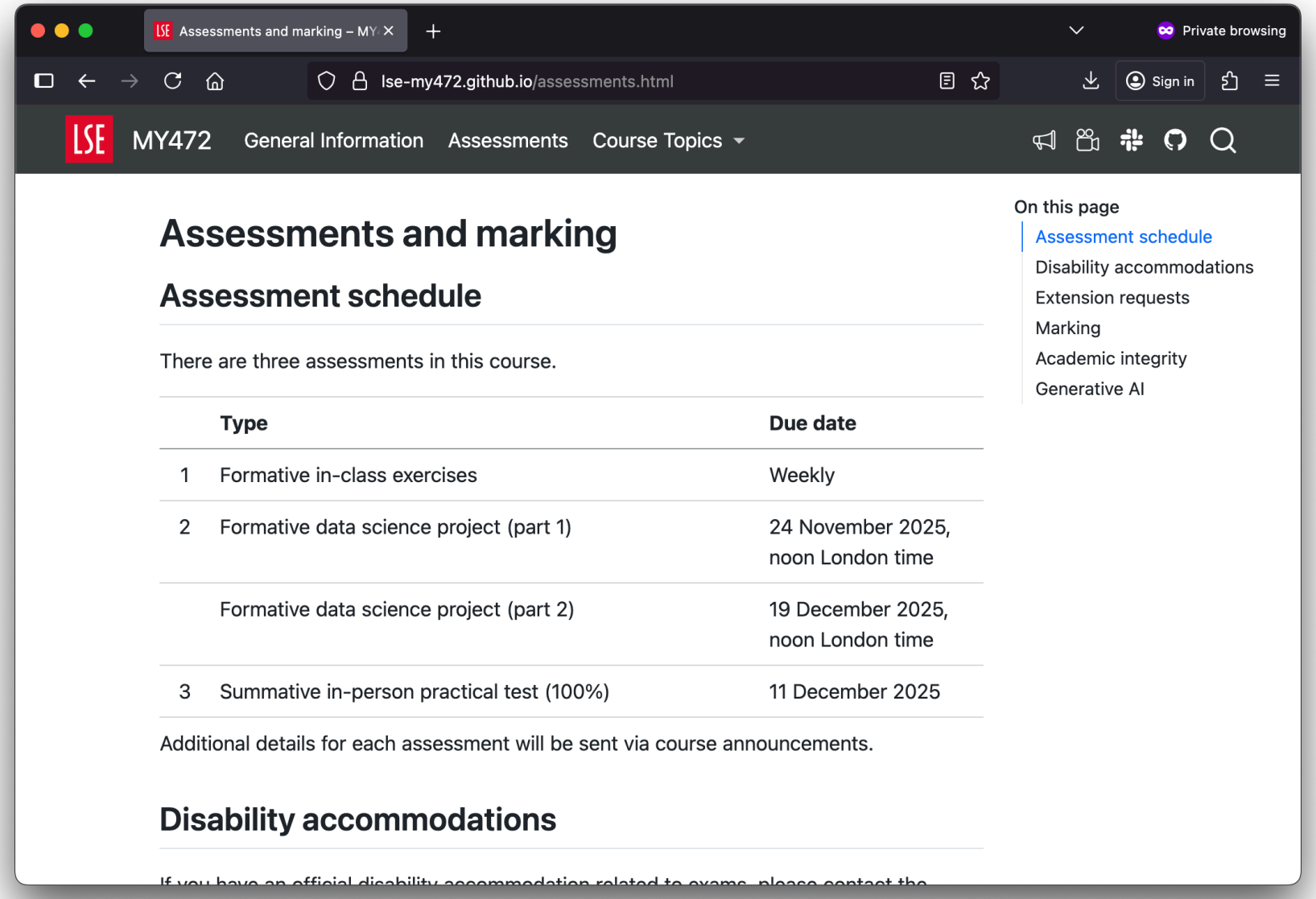
```
library("rvest")
url <- "https://lse-my472.github.io/week07/data/css2.html"

html <- read_html(url)
```

```
{html_document}
<html>
[1] <head>\n<meta http-equiv="Content-Type" content="text/h ...
[2] <body>\n        <div>\n            <img src="/assets/ls ...
```

```
html |>
  html_elements(xpath = "//div/p[2]") |>
  html_text()
```

```
[1] "A third paragraph now containing some text about web s ..."
```

