Introduction to Data Science

Session 5: Web data and technologies

Simon Munzert
Hertie School | GRAD-C11/E1339

Table of contents

Web data for data science

What is web data?

Data Descriptor | Open Access | Published: 02 August 2021

The Upworthy Research Archive, a time series of 32,487 experiments in U.S. media

J. Nathan Matias, Kevin Munger, Marianne Aubin Le Quere & Charles Ebersole

Scientific Data **8**, Article number: 195 (2021) | Cite this article

Abstract

The pursuit of audience attention online has led organizations to conduct thousands of behavioral experiments each year in media, politics, activism, and digital technology. One pioneer of A/B tests was Upworthy.com, a U.S. media publisher that conducted a randomized trial for every article they published. Each experiment tested variations in a headline and image "package," recording how many randomly-assigned viewers selected each variation. While none of these tests were designed to answer scientific questions, scientists can advance knowledge by meta-analyzing and data-mining the tens of thousands of experiments Upworthy conducted. This archive records the stimuli and outcome for every A/B test fielded by Upworthy between January 24, 2013 and April 30, 2015. In total, the archive includes 32,487 experiments, 150,817 experiment arms, and 538,272,878 participant assignments. The open access dataset is organized to support exploratory and confirmatory research, as well as meta-scientific research on ways that scientists make use of the archive.

British Journal of Political Science (2021), page 1 of 11
doi:10.1017/S0007123420000897

British Journal of Political Science

LETTER

The Comparative Legislators Database

Sascha Göbel[1]* and Simon Munzert[2]

[1]Faculty of Social Sciences, Goethe University Frankfurt am Main, Germany; and [2]Data Science Lab, Hertie School, Berlin, Germany
*Corresponding author. E-mail: sascha.goebel@soz.uni-frankfurt.de

(Received 7 June 2020; revised 12 November 2020; accepted 2 December 2020)

Abstract

Knowledge about political representatives' behavior is crucial for a deeper understanding of politics and policy-making processes. Yet resources on legislative elites are scattered, often specialized, limited in scope or not always accessible. This article introduces the Comparative Legislators Database (CLD), which joins micro-data collection efforts on open-collaboration platforms and other sources, and integrates with renowned political science datasets. The CLD includes political, sociodemographic, career, online presence, public attention, and visual information for over 45,000 contemporary and historical politicians from ten countries. The authors provide a straightforward and open-source interface to the database through an R package, offering targeted, fast and analysis-ready access in formats familiar to social scientists and standardized across time and space. The data is verified against human-coded datasets, and its use for investigating legislator prominence and turnover is illustrated. The CLD contributes to a central hub for versatile information about legislators and their behavior, supporting individual-level comparative research over long periods.

What is web data? (cont.)

Experimental evidence of massive-scale emotional contagion through social networks

Adam D. I. Kramer, Jamie E. Guillory, and Jeffrey T. Hancock

+ See all authors and affiliations

PNAS June 17, 2014 111 (24) 8788-8790; first published June 2, 2014; https://doi.org/10.1073/pnas.1320040111

Edited by Susan T. Fiske, Princeton University, Princeton, NJ, and approved March 25, 2014 (received for review October 23, 2013)

This article has Corrections. Please see:

Editorial Expression of Concern: Experimental evidence of massivescale emotional contagion through social networks - July 03, 2014

Correction for Kramer et al., Experimental evidence of massive-scale emotional contagion through social networks - July 03, 2014

Article | Figures & SI | Info & Metrics | PDF

Significance

We show, via a massive (*N* = 689,003) experiment on Facebook, that emotional states can be transferred to others via emotional contagion, leading people to experience the same emotions without their awareness. We provide experimental evidence that emotional contagion occurs without direct interaction between people (exposure to a friend expressing an emotion is sufficient), and in the complete absence of nonverbal cues.

The consequences of online partisan media

Andrew M. Guess[a,b,1,2], Pablo Barberá[c,1], Simon Munzert[d,1], and JungHwan Yang (양정환)[e,1]

[a]Department of Politics, Princeton University, Princeton, NJ 08544; [b]School of Public and International Affairs, Princeton University, Princeton, NJ 08544; [c]Department of Political Science and International Relations, University of Southern California, Los Angeles, CA 90089; [d]Data Science Lab, Hertie School, 10117 Berlin, Germany; and [e]Department of Communication, University of Illinois at Urbana-Champaign, Urbana, IL 61801

Edited by Christopher Andrew Bail, Duke University, Durham, NC, and accepted by Editorial Board Member Margaret Levi February 17, 2021 (received for review June 29, 2020)

What role do ideologically extreme media play in the polarization of society? Here we report results from a randomized longitudinal field experiment embedded in a nationally representative online panel survey (*N* = 1,037) in which participants were incentivized to change their browser default settings and social media following patterns, boosting the likelihood of encountering news with either a left-leaning (HuffPost) or right-leaning (Fox News) slant during the 2018 US midterm election campaign. Data on ≈ 19 million web visits by respondents indicate that resulting changes in news consumption persisted for at least 8 wk. Greater exposure to partisan news can cause immediate but short-lived increases in website visits and knowledge of recent events. After adjusting for multiple comparisons, however, we find little evidence of a direct impact on opinions or affect. Still, results from later survey waves suggest that both treatments produce a lasting and meaningful decrease in trust in the mainstream media up to 1 y later. Consistent with the minimal-effects tradition, direct consequences of online partisan media are limited, although our findings raise questions about the possibility of subtle, cumulative dynamics. The combination of experimentation and computational social science techniques illustrates a powerful approach for studying the long-term consequences of exposure to partisan news.

media | politics | polarization | computational social science

argues that media primarily reinforce existing predispositions (16). At the same time, more recent research strongly implies that newspapers and especially cable news can change people's voting behavior, especially those without strong partisan attachments (17–20). We propose an internet-age synthesis that views people's information environments through the lens of choice architecture (21): frictions, subtle design features, and default settings that structure people's online experience. In this view, small changes (or nudges) could disproportionately affect information consumption habits that have downstream consequences.

To that end, we designed a large, longitudinal online field experiment that subtly but naturalistically increased people's exposure to partisan news websites. Our choice of treatment is ecologically valid: Despite the importance of social media for agenda-setting (22) and public expression (23), more Americans continue to say that they get news from news websites or apps than social media sites (24). The intervention thus served as a nudge, boosting the likelihood that subjects encountered news framed with a partisan slant during their day-to-day web browsing experience, even if inadvertently. The powerful, sustained nature of the intervention and our ability to track participants with survey and behavioral data for months provided the opportunity to test a range of hypotheses about the long-term impact of online partisan media.

Our preregistered hypotheses were divided into two separate

What is web data? (cont.)

So what is web data, really?

- Not all data you get from the web is "web data".
- Web data is **data that is created on, for, or via the web**. By that definition, a survey dataset that you download from a data repository is not web data.
- On the other hand, survey data collected online (i.e., web/mobile questionnaires) is web data but we don't consider it in today's session.
- Examples of web data:
  - Online news articles
  - Social media network structures
  - Crowdsourced databases (e.g., Wikidata)
  - Server logs (e.g., viewership statistics)
  - Data from surveys, experiments, clickworkers
  - Just any website

What is web data? (cont.)

So what is web data, really?

- Not all data you get from the web is "web data".
- Web data is **data that is created on, for, or via the web**. By that definition, a survey dataset that you download from a data repository is not web data.
- On the other hand, survey data collected online (i.e., web/mobile questionnaires) is web data but we don't consider it in today's session.
- Examples of web data:
  - Online news articles
  - Social media network structures
  - Crowdsourced databases (e.g., Wikidata)
  - Server logs (e.g., viewership statistics)
  - Data from surveys, experiments, clickworkers
  - Just any website

And why is web data attractive?

- Data is abundant online.
- Human behavior increasingly takes place online.
- Countless services track human behavior.
- Getting data from the web is cheap and often quick.
- An analysis workflow that involves web data can often be easily updated.
- The vast majority of web data was not created with a data analysis purpose in mind. This fact is often a feature, not a bug.

Today, we focus on one particular way of collecting data from the web: **web scraping**. This also limits the type of web data we'll be talking about (basically: data from static webpages). But it'll be fun nevertheless.

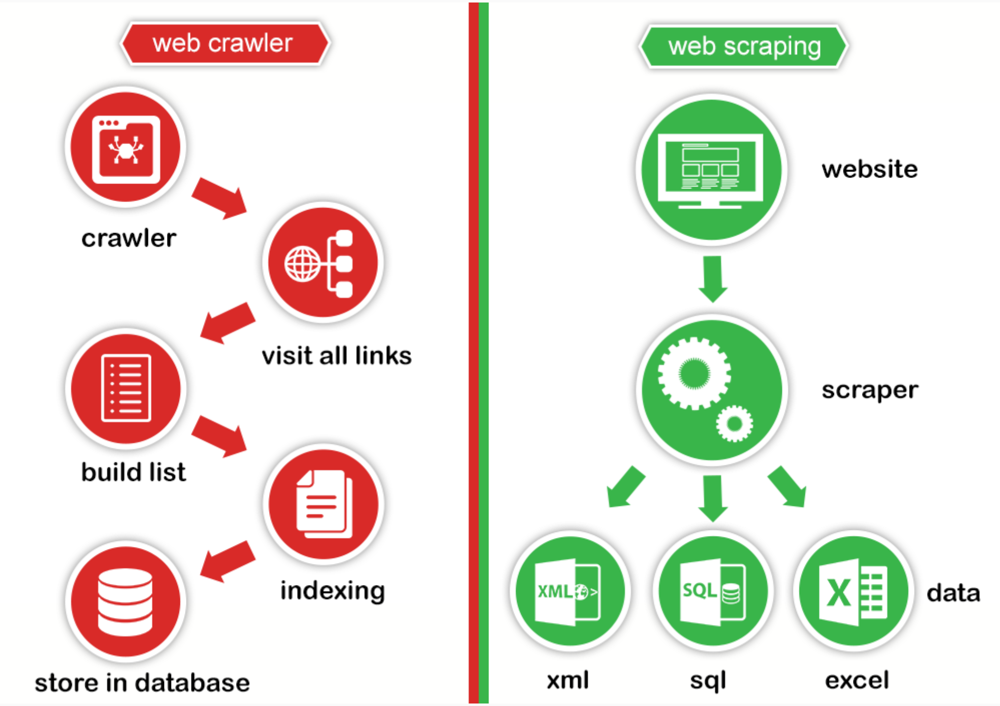
Web scraping

What is web scraping?

