Web Scraping 3

Introduction

In our previous discussions, we've focused on **web scraping**, where we learned how to extract data and save it in various formats such as CSV, TXT, or XLS files. Additionally, we've explored how to use packages to interact directly with APIs.

Before we move forward, let's differentiate between **web scraping** and **web crawling**

Web Scraping vs. Web Crawling

- **Web Scraping** refers to the process of extracting specific data from web pages. This involves downloading the HTML content of a webpage and then parsing it to collect the desired information. Web scraping is the process of extracting specific data from a webpage, such as text, images, or links, for later use.
- **Simple Example**: If you want to collect product prices from an e-commerce website, you would use scraping to get the price information from each product page.

Web Crawlin

- **Web Crawling** Web crawling is the process of automatically navigating through websites, following links from one page to another, to gather or index content from various pages across the web.
- **Simple Example**: A search engine like Google uses web crawling to visit webpages, gather information, and add those pages to its index.

Key Difference

- **Key Difference**:
  - **Web scraping** is about extracting specific data from a single page or set of pages, while
  - **web crawling** is about systematically exploring multiple pages and following links across websites.

Objective of This Lesson

In this lesson, our goal is to **scrape data directly from the HTML of a webpage**. We will delve into how to parse the HTML content and extract the data you need, *without* relying on pre-formatted files or APIs.

Web scarping

- Web scarping is a technique for converting data (in unstructured format) to a structured format which can be used

Example of Structured Data:

- A spreadsheet with customer information, where each row represents a customer and each column includes fields like “Name,” “Age,” “Email,” and “Purchase Amount.”

Example of Unstructured Data:

- A collection of customer reviews in free-text form, where people can write anything they want about their experience.
- Web scraping is a very useful tool for extracting data from web pages. Some websites will offer an API, a set of structured HTTP requests that return data as JSON, which you handle using the techniques from the note `webscraping_2.pdf`. Where possible, you should use the API, because typically it will give you more reliable data.

Scraping ethics and legalities

Legalities

- Legalities depend a lot on where you live. However, as a general principle,
  - if the data is public, non-personal, and factual, you're likely to be ok.
  - If the data isn't public, non-personal, or factual or you're scraping the data specifically to make money with it, you'll need to talk to a lawyer.
- To scrape webpages, you need to first understand a little bit about HTML, the language that describes web pages.

Terms of service (TOS)

- If you look closely, you'll find many websites include a "terms and conditions" or "terms of service" link somewhere on the page, and if you read that page closely you'll often discover that the site specifically prohibits web scraping.

Personally identifiable information

- Even if the data is public, you should be extremely careful about scraping personally identifiable information like names, email addresses, phone numbers, dates of birth, etc.

Copyright

- You also need to worry about copyright law. Copyright law is complicated, but it's worth taking a look at the US law which describes exactly what's protected.

A quick review from “webscraping_2.pdf” - Directly Access APIs

- **HTTP** or Hypertext Transfer Protocol

Web Browser vs Web-Based Application

- A **web browser** and a **web-based application** are both essential to interacting with content on the internet, but they serve different purposes:

Web Browser

1. Web Browser

A **web browser** is software that allows users to access and view websites and web-based applications on the internet. Examples include Chrome, Firefox, Safari, and Edge. Web browsers interpret HTML, CSS, and JavaScript code from websites and display it in a way that's understandable to users.based applications.

Web-Based Application

2. Web-Based Application

A **web-based application** (or web app) is an application that runs on a web server and can be accessed through a web browser. Instead of installing software locally, users interact with the application through the browser over the internet. Examples include Gmail, Google Docs, and online document editors. Web apps are designed to provide functionality (like editing documents, sending messages, or managing tasks) directly in the browser.

HTTP

- It is a protocol to used for transferring data over internet. It is the Foundation of Data Communication on the World Wide Web and is used by most websites and web-based applications.
- When we access a website or an API through our web browser, or web-based application, your device sends an HTTP request to the server hosting the website or API. The server then send an HTTP response back to our device, containing the request data or resources.