Extracting tables from webpages



Extracting tables from webpages

```r
# This will GET and parse HTML in one step
t <- read_html("https://lse-my472.github.io/assessments.html")
Sys.sleep(0.5) # a polite sleep
print(t)
```

```
{html_document}
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
[1] <head>\n<meta http-equiv="Content-Type" content="text ...
[2] <body class="nav-fixed quarto-light">\n\n<div id="qua ...
```

Extracting tables from webpages

The `html_table()` function returns a list of tables found in the HTML based on HTML table syntax (read more about this syntax)

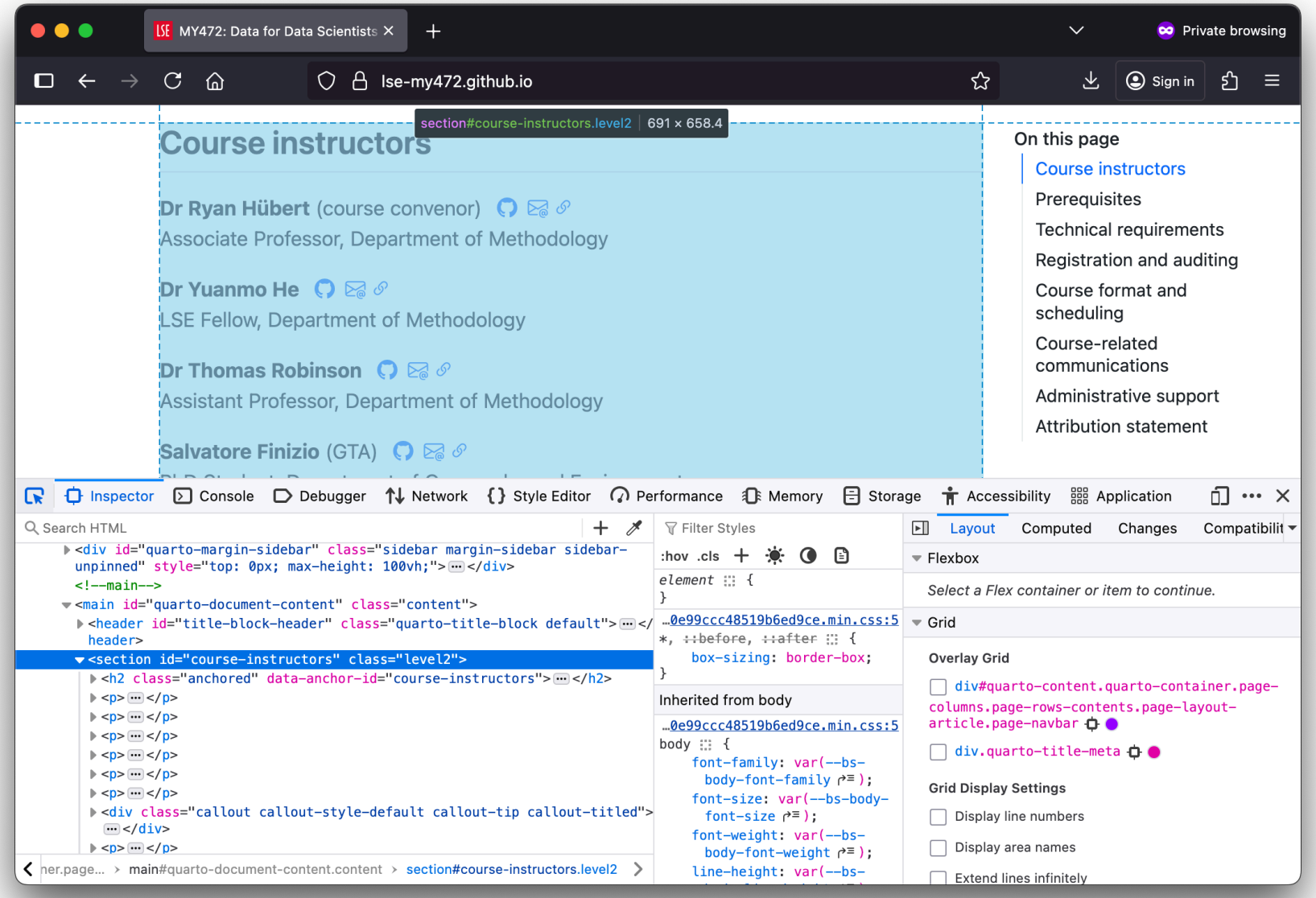
```
tables <- html_table(t)
length(tables)    # only 1 table
```

```
[1] 1
```

```
af <- tables[[1]] # assessments info
print(af)
```

```
# A tibble: 4 x 3
     `` Type                                      `Due date`
  <int> <chr>                                     <chr>
1     1 Formative in-class exercises              Weekly
2     2 Formative data science project (part 1)   24 Novemb~
3    NA Formative data science project (part 2)   19 Decemb~
4     3 Summative in-person practical test (100%) 11 Decemb~
```