1. Pulling (unstructured) data from websites (HTMLs)
2. Bringing it into shape (into an analysis-ready format)

The philosophy of scraping with R

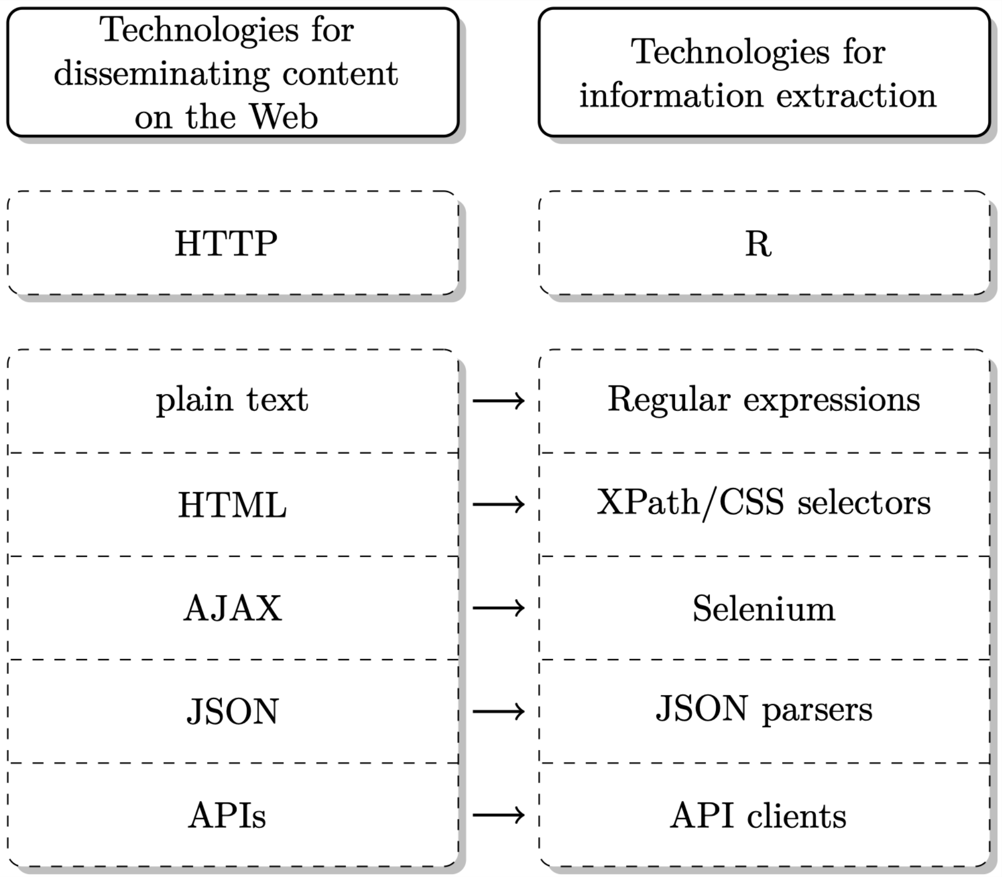
- No point-and-click procedure
- Script the entire process from start to finish
- **Automate**
  - The downloading of files
  - The scraping of information from web sites
  - Tapping APIs
  - Parsing of web content
  - Data tidying, text data processing
- Easily scale up scraping procedures
- Scheduling of scraping tasks



`Credit` prowebscraping.com

Technologies of the world wide web

- To fully unlock the potential of web data for data science, we draw on certain web technologies.
- Importantly, often a basic understanding of these technologies is sufficient as the focus is on web data collection, not web development.
- Specifically, we have to understand
  - How our machine/browser/R communicates with web servers (→ **HTTP/S**)
  - How websites are built (→ **HTML**, **CSS**, basics of **JavaScript**)
  - How content in webpages can be effectively located (→ **XPath**, **CSS selectors**)
  - How dynamic web applications are executed and tapped (→ **AJAX**, **Selenium**)
  - How data by web services is distributed and processed (→ **APIs**, **JSON**, **XML**)



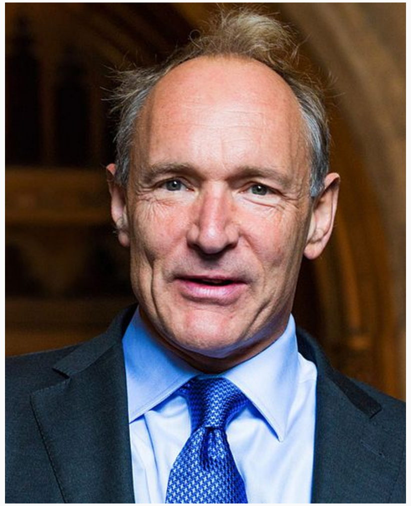
`Credit` ADCR

HTML basics

HTML background

What is HTML?

- **H**yper**T**ext **M**arkup **L**anguage
- Markup language = plain text + markups
- Originally specified by Tim Berners-Lee at CERN in 1989/90
- W3C standard for the construction of websites.
- The fundamentals of HTML haven't changed much recently. Current version is HTML 5.2 (published in 2017).



What is it good for?

- In the early days, the internet was mainly good for sharing texts. But plain text is boring. **Markup** is *fun*!
- HTML lies underneath of what you see in your browser. You don't see it because your browser interprets and renders it for you.
- A basic understanding of HTML helps us locate the information we want to retrieve.

HTML

HTML tree structure

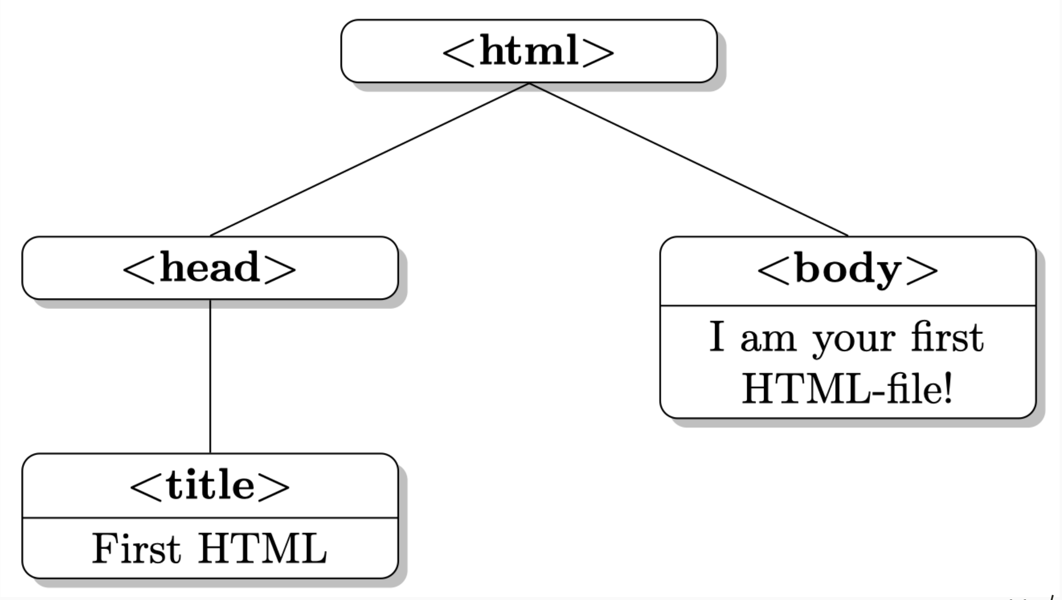
The DOM tree

- HTML documents are hierarchically structured. Think of them as a tree with multiple nodes and branches.
- When a webpage (HTML resource) is loaded, the browser creates a Document Object Model of that page - the **DOM Tree**.
- Think of it as a representation that considers all HTML elements as objects than can be accessed.

Parts of the tree

- The DOM is constituted of **nodes**, which are just data types that can be referred to - such as "attribute node", "text node", or "element node".
- A **node set** is a set of nodes. This will become relevant when you learn about XPath, which you can use to access multiple nodes (e.g., all `title` nodes).

```
<!DOCTYPE html>
  <html>
    <head>
      <title id=1>First HTML</title>
    </head>
  <body>
    I am your first HTML file!
  </body>
</html>
```



HTML: elements and attributes

Elements

- Elements are a combination of start tags, content, and end tags.
- Example: `<title>First HTML</title>`
- An element is everything from (including) the element's start tag to (including) the element's end tag, but also other elements that are nested within that element.
- Syntax:

Component	Representation
Element title	`title`
Start tag	`<title>`
End tag	`</title>`
Value	`First HTML`

Attributes

- Describe elements and are stored in the start tag.
- There are specific attributes for specific elements.
- Example: `<a href="http://www.r-datacollection.com/">Link to Homepage</a>`
- Syntax:
  - Name-value pairs: `name="value"`
  - Simple and double quotation marks possible
  - Several attributes per element possible

Why tags and attributes are important

- Tags structure HTML documents.
- In the context of web scraping, the structure can be exploited to locate and extract data from websites.