- HTTP defines a set of rules and standards for how these requests and responses are formatted and transmitted, including how data is structured, how errors are handled, and how security is enforced.
- By using a common protocol like HTTP, different devices and systems can communicated and exchange data with each other over the internet, regardless of their underlying hardware or software.

Web page Structure

- Webpages are usually consists of two types of code.
  - One focus on the appearance and format of the page called **HTML** -The other one Called **XML** similar to *HTML* but focus more on managing data in web.

What is HTML? Source

- To scrape webpages, you need to first understand a little bit about HTML
- HTML stands for Hyper **Text Markup Language**
- It is the standard markup language for creating Web pages and it describes the structure of a Web page.
- It tells the browser how to display the content.

What is HTML?

- HTML (HyperText Markup Language) is the standard language used to create webpages. It uses “tags” to tell the browser how to display content.

HTML Tags

1. **Tags**:

- HTML uses “tags” to define different parts of the webpage. A tag is usually written with angle brackets, like `<tagname>`.
- Most tags have an **opening tag** and a **closing tag**.
- The **closing tag** has a forward slash / before the tag name.

*Example:

```
<tagname>Content goes here</tagname>
```

HTML Attribute

2. **Attributes**:

- Some tags have **attributes** to provide additional information.
- Attributes are written inside the opening tag.
- They are written as `attribute="value"`.

Example:

```
<tagname attribute="value">Content goes here</tagname>
```

HTML Content

3. **Content**:

- The **content** is the text or elements between the opening and closing tags. This is what will be displayed on the webpage. For instance, in `<p>Hello World</p>`, “Hello World” is the content.

HTML Element

4. **Element**: An element is a complete structure that includes both the opening tag, the content, and the closing tag. For example, `<p>Hello World</p>` is a paragraph element.

Summary

In HTML:

- **Tag**: A tag is a keyword surrounded by angle brackets (< >) that tells the browser how to display content. Example: `<h1>` is a tag for a heading.
- **Attributes**: Attributes are additional information about an element, written inside the opening tag. Example: `<a href="https://example.com">Link</a>`. Here, `href` is an attribute that specifies the link destination.
- **Contents**: The contents are the actual data or text between the opening and closing tags. For example, in `<p>Hello, world!</p>`, "Hello, world!" is the content.
- **Element**: An element consists of the opening tag, contents, and the closing tag. For example, `<p>Hello, world!</p>` is a complete paragraph element.

Think of HTML as a way to tell a web browser how to display a page. HTML uses tags to define different parts of the page.

Basic HTML Structure

1. **HTML Tags**: Every HTML document needs certain tags to work properly. These tags tell the browser that it's an HTML file and provide the basic structure.
2. **Necessary Tags**:

- `<html>`: This tag tells the browser where the HTML document starts and ends. It's like the "container" for the whole page.
- `<head>`: This section is where you put important information about the document (e.g., title, character set, links to stylesheets). This information won't show on the page but helps the browser understand it.
- `<title>`: The title tag (inside `<head>`) defines the title shown in the browser tab.
- `<body>`: This tag is where all the content that you want to display on the webpage goes, like text, images, and links.

3. **Order of Tags**:

- Your HTML document should follow this order:

```
<!DOCTYPE html>
  <html>
    <head>
      <title>Your Page Title</title>
    </head>
    <body>
      <!-- Content goes here -->
    </body>
  </html>
```

- Notice the `<!DOCTYPE html>` at the top. This isn't a tag, but a declaration that tells the browser this is an HTML5 document.

`<!DOCTYPE html>`

- The `<!DOCTYPE html>` line at the top of an HTML file tells the browser to use the latest web standards when displaying the page. Without it, the browser might switch to an older, less consistent way of showing the content, which can cause things like spacing and layout to look different or broken on some browsers. Even though you might not see an immediate difference if you remove it, using `<!DOCTYPE html>` makes sure your page looks right across all modern browsers and follows the best practices for web design.