Extracting unstructured data from webpages

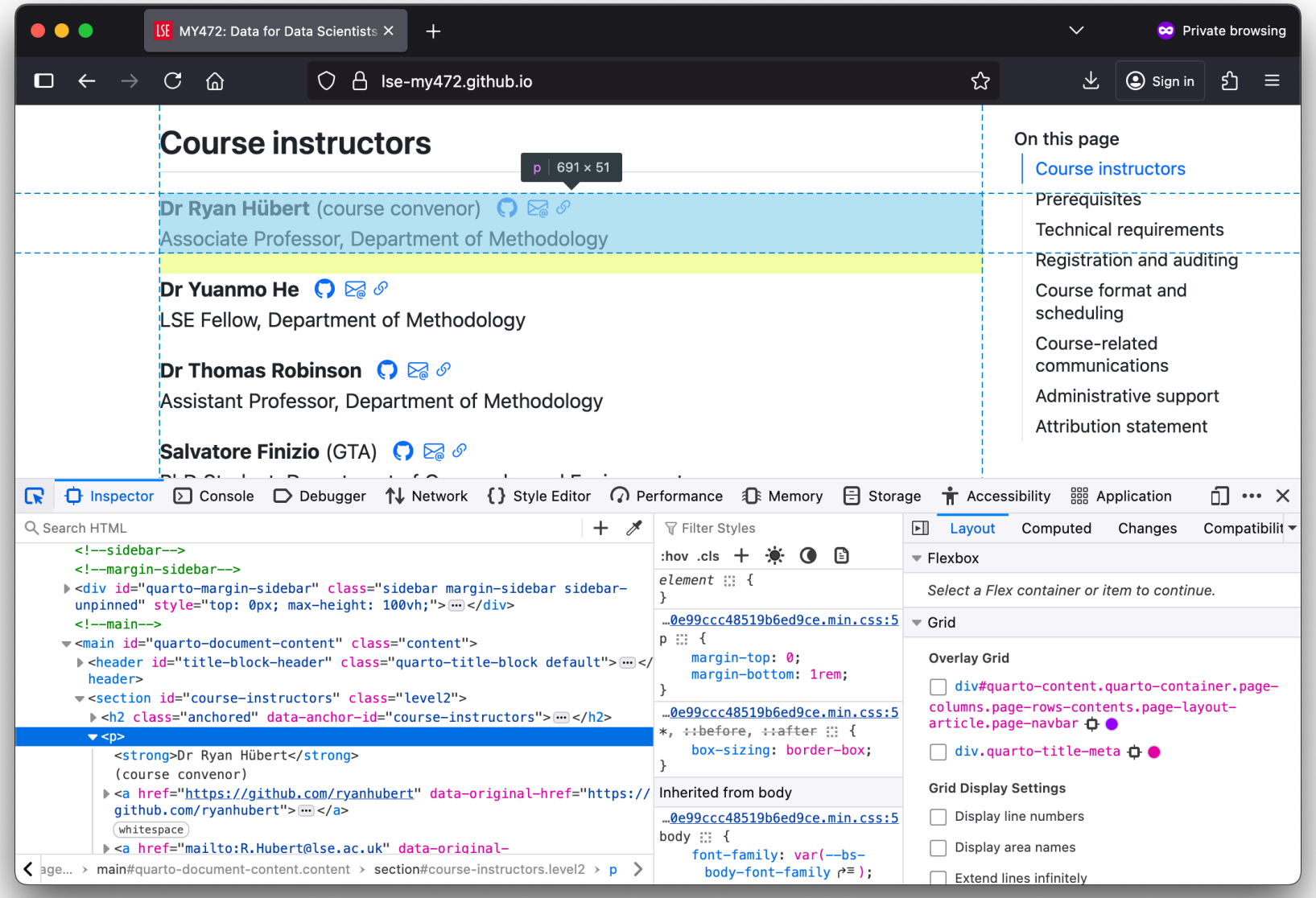


Extracting unstructured data from webpages

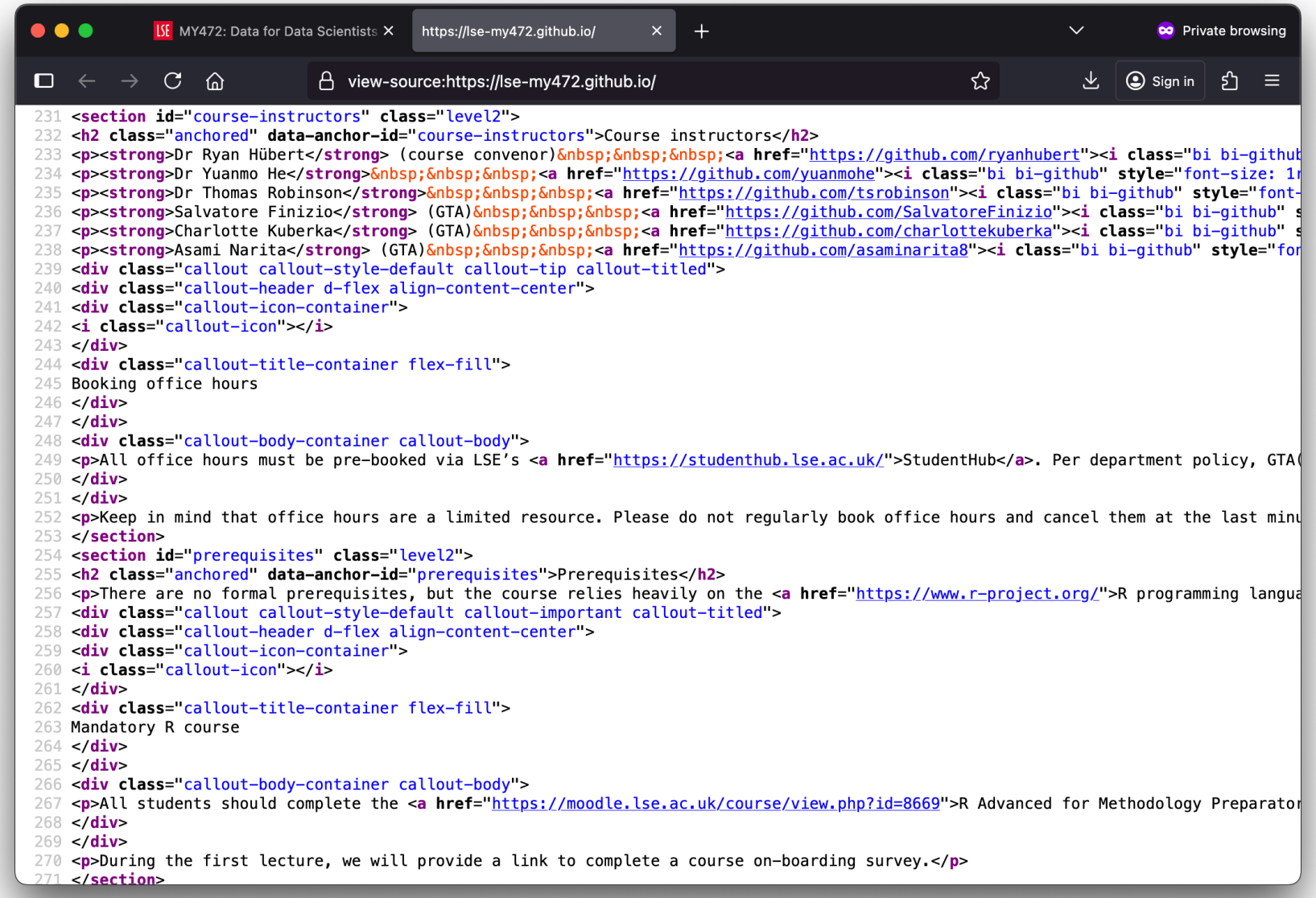
```
ins <- read_html("https://lse-my472.github.io") |>
    html_elements(css = "section#course-instructors")
print(ins)
```

```
{xml_nodeset (1)}
[1] <section id="course-instructors" class="level2"><h2 c ...
```