Important tags and attributes

Anchor tag `<a>`

- Links to other pages or resources.
- Classical links are always formatted with an anchor tag.
- The `href` attribute determines the target location.
- The value is the name of the link.

Link to another resource:

```
<a href="en.wikipedia.org/wiki/List_of_lists_of_lists">Link with absolute path</a>
```

Reference within a document:

```
<a id="top">Reference point</a>
```

Link to a reference within a document:

```
<a href="#top">Link to reference point</a>
```

Important tags and attributes

Heading tags `<h1>`, `<h2>`, …, and paragraph tag `<p>`

- Structure text and paragraphs.
- Heading tags range from level 1 to 6.
- Paragraph tag induces a line break.

Examples:

```
<p>This text is going to be a paragraph one day and separated from other text by line breaks.</p>
```

```
<h1>heading of level 1 - this will be BIG</h1>
...
<h6>heading of level 6 - the smallest heading</h6>
```

Important tags and attributes

Listing tags `<ul>`, `<ol>`, and `<dl>`

- The `<ol>` tag creates a numeric list.
- The `<ul>` tag creates an unnumbered list.
- The `<dl>` tag creates a description list.
- List elements within `<ol>` and `<ul>` are indicated with the `<li>` tag.

Example:

```
<ul>
  <li>Dogs</li>
  <li>Cats</li>
  <li>Fish</li>
</ul>
```

Important tags and attributes

Organizational and styling tags `<div>` and `<span>`

- They are used to group content over lines (`<div>`, creating a block-level element) or within lines (`<span>`, creating an inline-element).
- By grouping or dividing content into blocks, it's easier to identify or apply different styling to them.
- They do not change the layout themselves but work together with CSS (see later!).

Example of CSS definition:

```css
div.happy {
  color:pink;
  font-family:"Comic Sans MS";
  font-size:120%
}
span.happy {
  color:pink;
  font-family:"Comic Sans MS";
  font-size:120%
}
```

In the HTML document:

```html
<div class="happy">
  <p>I am a happy-styled paragraph</p>
</div>

unhappy text with <span class="happy">some
happiness</span>
```

Important tags and attributes

Form tag `<form>`

- Allows to incorporate HTML forms.
- Client can send information to the server via forms.
- Whenever you type something into a field or click on radio buttons in your browser, you are interacting with forms.

Example:

```html
<form name="submitPW" action="Passed.html" method="get">
  password:
  <input name="pw" type="text" value="">
  <input type="submit" value="SubmitButtonText"
</form>
```

Important tags and attributes

Table tags `<table>`, `<tr>`, `<td>`, and `<th>`

- Standard HTML tables always follow a standard architecture.
- The different tags allow defining the table as a whole, individual rows (including the heading), and cells.
- If the data is hidden in tables, scraping will be straightforward.

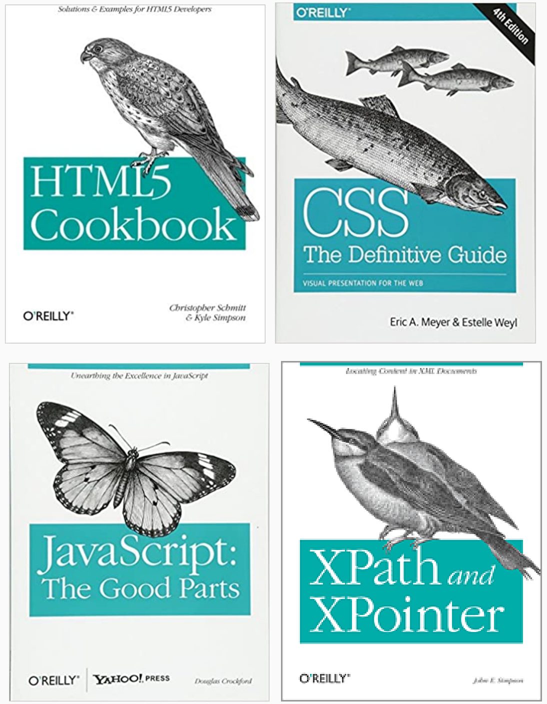
Example:

```
<table>
  <tr> <th>Rank</th> <th>Nominal GDP</th> <th>Name</th> </tr>
  <tr> <th></th> <th>(per capita, USD)</th> <th></th> </tr>
  <tr> <td>1</td> <td>170,373</td> <td>Lichtenstein</td> </tr>
  <tr> <td>2</td> <td>167,021</td> <td>Monaco</td> </tr>
  <tr> <td>3</td> <td>115,377</td> <td>Luxembourg</td> </tr>
  <tr> <td>4</td> <td>98,565</td> <td>Norway</td> </tr>
  <tr> <td>5</td> <td>92,682</td> <td>Qatar</td> </tr>
</table>
```

More resources on HTML

More HTML

- All in all there are over 100 HTML elements.
- But overall, it's still a fairly tight and easy-to-understand markup language.
- Knowing more about the rest is probably not necessary to become a good web scraper, but it helps parsing (in your brain) HTML documents quicker.



More resources

- Check out the excellent MDN Web Docs for an overview, which also point to additional tutorials and references.
- The W3Schools tutorials are also a classic.
- While you're at it, you might also want to learn about related technologies such as CSS (used to specify a webpage's appearance/layout) and JavaScript (used to enrich HTMLs with additional functionality and options to interact).

Accessing the web using your browser vs. R

Using your browser to access webpages

1. You click on a link, enter a URL, run a Google query, etc.
2. Browser/your machine sends request to server that hosts website.
3. Server returns resource (often an HTML document).
4. Browser interprets HTML and renders it in a nice fashion.

Using R to access webpages

1. You manually specify a resource.
2. R/your machine sends a request to the server that hosts the website.
3. The server returns a resource (e.g., an HTML file).
4. R parses the HTML, but does not render it in a nice fashion.
5. It's up to you to tell R what content to extract.

Interacting with your browser

On web browsers

- Modern browsers are complex pieces of software that take care of multiple operations while you browse the web. And they're basically all doing a good job.[1] Common operations are to retrieve resources, render and display information, and provide interface for user-webpage interaction.
- Although our goal is to automate web data retrieval, the browser is an important tool in web scraping workflow.