Examples of a Simple HTML Page

Example 1:

- Here's an example to help you visualize:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My First Web Page</title>
  </head>
  <body>
    <h1>Welcome to My Page</h1>
    <p>This is a paragraph of text on my webpage.</p>
  </body>
</html>
```

Explanation:

- `<!DOCTYPE html>`: Declares that this is an HTML5 document.
- `<html>`: The root tag for the HTML document.
- `<head>`: Contains information about the document (like the title).
- `<title>`: Sets the title shown in the browser tab.
- `<body>`: Contains the content of the webpage.
- `<h1>`: A heading tag, used for main headings.
- `<p>`: A paragraph tag, used for blocks of text.

Key Points to Remember

- Every opening tag (like `<html>`) should have a closing tag (like `</html>`).
- The basic structure is always:
  1. `<!DOCTYPE html>`
  2. `<html>` … `</html>`
  3. `<head>` … `</head>`
  4. `<body>` … `</body>`

Following this structure will ensure your HTML page works correctly!

Example 2: Adding a Line Break and an Image

```
<!DOCTYPE html>
<html>
  <body>
    <p>The First image is Imperial State of Iran in
    1970s.<br>
    The Second is Islamic Republic or Iran Now.</p>
    <img src="image.jpg"
alt="Iran 1970s">
    <img src="image2.jpg"
alt="Iran Now">
  </body>
</html>
```

Explanation:

- `<br>`: Inserts a line break, moving the text after it to a new line.
- `<img>`: Embeds an image. It has two attributes:
- `src`: Specifies the path to the image file.
- `alt`: Provides alternative text for the image, displayed if the image cannot load.

Example 3: Adding Links and Lists

```
<!DOCTYPE html>
<html>
  <body>
    <p>Check out
    <a href="https://github.com/">this website</a>!</p>

    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
  </body>
</html>
```

Explanation:

- `<a href="URL">`: Creates a hyperlink. The `href` attribute specifies the link's destination.
- `<ul>`: Creates an unordered (bulleted) list.
- `<li>`: Defines an item in a list.

HTML Editors

- Using Notepad or TextEdit
  - Step 1: Open Notepad (PC) or TextEdit (Mac)
  - Step 2: Write Some HTML. You may want to write or copy the HTML code we had into Notepad.
  - Step 3: Save the HTML Page. Fot instance first.htm. You can use either `.htm` or `.html` as file extension. There is no difference; it is up to you.
  - Step 4: View the HTML Page in Your Browser. Open the saved HTML file in your favorite browser (double click on the file, or right-click - and choose "Open with").

What is XML?

- eXtensible Markup Language

XML

- It is uded for storing and transporting data
- It provides a way to structure data with tags that defines elements, attributes, and values.
- It is more flexible than *HTML*, allowing user to create their own tags to define the structure data.
- It is often uswed for data exchange between different system, such as web services and database.

XML vs HTML

Here's a simple example that demonstrates the difference between XML and HTML:

XML Example:

```
<book>
    <title>Introduction to Programming</title>
    <author>John Doe</author>
    <price>29.99</price>
</book>
```

HTML Example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Book Store</title>
    </head>
    <body>
        <h1>Welcome to the Book Store</h1>
        <p>This is an introduction to our book collection.<
    </body>
</html>
```

- You may say, I can create a page that generates the title of the book, the author of the book and the price only by HTML, so why XML?
- You're absolutely right that HTML could display simple data like a book's title, author, and price on a webpage. The main difference between XML and HTML isn't just in **what they show**, but in **why and how they are used**.

Main Difference Using Our Example:

- If you want a webpage that **displays** a book's title, author, and price to someone, you'd use **HTML**.
- If you want to **store** book information in a way that a computer program can read, exchange, or process it later, you'd use **XML**.