Extracting unstructured data from webpages



Extracting unstructured data from webpages



Extracting unstructured data from webpages

CSS selectors are like regex: lots of ways to find the same thing

In this case: easiest way to do this is to look for specific text

```
# Get all `p` tags
ins <- html_elements(ins, css = "p")

# Only keep ones containing text "Dr" or "(GTA)"
library("tidyverse")
ins <- ins[str_detect(html_text(ins), "(Dr |[(]GTA[)])")]
print(ins)
```

```
{xml_nodeset (6)}
[1] <p><strong>Dr Ryan Hübert</strong> (course convenor)  ...
[2] <p><strong>Dr Yuanmo He</strong>   <a href="https://g ...
[3] <p><strong>Dr Thomas Robinson</strong>   <a href="htt ...
[4] <p><strong>Salvatore Finizio</strong> (GTA)   <a href ...
[5] <p><strong>Charlotte Kuberka</strong> (GTA)   <a href ...
[6] <p><strong>Asami Narita</strong> (GTA)   <a href="htt ...
```

Extracting unstructured data from webpages

Now let's collect data

```
name <- ins |>
  html_elements(css = "strong") |>
  html_text()
print(name)
```

```
[1] "Dr Ryan Hübert"     "Dr Yuanmo He"
[3] "Dr Thomas Robinson" "Salvatore Finizio"
[5] "Charlotte Kuberka"  "Asami Narita"
```

Extracting unstructured data from webpages

```
title <- ins |>
  html_text() |>
  str_extract("\\s\\s+(.+?),", group = 1)
print(title)
```

```
[1] "Associate Professor" "LSE Fellow"
[3] "Assistant Professor" "PhD Student"
[5] "PhD Student"         "PhD Student"
```

```
dept <- ins |>
  html_text() |>
  str_extract("\\s\\s+.+?, *(.+)$", group = 1)
print(dept)
```

```
[1] "Department of Methodology"
[2] "Department of Methodology"
[3] "Department of Methodology"
[4] "Department of Geography and Environment"
[5] "Department of Government"
[6] "Department of Methodology"
```

Extracting unstructured data from webpages

```
email <- ins |>
  html_elements(css = "a[href*='@lse.ac.uk']") |>
  html_attr("href")
print(email)
```

```
[1] "mailto:R.Hubert@lse.ac.uk"
[2] "mailto:Y.He54@lse.ac.uk"
[3] "mailto:T.Robinson7@lse.ac.uk"
[4] "mailto:S.Finizio@lse.ac.uk"
[5] "mailto:L.Kuberka@lse.ac.uk"
[6] "mailto:A.Narita@lse.ac.uk"
```

```
email <- email |>
    str_replace("mailto[:]", "")
print(email)
```

```
[1] "R.Hubert@lse.ac.uk"    "Y.He54@lse.ac.uk"
[3] "T.Robinson7@lse.ac.uk" "S.Finizio@lse.ac.uk"
[5] "L.Kuberka@lse.ac.uk"   "A.Narita@lse.ac.uk"
```