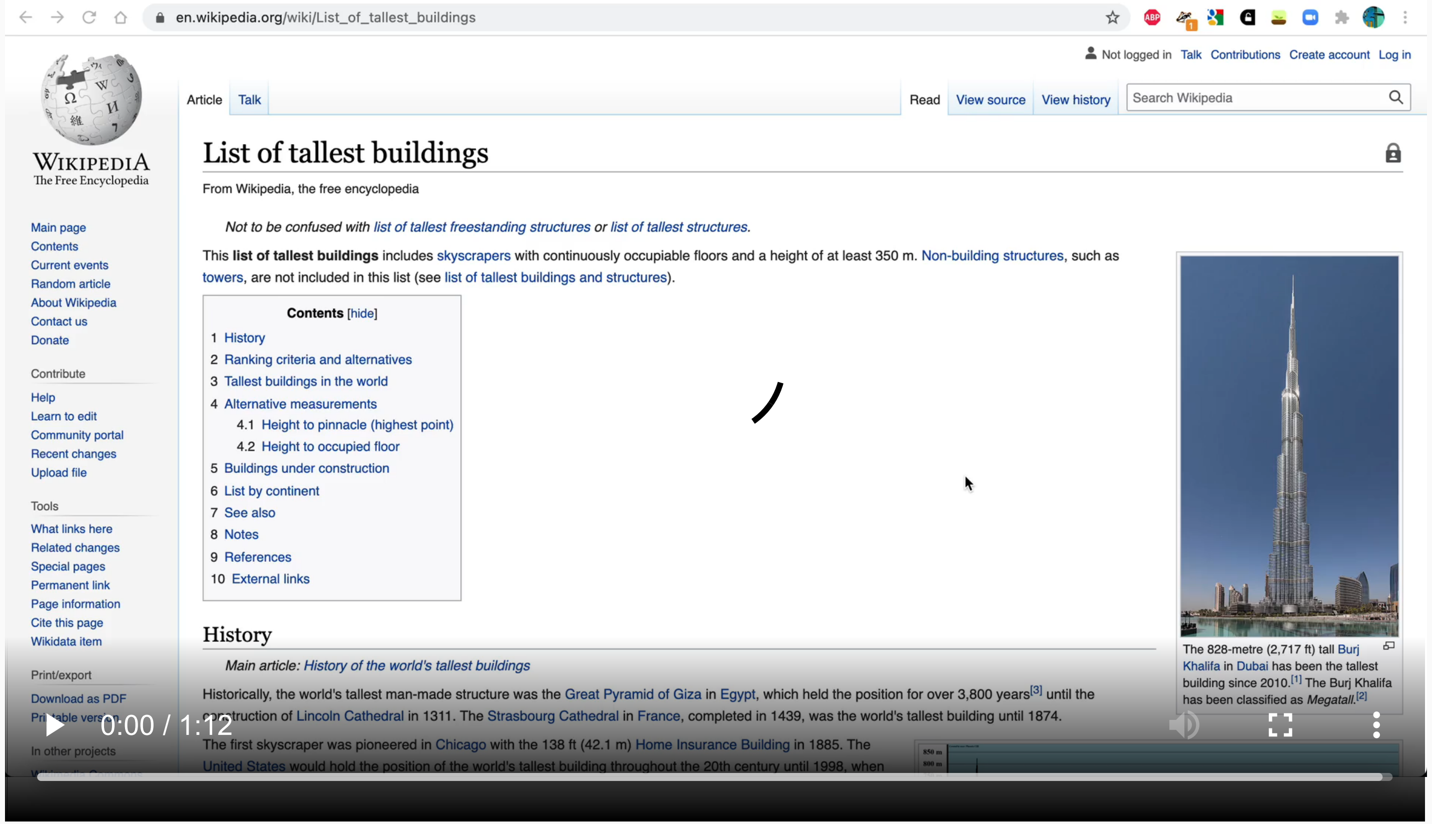
The use of browsers for web scraping

- Give you an intuitive impression of the architecture of a webpage
- Allow you to inspect the source code
- Let you construct XPath/CSS selector expressions with plugins
- Render dynamic web content (JavaScript interpreter)

[1] Check out this Wikipedia article on the Browser Wars that happened in the 1990s and 2000s (yes, there was Browser War I and Browser War II - and for once Germany was not to blame) to relive some of your instructor's pains when he started to look into this "internet".

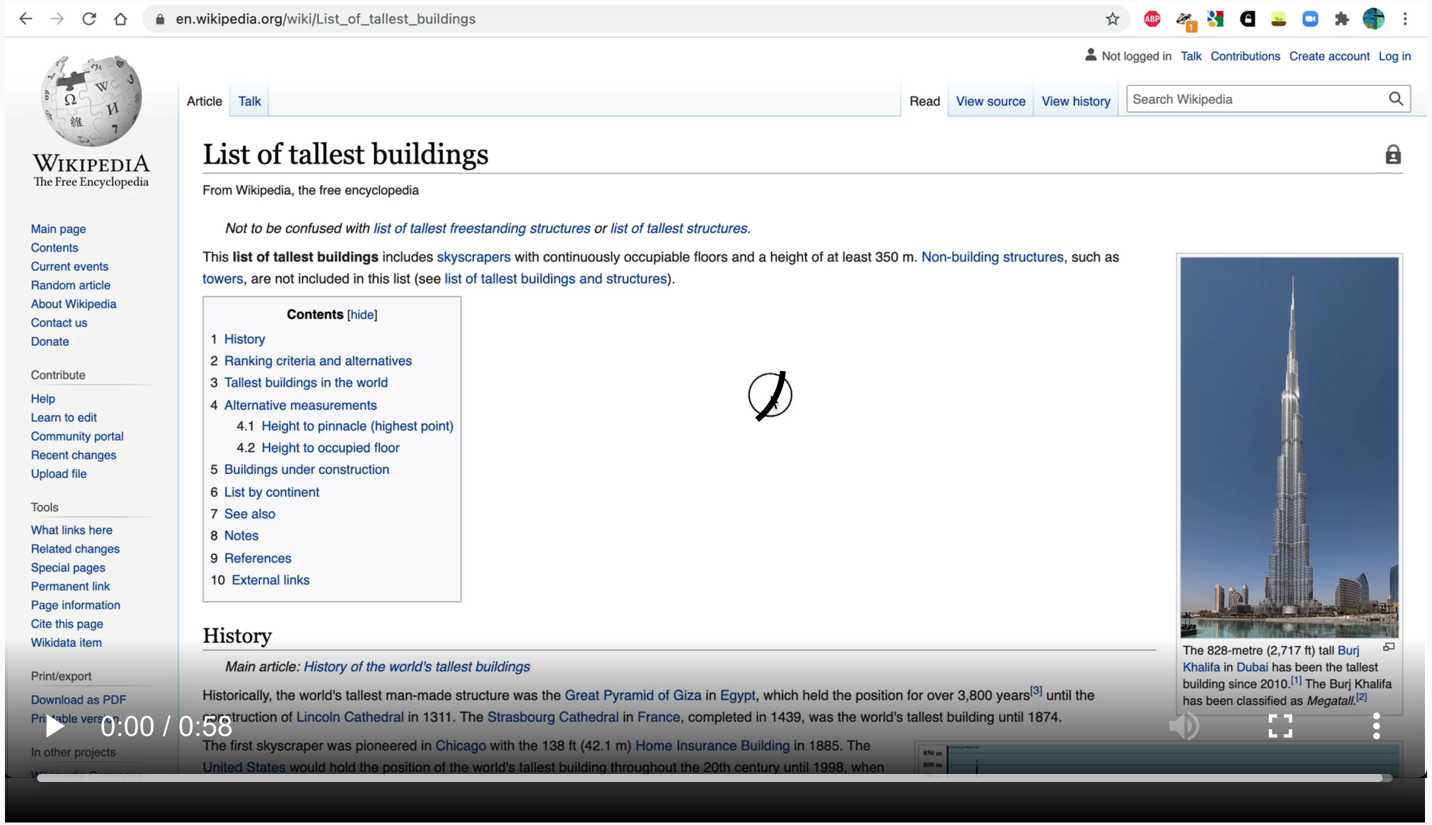
Inspecting HTML source code

- Goal: retrieving data from a Wikipedia page on List of tallest buildings
- Right-click on page (anywhere)
- Select `View Page Source`
- HTML (CSS, JavaScript) code can be ugly
- But looking more closely, we find the displayed information



Inspecting the live HTML source code with the DOM

- Goal: retrieving data from a Wikipedia page on List of tallest buildings
- Right-click on the element of interest
- Select `Inspect`
- The Web Developer Tools window pops up
- Corresponding part in the HTML tree is highlighted
- Interaction with the tree possible!



When to do what with your browser

When to inspect the complete page source

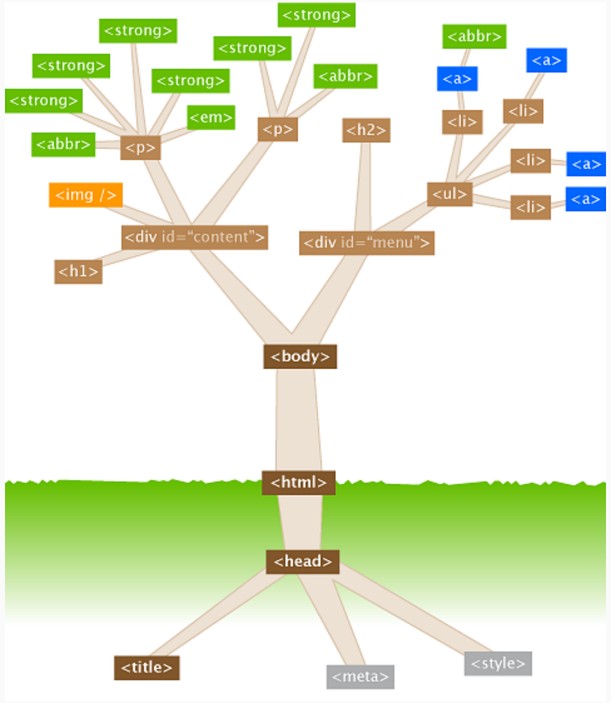
- Check whether data is in static source code (the search function helps!)
- For small HTML files: understand structure

When to use the DOM explorer

- Almost always
- Particularly useful to construct XPath/CSS selector expressions
- To monitor dynamic changes in the DOM tree

A note on browser differences

- Inspecting the source code (as shown on the following slides) works more or less identically in Chrome and Firefox.
- In Safari, go to → `Preferences`, then → `Advanced` and select `Show Develop menu in menu bar`. This unlocks the `Show Page Source` and `Inspect` options and the Web Developer Tools.



`Credit` watershedcreative.com

XPath basics

Accessing the DOM tree with R

Different perspectives on HTML

- HTML documents are human-readable.
- HTML tags structure the document, comprising the DOM.
- **Web user perspective**: The browser interprets the code and renders the page.
- **Web scraper perspective**: Parse the document retaining the structure, use the tree/tags to locate information.

Accessing the DOM tree with R

Different perspectives on HTML

- HTML documents are human-readable.
- HTML tags structure the document, comprising the DOM.
- **Web user perspective**: The browser interprets the code and renders the page.
- **Web scraper perspective**: Parse the document retaining the structure, use the tree/tags to locate information.

HTML parsing

- Our goal is to get HTML into R while retaining the tree structure. That's similar to getting a spreadsheet into R and retaining the rectangular structure.
- HTML is human-readable, so we could also import HTML files as plain text via `readLines()`. That's a bad option though - the document's structure would not be retained.
- The `xml2` package allows us to parse XML-style documents. HTML is a "flavor" of XML, so it works for us.
- The `rvest` package, which we will mainly use for scraping, wraps the `xml2` package, so we rarely have to load it manually.
- There is one high-level function to remember: `read_html()`. It represents the HTML in a list-style fashion.

Accessing the DOM tree with R (cont.)