Think of it Like This:

- **HTML is like a newspaper:** It arranges and displays information for people to read easily.
- **XML is like a filing cabinet:** It organizes information so that other systems can understand it, retrieve it, and use it in different ways.
- So, while **HTML** is used to make things look good on a webpage, **XML** is used to store and structure data that can be shared and used by different programs.

Example :

Complete Code

```
<?xml version="1.0" encoding="UTF-8"?>
<webpage>
<style>
body {
  background-color: lightblue;
}
heading{
    color: green;
    font-family: verdana;
    font-size: 24px;
    }
paragraph{
    color: blue;
    font-family: verdana;
    font-size: 22px;
    }
```

Example:

Code Part 1_a

```
<?xml version="1.0" encoding="UTF-8"?>
<webpage>
<style>
body {
  background-color: lightblue;
}
heading{
    color: green;
    font-family: verdana;
    font-size: 24px;
    }
paragraph{
    color: blue;
    font-family: verdana;
    font-size: 22px;
    }
```

Code Part 1_b

```
h1 {
  color: red;
  text-align: center;
}

p {
  font-family: verdana;
  font-size: 20px;
}
</style>
  <content>
    <heading>Welcome to XML</heading>
    <br/>
    <paragraph>This is a paragraph of text.</paragraph>
  </content>
</webpage>
```

Code Part 2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <h1>This is a headline</h1>
    <p>This is a paragraph</p>
    <p>This is another paragraph
    <br/>
    <p>With line break.</p>
  </body>
</html>
```

CSS

- Cascading Style Sheets

What is CSS (Cascading Style Sheets)?

- CSS is the language we use to style an HTML document.
- CSS describes how HTML elements should be displayed.

Why Use CSS?

- CSS is used to define styles for your web pages, including the design, layout, and variations in display for different devices and screen sizes.

```
body {
  background-color: lightblue;
}

h1 {
  color: white;
  text-align: center;
}

p {
  font-family: verdana;
  font-size: 20px;
}
```

Example 1:

Code Part 1:

```
<!DOCTYPE html>
<html>
<head>
<style>
body {
  background-color: lightblue;
}

h1 {
  color: red;
  text-align: center;
}
```

Example:

Code Part 2:

```
p {
  font-family: verdana;
  font-size: 20px;
}
</style>
<title>This is the page title</title>
</head>
<body>

<h1>This is a headline</h1>
<p>This is a paragraph.</p>
<p>This is another paragraph
<br> with line break.</p>
<p>This is a new paragraph.</p>

</body>
```

Example 2:

Code part 1:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
    body {
      background-color: lightblue;
    }

    h1 {
        color: red;
        text-align: center;
}
```

Code part 2:

```
p {
  font-family: verdana;
  font-size: 20px;
}
</style>
  <title>This is the page title</title>
    </head>
    <body>

    <h1>This is a headline</h1>
    <p>This is a paragraph.</p>
    <p>This is another paragraph
      <br> with line break.</p>
    <p>This is a new paragraph.</p>

  </body>
</html>
```

- `p` is a selector in CSS (it points to the HTML element you want to style).
- color is a property, and red is the property value text-align is a property, and center is the property value

Extracting data

Install and load packages

```
library(tidyverse)
library(rvest)
```

Read URL

Use `read_html()` to read the HTML for that page into R. This returns an xml_document object which you'll then manipulate using {rvest} functions:

```
html <- read_html(
  "https://rvest.tidyverse.org/articles/starwars.html")

html
```

```
{html_document}
<html lang="en">
[1] <head>\n<meta http-equiv="Content-Type" content="text/h
```

Find elements

- NOTE: This is the hardest part of Web scrapping!

CSS is short for cascading style sheets, and is a tool for defining the visual styling of HTML documents. CSS includes a miniature language for selecting elements on a page called CSS selectors. Here three most important

1. `p` selects all
2. `.title` selects all elements with `class` "title".
3. `#title` selects the element with the `id` attribute that equals "title".