Extracting unstructured data from webpages

And we can turn this into a nice tibble

```
df <- tibble(name, title, dept, email)
print(df)
```

```
# A tibble: 6 x 4
  name                title                dept                email
  <chr>               <chr>                <chr>               <chr>
1 Dr Ryan Hübert      Associate Professor  Department o~ R.Hu~
2 Dr Yuanmo He        LSE Fellow           Department o~ Y.He~
3 Dr Thomas Robinson  Assistant Professor  Department o~ T.Ro~
4 Salvatore Finizio   PhD Student          Department o~ S.Fi~
5 Charlotte Kuberka   PhD Student          Department o~ L.Ku~
6 Asami Narita        PhD Student          Department o~ A.Na~
```

Dynamic webscraping: a motivational example

In previous years: we scraped data from the U.S. Social Security Administration (at `https://www.ssa.gov/`)

A change in the website has caused this to stop working:

```
library("httr")
ua <- "LSE-MY472/2025 (educational use; contact: R.Hubert@lse.ac.uk)
url <- "https://www.ssa.gov/oact/babynames/numberUSbirths.html"
response <- GET(url, user_agent(ua))
response
```

```
Response [https://www.ssa.gov/oact/babynames/numberUSbirths.html]
  Date: 2025-11-07 10:33
  Status: 403
  Content-Type: text/html
  Size: 409 B
<HTML><HEAD>
<TITLE>Access Denied</TITLE>
</HEAD><BODY>
<H1>Access Denied</H1>
```

Dynamic webscraping: a motivational example

Doesn't seem to be due to change in policy—see `robots.txt`:

```
User-Agent: *

Disallow: /admin/
# Disallow: /agency/shutdown/
Disallow: /agency/commissioner/assets/materials/coss-letter-to-senator-warre
Disallow: /agency/commissioner/assets/materials/COSS-Open-Letter-90th-Annive
Disallow: /cgi-bin/
Disallow: /dev/
Disallow: /hlp/
Disallow: /info/
Disallow: /javascript/
Disallow: /myaccount/help/
Disallow: /myaccount/materials/pdfs/SSA-7004.pdf
Disallow: /onlineservices/eservices/
Disallow: /ssaexpress/
Disallow: /w3c/
Disallow: /agency/shutdown/materials/contingency-plan-08-05-19.pdf
Disallow: /agency/shutdown/materials/contingency-plan-08-05-19.html
Disallow: /news/mip/
Disallow: /menu/
Disallow: /news/materials/pdfs/2023-psrw-letter.pdf
```

Scraping with Selenium

Selenium is software that enables browser automation

- `https://www.selenium.dev/`
- Originally developed for web testing purposes

`{RSelenium}`: an R wrapper for Selenium

- Launchs a browser session and all communication will be routed through that browser session

There is also a Python wrapper for Selenium: the `selenium` module

Selenium WebDrivers

Selenium uses WebDrivers to automate browsers

- WebDrivers are APIs for web browsers

Selenium's WebDrivers allow for:

- Automation of normal browsers like Chrome, Firefox, etc.
  - Selenium launches them just like a human would, and they're visible on screen
  - Useful if you need to watch the automated web browsing
- Automation of **headless browsers**
  - Full browser functionality, but no visible browser window
  - Useful when you do not have a graphical interface or do not want to clutter your interface

Set up and useful information

To run `{RSelenium}` we are going to use Firefox because it tends to be less buggy than Chrome

Selenium requires you to install the “Java Development Kit” from Oracle, which you can find here: `https://www.oracle.com/java/technologies/downloads/`

In this course, you will learn how to use Selenium through R, but it’s (probably) better to use Python

- Very similar process, just have to learn slightly different syntax

{RSelenium}: basic navigation

```
# Load the `{RSelenium}` library
library("RSelenium")

# Create browser instance
rD <- rsDriver(browser=c("firefox"), phantomver = NULL)
browser <- rD$client

# Navigate to a url
browser$navigate("https://www.lse.ac.uk/")

# Navigate back
browser$goBack()

# End a browser instance
browser$close()   # Close browser window
rD$server$stop() # Stop server
```