Getting HTML into R

Parsing a website is straightforward:

```
R> library(rvest)
R> parsed_doc <- read_html("https://google.com")
R> parsed_doc
```

```
## {html_document}
## <html lang="de" dir="ltr">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body>\n<div class="signin"><a href="https://accounts.google.com/ServiceL ...
```

There are various functions to inspect the parsed document. They aren't really helpful - better use the browser instead if you want to dive into the HTML.

```
R> xml2::html_structure(parsed_doc)
R> xml2::as_list(parsed_doc)
```

What's XPath?

Definition

- Short for **XML Path Language**, another W3C standard.
- A query language for XML-based documents (including HTML).
- With XPath we can access node sets (e.g., elements, attributes) and extract content.

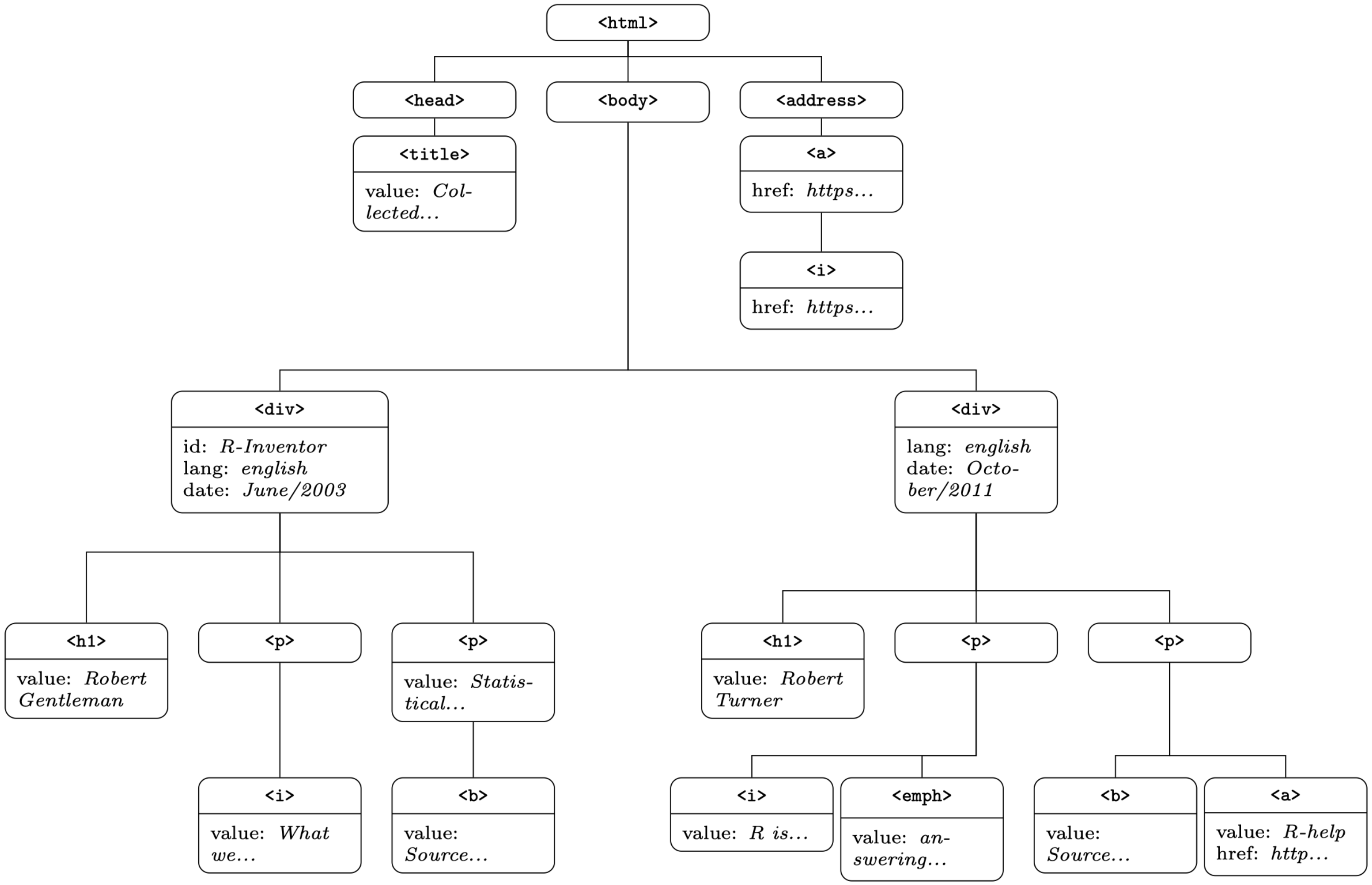
Why XPath for web scraping?

- Source code of webpages (HTML) structures both layout and content.
- Not only content, but context matters!
- XPath enables us to extract content based on its location in the document (and potentially other features).
- With XPath, we can tell R to do things like:
    1. Give me all `<li>` elements in the document!
    2. Look for all `<table>` elements in the document and give me the third one!
    3. Extract all content in `<p>` elements that is labelled with `class=newscontent`!

Example: source code

```html
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
 <head>
  <title>Collected R wisdoms</title>
 </head>
 <body>
  <div id="R Inventor" lang="english" date="June/2003">
   <h1>Robert Gentleman</h1>
    <p><i>'What we have is nice, but we need something very different'</i></p>
    <p><b>Source: </b>Statistical Computing 2003, Reisensburg</p>
  </div>
  <div lang="english" date="October/2011">
   <h1>Rolf Turner</h1>
    <p><i>'R is wonderful, but it cannot work magic'</i>
    <br><emph>answering a request for automatic generation of 'data from a known mean and 95% CI'</emph></p>
    <p><b>Source: </b><a href="https://stat.ethz.ch/mailman/listinfo/r-help">R-help</a></p>
  </div>
  <address>
   <a href="http://www.rdatacollectionbook.com"><i>The book homepage</i></a>
  </address>
 </body>
</html>
```

Example: DOM tree



Applying XPath on HTML in R

- Load package `rvest`
- Parse HTML document with `read_html()`

```
R> library(rvest)
R> parsed_doc <- read_html("materials/fortunes.html")
R> parsed_doc
```

```
## {html_document}
## <html>
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body>\n<div id="R Inventor" lang="english" date="June/2003">\n  <h1>Robe ...
```

- Query document using `html_elements()`
- `rvest` can process XPath queries as well as CSS selectors.
- Today, we'll focus on XPath:

```
R> html_elements(parsed_doc, xpath = "//div[last()]/p/i")
```

```
## {xml_nodeset (1)}
## [1] <i>'R is wonderful, but it cannot work magic'</i>
```

Grammar of XPath

Basic rules

1. We access nodes/elements by writing down the hierarchical structure in the DOM that locates the element set of interest.
2. A sequence of nodes is separated by `/`.
3. The easiest localization of a element is given by the absolute path (but often not the most efficient one!).
4. Apply XPath on DOM in R using `html_elements()`.

```
R> html_elements(parsed_doc, xpath = "//div[last()]/p/i")
```

```
## {xml_nodeset (1)}
## [1] <i>'R is wonderful, but it cannot work magic'</i>
```

Grammar of XPath

Absolute vs. relative paths

Absolute paths start at the root element and follow the whole way down to the target element (with simple slashes, `/`).

```
R> html_elements(parsed_doc, xpath = "/html/body/div/p/i")
```

```
## {xml_nodeset (2)}
## [1] <i>'What we have is nice, but we need something very different'</i>
## [2] <i>'R is wonderful, but it cannot work magic'</i>
```

Relative paths skip nodes (with double slashes, `//`).

```
R> html_elements(parsed_doc, xpath = "//body//p/i")
```

```
## {xml_nodeset (2)}
## [1] <i>'What we have is nice, but we need something very different'</i>
## [2] <i>'R is wonderful, but it cannot work magic'</i>
```

Relative paths are often preferrable. They are faster to write and more comprehensive. On the other hand, they are less targeted and therefore potentially less robust, and running them takes more computing time, as the entire tree has to be evaluated. But that's usually not relevant for reasonably small documents.

Grammar of XPath

The wildcard operator

- Meta symbol `*`
- Matches any element
- Works only for one arbitrary element
- Far less important than, e.g., wildcards in content-based queries (regex!)

```
R> html_elements(parsed_doc, xpath = "/html/body/div/*/i")
```

```
## {xml_nodeset (2)}
## [1] <i>'What we have is nice, but we need something very different'</i>
## [2] <i>'R is wonderful, but it cannot work magic'</i>
```

```
R> # the following does not work:
R> html_elements(parsed_doc, xpath = "/html/body/div/*/i")
```

```
## {xml_nodeset (2)}
## [1] <i>'What we have is nice, but we need something very different'</i>
## [2] <i>'R is wonderful, but it cannot work magic'</i>
```

Grammar of XPath

Navigational operators `"."` and `".."`

- `"."` accesses elements on the same level ("self axis"), which is useful when working with predicates (see later!).
- `".."` accesses elements at a higher hierarchical level.

```
R> html_elements(parsed_doc, xpath = "//title/..")
```

```
## {xml_nodeset (1)}
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
```