Example

```
html <- minimal_html("
  <h1>This is a heading</h1>
  <p id='first'>This is a paragraph</p>
  <p class='important'>
  This is an important paragraph</p>
")
```

html_element() and html_elements()

"p"

html_element() always returns the same number of outputs as inputs.

```
html |> html_element("p")
```

```
{html_node}
<p id="first">
```

Use html_elements() to find all elements that match the selector:

```
html |> html_elements("p")
```

```
{xml_nodeset (2)}
[1] <p id="first">This is a paragraph</p>
[2] <p class="important">\n  This is an important paragraph
```

html_element() and html_elements()

"important"

```
html |> html_element(".important") # .important selects al
```

```
{html_node}
<p class="important">
```

```
html |> html_elements(".important") # .important selects a
```

```
{xml_nodeset (1)}
[1] <p class="important">\n  This is an important paragraph
```

html_element() and html_elements()

"#first"

```
html |> html_element("#first") # The #first selects the ele
```

```
{html_node}
<p id="first">
```

```
html |> html_elements("#first") # The #first selects the el
```

```
{xml_nodeset (1)}
[1] <p id="first">This is a paragraph</p>
```

`html_element()` and `html_elements()`

- Another difference between `html_element()` and `html_elements()`. When you use a selector that doesn't match any elements
- `html_elements()` returns a vector of length 0

```
html |> html_elements("b")
```

```
{xml_nodeset (0)}
```

- `html_element()` returns a missing value.

```
html |> html_element("b")
```

```
{xml_missing}
<NA>
```

Nesting selections

- We are using `html_elements()` to identify *elements that will become observations* then
- using `html_element()` to find *elements that will become variables*.

```
html <- minimal_html("
  <ul>
    <li><b>C-3PO</b> is a <i>droid</i>
    that weighs <span class='weight'>167 kg</span></li>
    <li><b>R4-P17</b> is a <i>droid</i></li>
    <li><b>R2-D2</b> is a <i>droid</i>
    that weighs <span class='weight'>96 kg</span></li>
    <li><b>Yoda</b> weighs <span class='weight'>
    66 kg</span></li>
  </ul>
  ")
```

- We can use `html_elements()` to make a vector where each element corresponds to a different character:

```
characters <- html |> html_elements("li")
characters
```

```
{xml_nodeset (4)}
[1] <li>\n<b>C-3PO</b> is a <i>droid</i>\n    that weighs <
[2] <li>\n<b>R4-P17</b> is a <i>droid</i>\n</li>
[3] <li>\n<b>R2-D2</b> is a <i>droid</i> \n    that weighs
[4] <li>\n<b>Yoda</b> weighs <span class="weight">\n    66
```

```
 html |> html_element("li")
```

```
{html_node}
<li>
[1] <b>C-3PO</b>
[2] <i>droid</i>
[3] <span class="weight">167 kg</span>
```

To extract the name of each character, we use `html_element()`

```
characters |> html_elements("b")
```

```
{xml_nodeset (4)}
[1] <b>C-3PO</b>
[2] <b>R4-P17</b>
[3] <b>R2-D2</b>
[4] <b>Yoda</b>
```

```
characters |> html_element("b")
```

```
{xml_nodeset (4)}
[1] <b>C-3PO</b>
[2] <b>R4-P17</b>
[3] <b>R2-D2</b>
[4] <b>Yoda</b>
```

- The distinction between `html_element()` and `html_elements()`isn't important for name, but it is important for **weight**. We want to get one weight for each character, even if there's no weight .

```
characters |> html_element(".weight")
```

```
{xml_nodeset (4)}
[1] <span class="weight">167 kg</span>
[2] NA
[3] <span class="weight">96 kg</span>
[4] <span class="weight">\n    66 kg</span>
```