{RSelenium}: interacting with a webpage

```
# Get the page source (HTML)
browser$getPageSource() # returns list object

# Find an element on a webpage
some_element <- browser$findElement(using = "xpath",
                                    value = "...")

# Click on an element you found (if button)
some_element$clickElement()

# Type text into an element (if text box)
search_box <- browser$findElement(using = "xpath",
                                  value = "...")
search_box$sendKeysToElement(list("some text"))

# Press enter key (e.g. for text box)
search_box$sendKeysToElement(list(key = "enter"))
```

Getting the SSA data

We can get the SSA data using {RSelenium}

```
library("RSelenium")
rD <- rsDriver(browser=c("firefox"), phantomver = NULL)
browser <- rD$client
browser$navigate("https://www.ssa.gov/oact/babynames/numberUSbirths.html")
html <- browser$getPageSource()
html <- read_html(html[[1]]) # Convert raw HTML to tree object
table <- html_table(html)
table <- table[[1]]
head(table)
```

```
# A tibble: 6 × 4
  `Year of birth` Male    Female  Total
            <int> <chr>   <chr>   <chr>
1            1880 118,399 97,604  216,003
2            1881 108,276 98,855  207,131
3            1882 122,031 115,694 237,725
4            1883 112,475 120,061 232,536
5            1884 122,738 137,585 260,323
6            1885 115,945 141,947 257,892
```

Outline

Webscraping has gotten more difficult

Partly due to AI and partly due to cyber threats:

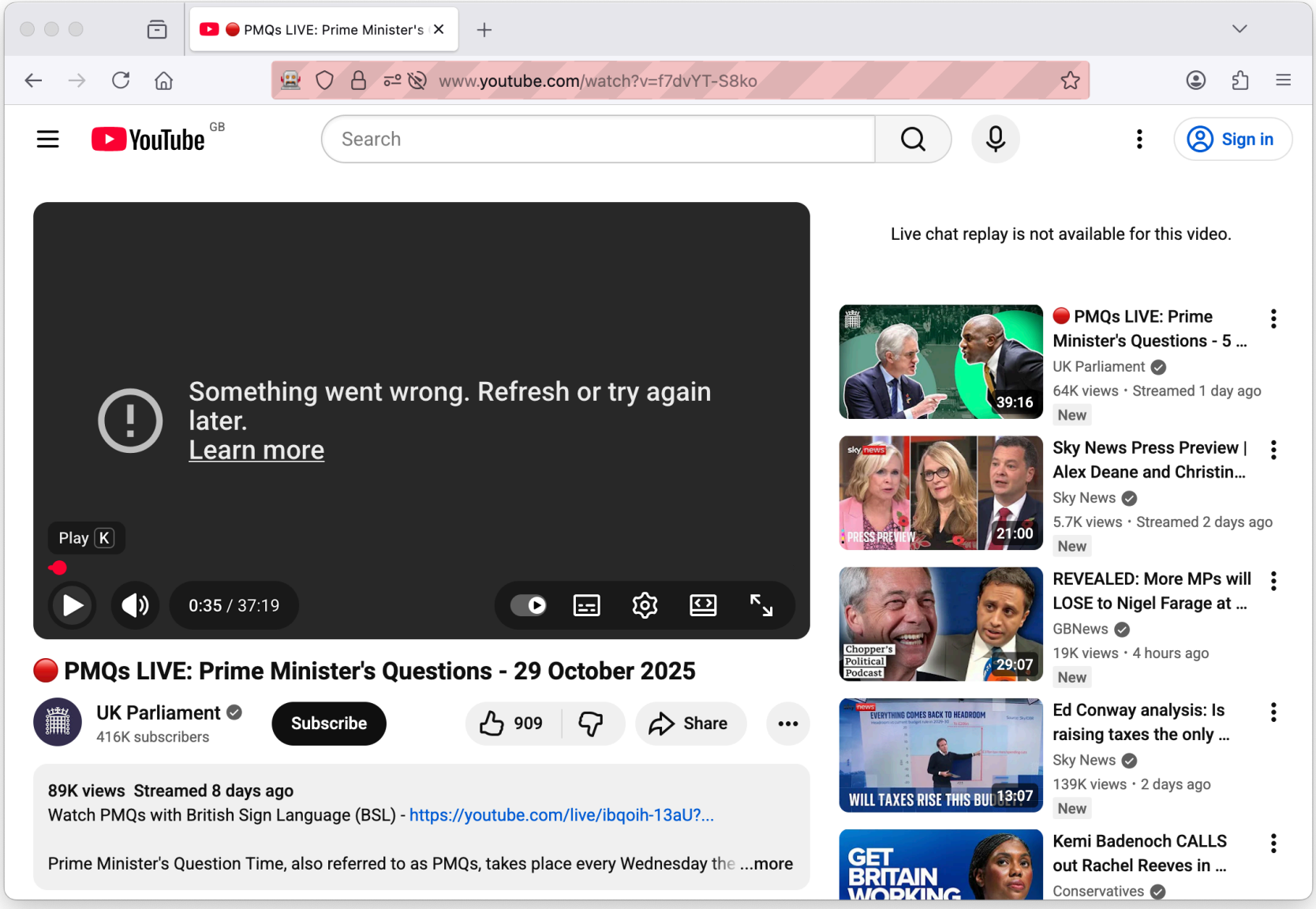
- Websites' terms of service have become much more stringent
- Their servers are more aggressive about blocking automation