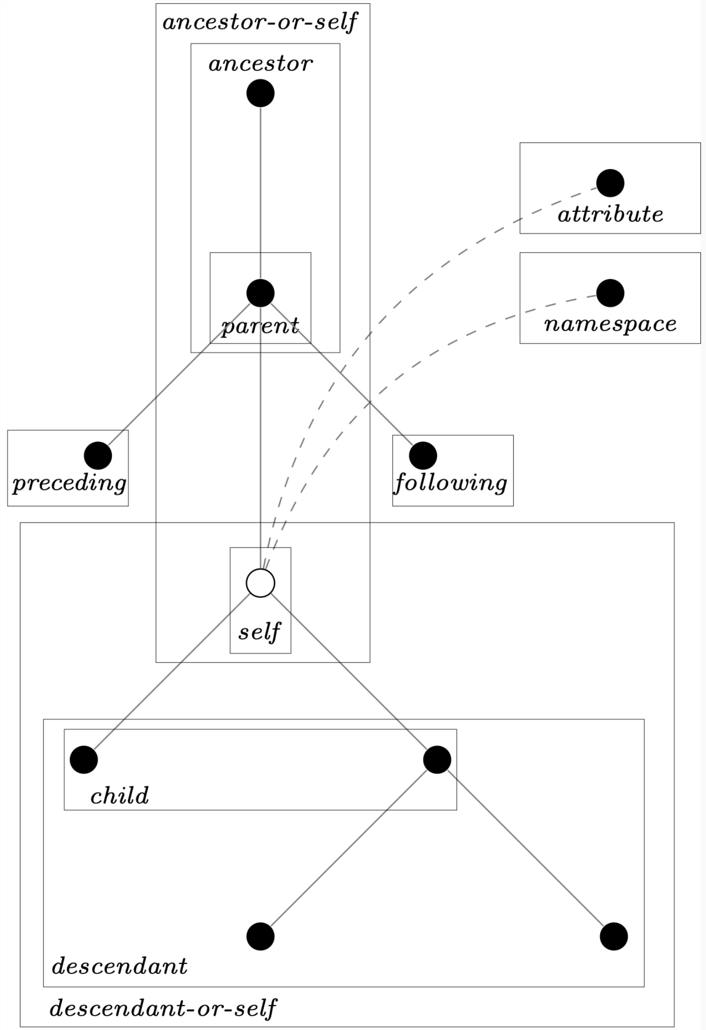
```
R> html_elements(parsed_doc, xpath = "//div[starts-with(./@id, 'R')]")
```

```
## {xml_nodeset (1)}
## [1] <div id="R Inventor" lang="english" date="June/2003">\n  <h1>Robert Gentl ...
```

Element (node) relations ("axes") in XPath

Family relations between elements

- The tools learned so far are sometimes not sufficient to access specific elements without accessing other, undesired elements as well.
- Relationship statuses are useful to establish unambiguity.
- Can be combined with other elements of the grammar
- Basic syntax: `element1/relation::element2`
- We describe relation of `element2` to `element1`
- `element2` is to be extracted - we always extract the element at the end!



Element (node) relations in XPath

Axis name	Description
`ancestor`	All ancestors (parent, grandparent etc.) of the current element
`ancestor-or-self`	All ancestors of the current element and the current element itself
`attribute`	All attributes of the current element
`child`	All children of the current element
`descendant`	All descendants (children, grandchildren etc.) of the current element
`descendant-or-self`	All descendants of the current element and the current element itself
`following`	Everything in the document after the closing tag of the current element
`following-sibling`	All siblings after the current element
`parent`	The parent of the current element
`preceding`	All elements that appear before the current element, except ancestors/attribute elements
`preceding-sibling`	All siblings before the current element
`self`	The current element

Element (node) relations in XPath

Example: access the `<div>` elements that are ancestors to an `<a>` element:

```
R> html_elements(parsed_doc, xpath = "//a/ancestor::div")
```

```
## {xml_nodeset (1)}
## [1] <div lang="english" date="October/2011">\n  <h1>Rolf Turner</h1>\n  <p><i ...
```

Another example: Select all `<h1>` nodes that precede a `<p>` node:

```
R> html_elements(parsed_doc, xpath = "//p/preceding-sibling::h1")
```

```
## {xml_nodeset (2)}
## [1] <h1>Robert Gentleman</h1>
## [2] <h1>Rolf Turner</h1>
```

Predicates

What are predicates?

- Predicates are conditions based on an element's features (`true/false`).
- Think of them as ways to filter nodesets.
- They are applicable to a variety of features: name, value attribute.
- Basic syntax: `element[predicate]`

Select all first `<p>` elements that are children of a `<div>` element, using a **numeric predicate**:

```
R> html_elements(parsed_doc, xpath = "//div/p[1]")
```

```
## {xml_nodeset (2)}
## [1] <p><i>'What we have is nice, but we need something very different'</i></p>
## [2] <p><i>'R is wonderful, but it cannot work magic'</i> <br><emph>answering  ...
```

Predicates

What are predicates?

- Predicates are conditions based on an element's features (`true/false`).
- Think of them as ways to filter nodesets.
- They are applicable to a variety of features: name, value attribute.
- Basic syntax: `element[predicate]`

Select all first `<p>` elements that are children of a `<div>` element, using a **numeric predicate**:

```
R> html_elements(parsed_doc, xpath = "//div/p[1]")
```

```
## {xml_nodeset (2)}
## [1] <p><i>'What we have is nice, but we need something very different'</i></p>
## [2] <p><i>'R is wonderful, but it cannot work magic'</i> <br><emph>answering  ...
```

Can you find out what the following expressions do?

```
R> html_elements(parsed_doc, xpath = "//div/p[last()-1]")
R> html_elements(parsed_doc, xpath = "//div[count(./@*)>2]")
R> html_elements(parsed_doc, xpath = "//*[string-length(text())>50]")
```

Predicates (cont.)

Select all `<div>` nodes that contain an attribute named `'October/2011'`, using a **textual predicate**:

```
R> html_elements(parsed_doc, xpath ="//div[@date='October/2011']")
```

```
## {xml_nodeset (1)}
## [1] <div lang="english" date="October/2011">\n  <h1>Rolf Turner</h1>\n  <p><i ...
```

Rudimentary string matching is also possible using string functions like `contains()`, `starts-with()`, or `ends-with()`.

Predicates (cont.)

Select all `<div>` nodes that contain an attribute named `'October/2011'`, using a **textual predicate**:

```
R> html_elements(parsed_doc, xpath ="//div[@date='October/2011']")
```

```
## {xml_nodeset (1)}
## [1] <div lang="english" date="October/2011">\n  <h1>Rolf Turner</h1>\n  <p><i ...
```

Rudimentary string matching is also possible using string functions like `contains()`, `starts-with()`, or `ends-with()`.

Can you tell what the following calls do?

```
R> html_elements(parsed_doc, xpath = "//div[starts-with(./@id, 'R')]")
R> html_elements(parsed_doc, xpath = "//div[substring-after(./@date, '/')='2003']//i")
```

Content extraction

- Until now, we used XPath expressions to extract complete nodes or nodesets (that is, elements with tags).
- However, in most cases we're interested in extracting the content only.
- To that end, we can use extractor functions that are applied on the output of XPath query calls.

Function	Argument	Return value
`html_text()`		Element value
`html_text2()`		Element value (with a bit more cleanup)
`html_attr()`	`name`	Element attribute
`html_attrs()`		(All) element attributes
`html_name()`	`trim`	Element name
`html_children()`		Element children

Content extraction (cont.)

Extracting **element values/content**:

```
R> html_elements(parsed_doc, xpath = "//title") %>% html_text2()
```

```
## [1] "Collected R wisdoms"
```

Extracting **attributes**:

```
R> html_elements(parsed_doc, xpath = "//div[1]") %>% html_attrs()
```

```
## [[1]]
##            id          lang          date
## "R Inventor"     "english"   "June/2003"
```

Extracting **attribute values**:

```
R> html_elements(parsed_doc, xpath = "//div") %>% html_attr("lang")
```

```
## [1] "english" "english"
```

More XPath?

Training resources

- XPath is a little language of its own. As always with languages, mastery comes with practice.
- A good environment for practice is the XPath expression testbed at whitebeam.org.
- Also check out this cheat sheet.

XPath creator tools

- Now, do you really have to construct XPath expressions by your own? No! At least not always.
- **SelectorGadget**: http://selectorgadget.com is a browser plugin that constructs XPath statements via a point-and-click approach. The generated expressions are not always efficient and effective though (more on this later).
- Web developer tools - the internal browser functionality to study the DOM, among other things, also lets you extract XPath statements for selected nodes. These are specific to unique nodes/elements though, and therefore less helpful to extract node sets. (But they come in handy when we want to script live navigation, e.g. for Selenium.)

CSS basics

What is CSS?

Background

- **C**ascading **S**tyle **S**heets (CSS) is a style sheet language that allows web developers to adjust the "look and feel" of websites.
- By using CSS to adjust style features such as layout, colors, and fonts, it's easier to separate content (HTML) from presentation (CSS).

Three ways to insert CSS into HTML

1. **External CSS.** Inside `<head>` with a reference to the external file inside the `<link>` element.
2. **Internal CSS.** Inside `<head>` and stored in `<style>` elements.
3. **Inline CSS.** Inside `<body>` using the `style` attribute of elements.

What is CSS?

Background

- **C**ascading **S**tyle **S**heets (CSS) is a style sheet language that allows web developers to adjust the "look and feel" of websites.
- By using CSS to adjust style features such as layout, colors, and fonts, it's easier to separate content (HTML) from presentation (CSS).

Three ways to insert CSS into HTML

1. **External CSS.** Inside `<head>` with a reference to the external file inside the `<link>` element.
2. **Internal CSS.** Inside `<head>` and stored in `<style>` elements.
3. **Inline CSS.** Inside `<body>` using the `style` attribute of elements.