- `html_elements()` finds all weight s that are children of characters. There's only three of these, so we lose the connection between names and weights:

```
characters |> html_elements(".weight")
```

```
{xml_nodeset (3)}
[1] <span class="weight">167 kg</span>
[2] <span class="weight">96 kg</span>
[3] <span class="weight">\n    66 kg</span>
```

Text and attributes

- `html_text2()` extracts the plain text contents of an HTML element:

```
characters |>
  html_element("b") |>
  html_text2() -> name
name
```

```
[1] "C-3PO"  "R4-P17" "R2-D2"  "Yoda"
```

```
characters |>
  html_element(".weight") |>
  html_text2() ->
  weight
weight
```

```
[1] "167 kg" NA       "96 kg"  "66 kg"
```

```
#> [1] "167 kg" NA       "96 kg"  "66 kg"
```

```
characters |>
  html_elements(".weight") |>
  html_text2()
```

```
[1] "167 kg" "96 kg"  "66 kg"
```

```
#> [1] "167 kg"        "96 kg"  "66 kg"
```

```
tibble(name, weight)
```

```
# A tibble: 4 x 2
  name    weight
  <chr>   <chr>
1 C-3PO   167 kg
2 R4-P17  <NA>
3 R2-D2   96 kg
4 Yoda    66 kg
```

HTML ATTRIBUTE

- `html_attr()` extracts data from attributes:

```
html <- minimal_html("
  <p><a href='https://en.wikipedia.org/wiki/Cat'>
  cats</a></p>
  <p><a href='https://en.wikipedia.org/wiki/Dog'>
  dogs</a></p>
")
html
```

```
{html_document}
<html>
[1] <head>\n<meta http-equiv="Content-Type" content="text/h
[2] <body>\n<p><a href="https://en.wikipedia.org/wiki/Cat">
```

```
html |>
  html_elements("p")
```

```
{xml_nodeset (2)}
[1] <p><a href="https://en.wikipedia.org/wiki/Cat">\n  cats
[2] <p><a href="https://en.wikipedia.org/wiki/Dog">\n  dogs
```

```
html |>
  html_elements("a")
```

```
{xml_nodeset (2)}
[1] <a href="https://en.wikipedia.org/wiki/Cat">\n  cats</a
[2] <a href="https://en.wikipedia.org/wiki/Dog">\n  dogs</a
```

```
html |>
  html_attr("href")
```

```
[1] NA
```

- `html_attr()` always returns a string, so if you're extracting numbers or dates, you'll need to do some post-processing.
- You need to prepare it

```
html |>
  html_elements("p") |>
  html_element("a") |>
  html_attr("href")
```

```
[1] "https://en.wikipedia.org/wiki/Cat" "https://en.wikipec
```

Or

```
html |>
  html_elements("a") |>
  html_attr("href")
```

```
[1] "https://en.wikipedia.org/wiki/Cat" "https://en.wikipec
```

Tables

- HTML tables are built up from four main elements: `<table>`, `<tr>` (table row), `<th>` (table heading), and `<td>` (table data). Here's a simple HTML table with two columns and three rows:

```
html <- minimal_html("
                     <table class='mytable'>
                     <tr><th>x</th>   <th>y</th></tr>
                     <tr><td>1.5</td> <td>2.7</td></tr>
                     <tr><td>4.9</td> <td>1.3</td></tr>
                     <tr><td>7.2</td> <td>8.1</td></tr>
                     </table>
                     ")
```

- In our html we are lucky, the data is stored in an HTML table.
- We need to use `html_element()` to identify the table we want to extract and then use `html_table()` to get the table in R

```
html |>
  html_element(".mytable")
```

```
{html_node}
<table class="mytable">
[1] <tr>\n<th>x</th>   <th>y</th>\n</tr>\n
[2] <tr>\n<td>1.5</td> <td>2.7</td>\n</tr>\n
[3] <tr>\n<td>4.9</td> <td>1.3</td>\n</tr>\n
[4] <tr>\n<td>7.2</td> <td>8.1</td>\n</tr>
```

```
html |>
  html_element(".mytable") |>
  html_table()
```

```
# A tibble: 3 x 2
      x     y
  <dbl> <dbl>
1   1.5   2.7
2   4.9   1.3
3   7.2   8.1
```