It is increasingly infeasible to collect data via webscraping

Do not get in the habit of trying to *hide* who you are to evade a server's policies (e.g., by using `{RSelenium}`)

- It's unethical (and potentially illegal)
- It isn't always successful

Webscraping has gotten more difficult



Webscraping has gotten more difficult

Permissions and Restrictions

You may access and use the Service as made available to you, as long as you comply with this Agreement and the law. You may view or listen to Content for your personal, non-commercial use. You may also show YouTube videos through the embeddable YouTube player.

The following restrictions apply to your use of the Service. You are not allowed to:

1. access, reproduce, download, distribute, transmit, broadcast, display, sell, license, alter, modify or otherwise use any part of the Service or any Content except: (a) as specifically permitted by the Service; (b) with prior written permission from YouTube and, if applicable, the respective rights holders; or (c) as permitted by applicable law;
2. circumvent, disable, fraudulently engage, or otherwise interfere with the Service (or attempt to do any of these things), including security-related features or features that: (a) prevent or restrict the copying or other use of Content; or (b) limit the use of the Service or Content;
3. access the Service using any automated means (such as robots, botnets or scrapers) except: (a) in the case of public search engines, in accordance with YouTube's robots.txt file; (b) with YouTube's prior written permission; or (c) as permitted by applicable law;

What should you do?

This is an area where you need to develop good judgement

Servers are increasingly paranoid about cyberattacks, so they might think you are attacking even if you are compliant

You can use strategies to get around clunky server blocks, but:

- Check documentation and TOS—are you breaking rules?
- Are you being blocked because you're being a jerk and sending requests to the server too quickly?

Again, **use good judgement**—there will be consequences if you are too cavalier and get caught

For you while at LSE

Different people have different attitudes about risk

But you will have to explain how you collected your data

If you use a webscraper to collect data, you will be asked to explain whether you violated any terms of service

- A potential red flag: needing to use `{RSelenium}`

If you violate TOS: the bar is very high (but not impossible) for justifying why your data collection is necessary

You could fail your capstone/dissertation (or worse)

What should I do to get data I want?

1. Is there a direct download or an API? Then use that!

2. If not: ask yourself if collecting this data seems sketchy: are you stealing someone else's work? are you revealing personal information? are you breaking laws?

3. If you decide you can proceed: consult the `robots.txt` file and the site's terms of service to see if you can use automated methods

4. Create a bot-like webscraper that identifies who you are and your goal, and that does not overwhelm the remote webserver

5. If you get server responses indicating you do not have permission (e.g., 403 Forbidden), re-evaluate your answer to #2
   - ▷ Sometimes purely *techincal* reasons for blocking

6. Use browser automation if you are sure it is ethical