External CSS

```
<head>
  <link rel="stylesheet" href="mystyle.css">
</head>
```

Internal CSS

```
<head>
   <style>
      h1 {
         color: red;
         margin-left: 20px;
      }
   </style>
</head>
```

Inline CSS

```
<p style="color: blue;">This is a paragraph.</p>
```

CSS selectors

Selectors

- CSS selectors find/select the HTML elements that should be styled.
- There are various categories of selectors. In addition to generic element selectors (which selected just based on the element name, such as `<p>`), we often care about:
  - **CSS id selectors**, which use the `id` attribute of an HTML element. Think of them as "labels", as in `<p id="para1">`. The respective CSS selector would be `#para1`.
  - **CSS class selectors**, which use the `class` attribute of an HTML element, as in `<p class = "center large">`. Note that these can refer to more than one class (here: `center` and `large`). The respective CSS selector would be `p.center.large`.

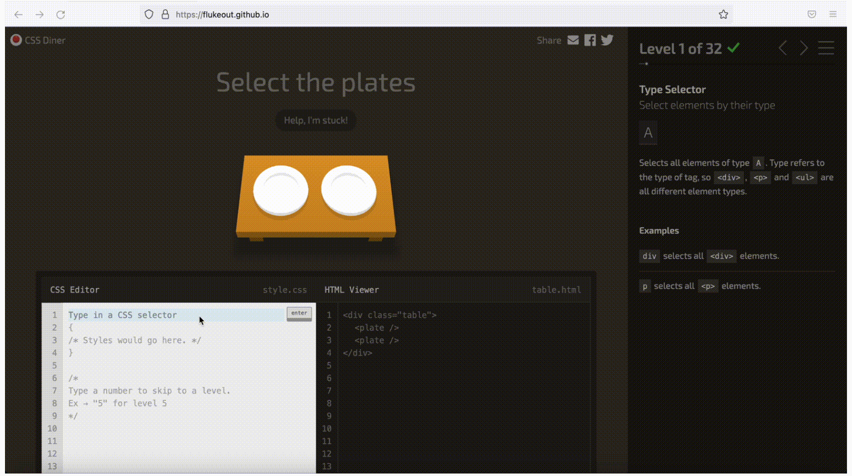
CSS selectors

Selectors

- CSS selectors find/select the HTML elements that should be styled.
- There are various categories of selectors. In addition to generic element selectors (which selected just based on the element name, such as `<p>`), we often care about:
  - **CSS id selectors**, which use the `id` attribute of an HTML element. Think of them as "labels", as in `<p id="para1">`. The respective CSS selector would be `#para1`.
  - **CSS class selectors**, which use the `class` attribute of an HTML element, as in `<p class = "center large">`. Note that these can refer to more than one class (here: `center` and `large`). The respective CSS selector would be `p.center.large`.

Writing CSS selectors

- Just as XPath, CSS selectors are a little language of their own.
- I won't teach you more about it, but you might nevertheless want to learn it.
- Check out the CSS diner tutorial at https://flukeout.github.io/. It's one of the best tutorials of anything out there.



Scraping static webpages with R

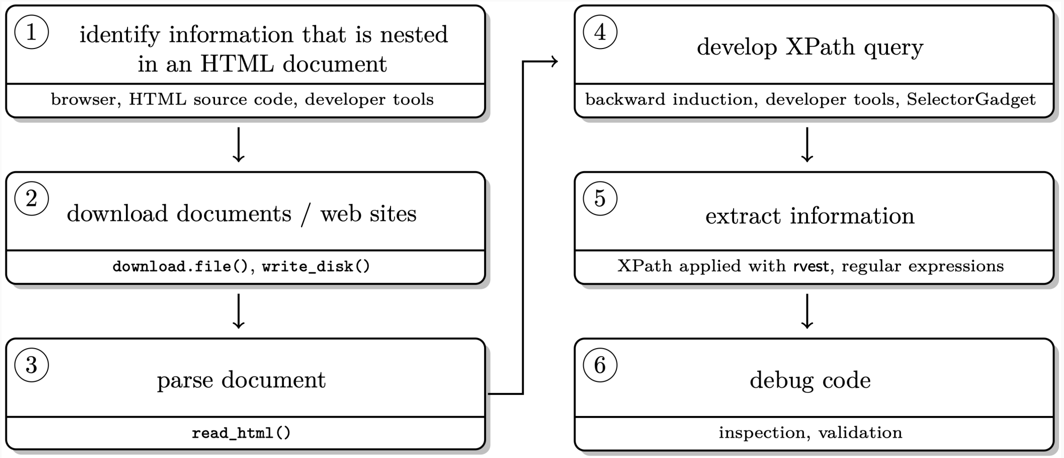
The scraping workflow

Key tools for scraping static webpages

1. You are able to inspect HTML pages in your browser using the web developer tools.
2. You are able to parse HTML into R with `rvest`.
3. You are able to speak XPath (or CSS selectors).
4. You are able to apply XPath expressions with `rvest`.
5. You are able to tidy web data with R/`dplyr`/`regex`.

The big picture

- Every scraping project is different, but the coding pipeline is fundamentally similar.
- The (technically) hardest steps are location (XPath, CSS selectors) and extraction (clean-up), sometimes the scaling (from one to multiple sources).



Web scraping with rvest

`rvest` is a suite of scraping tools. It is part of the tidyverse and has made scraping with R much more convenient.

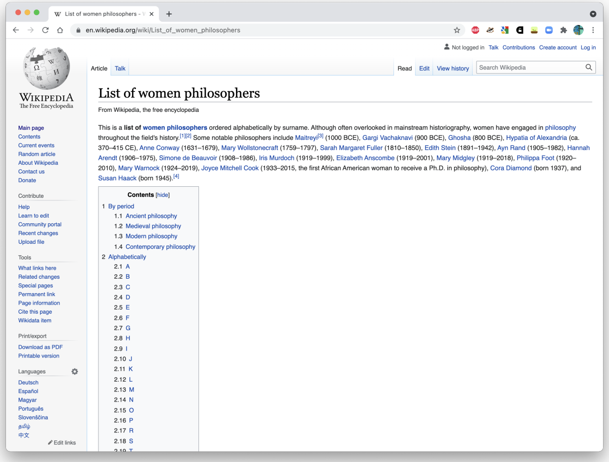
There are three key `rvest` verbs that you need to learn.[1]

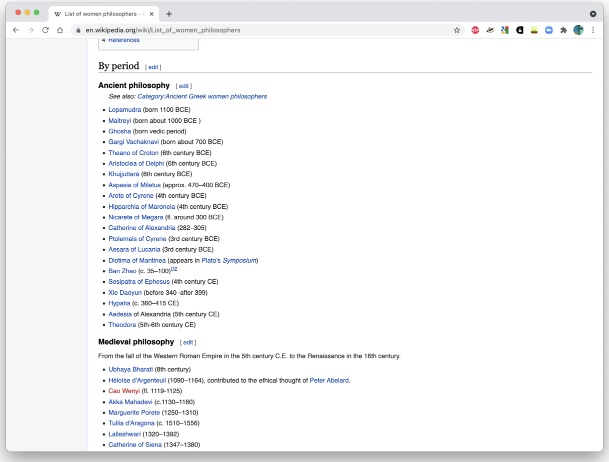
1. `read_html()`: Read (parsing) an HTML resource.
2. `html_elements()`: Find elements that match a CSS selector or XPath expression.
3. `html_text2()`: Extract the text/value inside the node set.

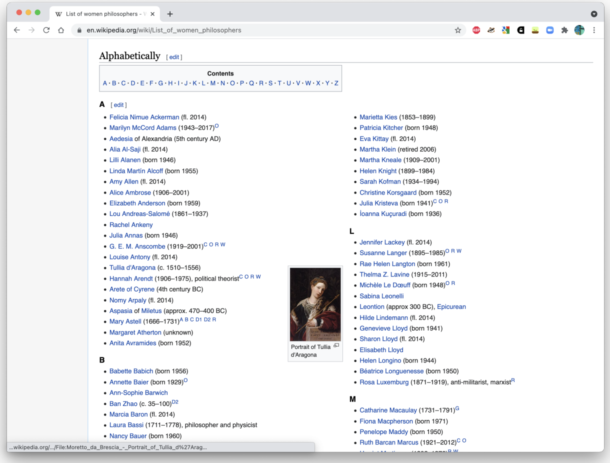
[1] There is more in `rvest` than what we can cover today. Have a glimpse at the overview at tidyverse.org and at this excellent (unofficial) cheat sheet.

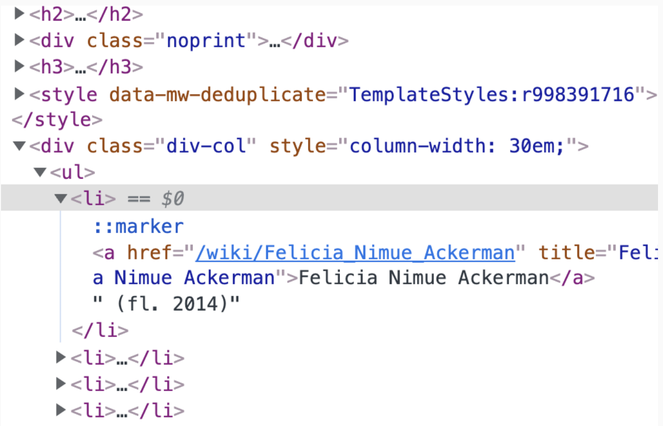
Web scraping with rvest: example

- We are going to scrape a information from a Wikipedia article on women philosophers available at https://en.wikipedia.org/wiki/List_of_women_philosophers.
- The article provides two types of lists - one by period and one sorted alphabetically. We want the alphabetical list.
- The information we are actually interested in - names - is stored in unordered list elements.