- Note that `x` and `y` have automatically been converted to numbers. This automatic conversion doesn't always work, so in more complex scenarios you may want to turn it off with `convert = FALSE` and then do your own conversion.

```
html |>
  html_element(".mytable") |>
  html_table(convert = FALSE)
```

```
# A tibble: 3 x 2
  x     y
  <chr> <chr>
1 1.5   2.7
2 4.9   1.3
3 7.2   8.1
```

- Now we use our conversion

```
html |>
  html_element(".mytable") |>
  html_table(convert = FALSE) ->
  table
```

```r
table |>
  mutate(x = parse_number(x), y = parse_double(y))
```

```
# A tibble: 3 x 2
      x     y
  <dbl> <dbl>
1   1.5   2.7
2   4.9   1.3
3   7.2   8.1
```

Finding the right selectors

The hardest part of scraping is to figuring out the selector you need. here are two main tools that are available to help you with this process: SelectorGadget and your browser's developer tools.

1. **SelectorGadget**: It doesn't always work, but when it does, it's magic! You can learn how to install and use **SelectorGadget** by
2. reading SelectorGadget and
3. watchingYoutube

Every modern browser comes with some toolkit for developers, but we recommend **Chrome**.

Examples:

- **Note: There's some risk that these examples may no longer work when you run them

— that's the fundamental challenge of web scraping; if the structure of the site changes, then you'll have to change your scraping code.**

Exampl1: StarWars

- We'd encourage you to navigate to that page now and use "Inspect Element" to inspect one of the headings that's the title of a Star Wars movie. *While you are on the webpage right click and when menu is open scroll down to bottom of the page and then click "Inspect"*
- You should be able to see that each movie has a shared structure that looks like this:

```
<section>
  <h2 data-id="1">The Phantom Menace</h2>
  <p>Released: 1999-05-19</p>
  <p>Director: <span class="director">George Lucas</span></

  <div class="crawl">
    <p>...</p>
    <p>...</p>
    <p>...</p>
  </div>
</section>
```

Step 1

- We'll start by reading the HTML and extracting all the `<section>`

elements: - Step1: Read URL

```
url <- "https://rvest.tidyverse.org/articles/starwars.html"

html <- read_html(url)
html
```

```
{html_document}
<html lang="en">
[1] <head>\n<meta http-equiv="Content-Type" content="text/h
[2] <body>\n    <a href="#container" class="visually-hidden
```

Step 2: Identify elements

```
section <- html |> html_elements("section")
section
```

```
{xml_nodeset (7)}
[1] <section><h2 data-id="1">\nThe Phantom Menace\n</h2>\n<
[2] <section><h2 data-id="2">\nAttack of the Clones\n</h2>\
[3] <section><h2 data-id="3">\nRevenge of the Sith\n</h2>\n
[4] <section><h2 data-id="4">\nA New Hope\n</h2>\n<p>\nRele
[5] <section><h2 data-id="5">\nThe Empire Strikes Back\n</h
[6] <section><h2 data-id="6">\nReturn of the Jedi\n</h2>\n<
[7] <section><h2 data-id="7">\nThe Force Awakens\n</h2>\n<p
```

- This retrieves seven elements matching the seven movies found on that page, suggesting that using section as a selector is good.

```
html |> html_elements("section") |>
  html_elements("h2")
```

```
{xml_nodeset (7)}
[1] <h2 data-id="1">\nThe Phantom Menace\n</h2>
[2] <h2 data-id="2">\nAttack of the Clones\n</h2>
[3] <h2 data-id="3">\nRevenge of the Sith\n</h2>
[4] <h2 data-id="4">\nA New Hope\n</h2>
[5] <h2 data-id="5">\nThe Empire Strikes Back\n</h2>
[6] <h2 data-id="6">\nReturn of the Jedi\n</h2>
[7] <h2 data-id="7">\nThe Force Awakens\n</h2>
```

```
section |>
  html_element("h2") |>
  html_text2()
```

```
[1] "The Phantom Menace"      "Attack of the Clones"
[3] "Revenge of the Sith"     "A New Hope"
[5] "The Empire Strikes Back" "Return of the Jedi"
[7] "The Force Awakens"
```

```
section |>
  html_element("p") |>
  html_text2()
```

```
[1] "Released: 1999-05-19" "Released: 2002-05-16" "Released
[4] "Released: 1977-05-25" "Released: 1980-05-17" "Released
[7] "Released: 2015-12-11"
```

We have to removed released

```
section |>
  html_element("p") |>
  html_text2() |>
  str_remove_all("Released: ")
```

```
[1] "1999-05-19" "2002-05-16" "2005-05-19" "1977-05-25" "19
[6] "1983-05-25" "2015-12-11"
```

- The class is character

```
section |>
  html_element("p") |>
  html_text2() |>
  str_remove_all("Released: ") |>
  class()
```

```
[1] "character"
```

- Now we need to parse it to date

```
section |>
  html_element("p") |>
  html_text2() |>
  str_remove_all("Released: ") |>
  parse_date()
```

```
[1] "1999-05-19" "2002-05-16" "2005-05-19" "1977-05-25" "19
[6] "1983-05-25" "2015-12-11"
```

- Let us check the class

```
section |>
  html_element("p") |>
  html_text2() |>
  str_remove_all("Released: ") |>
  parse_date() |>
  class()
```

```
[1] "Date"
```

```
section |>
  html_element(".crawl") |>
  html_text2()
```

```
[1] "Turmoil has engulfed the Galactic Republic. The taxati
[2] "There is unrest in the Galactic Senate. Several thousa
[3] "War! The Republic is crumbling under attacks by the ru
[4] "It is a period of civil war. Rebel spaceships, strikin
[5] "It is a dark time for the Rebellion. Although the Deat
[6] "Luke Skywalker has returned to his home planet of Tato
[7] "Luke Skywalker has vanished. In his absence, the sinis
```

```
section |>
  html_element(".director") |>
  html_text2()
```

```
[1] "George Lucas"     "George Lucas"     "George Lucas"
[5] "Irvin Kershner"   "Richard Marquand" "J. J. Abrams"
```

- Once we've done that for each component, we can wrap all the results up into a tibble:

```
tibble(
  title = section |>
    html_element("h2") |>
    html_text2(),
  released = section |>
    html_element("p") |>
    html_text2() |>
    str_remove("Released: ") |>
    parse_date(),
  director = section |>
    html_element(".director") |>
    html_text2(),
  intro = section |>
    html_element(".crawl") |>
    html_text2()
)
```

Example:

Please see `4_web_scraping.R` `5_web_scraping_table.R` `6_practice_web_scraping.R` and more

NOTE: {rvest} 1.0.0 renamed a number of functions.

- `set_values()` -> `html_form_set()`
- `submit_form()` -> `session_submit()`
- `xml_tag()` -> `html_name()`
- `xml_node()` & `html_node()` -> `html_element()`
- `xml_nodes()` & `html_nodes()` -> `html_elements()`

- **Additionally all session related functions gained a common prefix:**
- `html_session()` -> `session()`
- `forward()` -> `session_forward()`
- `back()` -> `session_back()`
- `jump_to()` -> `session_jump_to()`
- `follow_link()` -> `session_follow_link()`

Therefore, you may want to do some adjustment on the `.R` files that we refered you. I have already fixed `4_web_scraping.R` and `5_web_scraping_table.R`