Scraping HTML tables: example (cont.)

Step 1: Parse the page

```
R> url_p <- read_html("https://en.wikipedia.org/wiki/List_of_women_philosophers")
```

Scraping HTML tables: example (cont.)

Step 1: Parse the page

```
R> url_p ← read_html("https://en.wikipedia.org/wiki/List_of_women_philosophers")
```

Step 2: Develop an XPath expression (or multiple) that select the information of interest and apply it

```
R> elements_set ← html_elements(url_p, xpath = "//h2/span[text()='Alphabetically']//following::li/a[1]")
```

Scraping HTML tables: example (cont.)

Step 1: Parse the page

```
R> url_p ← read_html("https://en.wikipedia.org/wiki/List_of_women_philosophers")
```

Step 2: Develop an XPath expression (or multiple) that select the information of interest and apply it

```
R> elements_set ← html_elements(url_p, xpath = "//h2/span[text()='Alphabetically']//following::li/a[1]")
```

The XPath expression reads:

- `//h2`: Look for `h2` elements anywhere in the document.
- `/span[text()='Alphabetically']`: Within that element look for `span` elements with the content `"Alphabetically"`.
- `//following::li`: In the DOM tree following that element (at any level), look for `li` elements.
- `/a[1]` within these elements look for the first `a` element you can find.

Scraping HTML tables: example (cont.)

Step 3: Extract information and clean it up

```
R> phil_names ← elements_set %>% html_text2()
R> phil_names[c(1:2, 101:102)]
```

```
## [1] "A"                    "B"                    "Elisabeth of Bohemia"
## [4] "Dorothy Emmet"
```

Scraping HTML tables: example (cont.)

Step 3: Extract information and clean it up

```
R> phil_names <- elements_set %>% html_text2()
R> phil_names[c(1:2, 101:102)]
```

```
## [1] "A"                    "B"                    "Elisabeth of Bohemia"
## [4] "Dorothy Emmet"
```

Step 4: Clean up (here: select the subset of links we care about)

```
R> names_iffer <-
+   seq_along(phil_names) >=  seq_along(phil_names)[str_detect(phil_names, "Felicia Nimue Ackerman")] &
+   seq_along(phil_names) <=   seq_along(phil_names)[str_detect(phil_names, "Alenka Zupančič")]
R> philosopher_names_clean <- phil_names[names_iffer]
R> length(philosopher_names_clean)
```

```
## [1] 267
```

```
R> philosopher_names_clean[1:5]
```

```
## [1] "Felicia Nimue Ackerman" "Marilyn McCord Adams"   "Aedesia"
## [4] "Alia Al-Saji"           "Lilli Alanen"
```

Quick-n-dirty static webscraping with SelectorGadget

The hassle with XPath

- The most cumbersome part of web scraping (data tidying aside) is the construction of XPath expressions that match the components of a page you want to extract.
- It will take a couple of scraping projects until you'll truly have mastered XPath.

A much-appreciated helper

- **SelectorGadget** is a JavaScript browser plugin that constructs XPath statements (or CSS selectors) via a point-and-click approach.
- It is available here: http://selectorgadget.com/ (there is also a Chrome extension).
- The tool is magic and you will love it.

Quick-n-dirty static webscraping with SelectorGadget

The hassle with XPath

- The most cumbersome part of web scraping (data tidying aside) is the construction of XPath expressions that match the components of a page you want to extract.
- It will take a couple of scraping projects until you'll truly have mastered XPath.

A much-appreciated helper

- **SelectorGadget** is a JavaScript browser plugin that constructs XPath statements (or CSS selectors) via a point-and-click approach.
- It is available here: http://selectorgadget.com/ (there is also a Chrome extension).
- The tool is magic and you will love it.

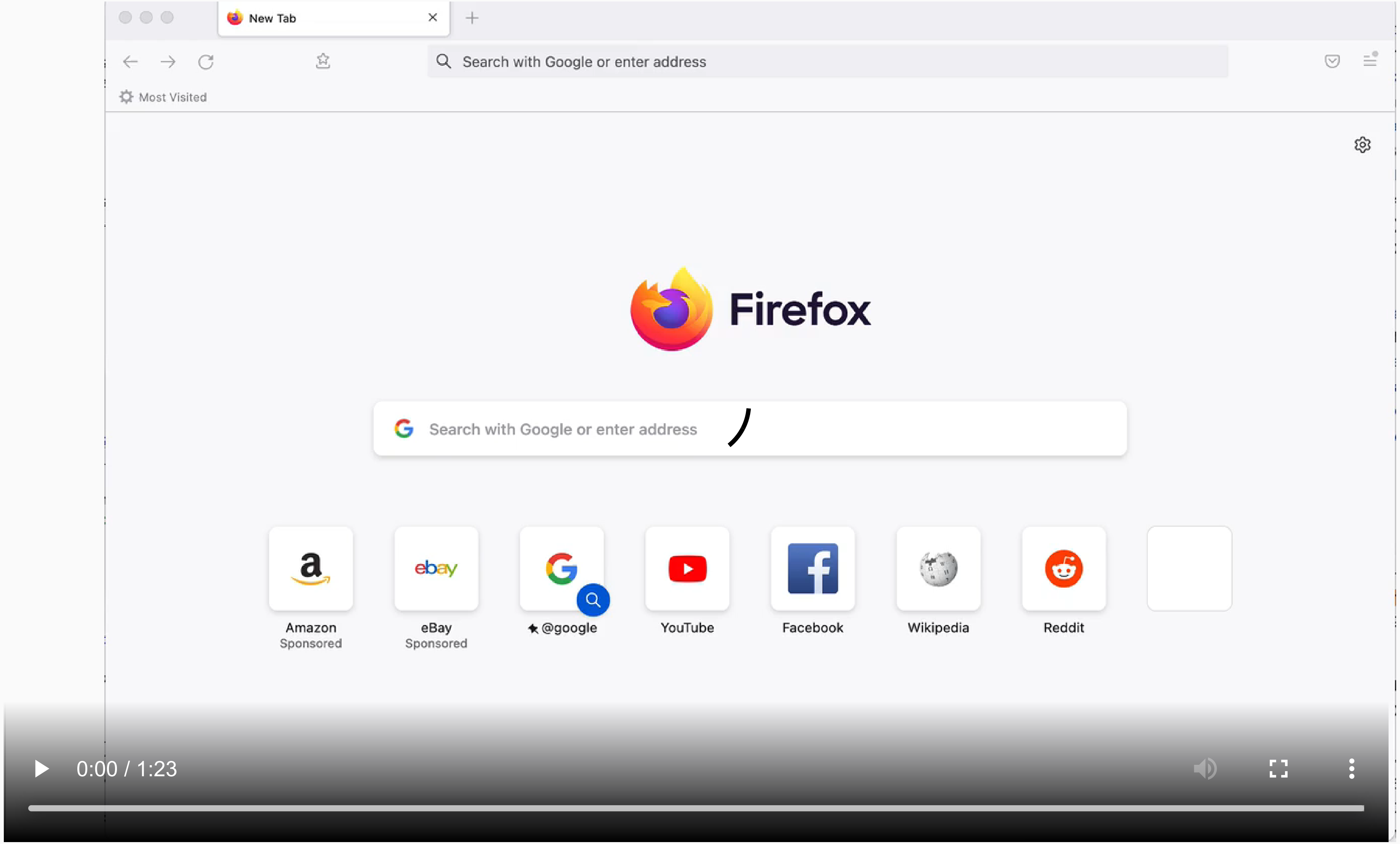
What does SelectorGadget do?

- You activate the tool on any webpage you want to scrape.
- Based on your selection of components, the tool learns about your desired components and generates an XPath expression (or CSS selector) for you.

Under the hood

- Based on your selection(s), the tool looks for similar elements on the page.
- The underlying algorithm, which draws on Google's diff-match-patch libraries, focuses on CSS characteristics, such as tag names and `<div>` and `<span>` attributes.

SelectorGadget: example



SelectorGadget: example (cont.)

```
R> library(rvest)
R> url_p <- read_html("https://www.nytimes.com")
R> # xpath: paste the expression from Selectorgadget!
R> # note: we use single quotation marks here (' instead of ") to wrap around the expression!
R> xpath <- '//*[contains(concat( " ", @class, " " ), concat( " ", "erslblw0", " " ))]//*[contains(concat( " ",
R> headlines <- html_elements(url_p, xpath = xpath)
R> headlines_raw <- html_text(headlines)
R> length(headlines_raw)
R> head(headlines_raw)
```

```
## [1] 29
```

```
## [1] "Retailers' Latest Headache: Shutdowns at Their Vietnamese SuppliersRetailers' Latest Headache: Shutdowns at T
## [2] "With virus restrictions waning, it's becoming clear: Britain's gas crisis is a Brexit crisis, too. Here's why
## [3] "Business updates: U.S. stock futures signaled a rebound as bond yields fell back."
## [4] "Republicans at Odds Over Infrastructure Bill as Vote ApproachesRepublicans at Odds Over Infrastructure Bill a
## [5] "Liberals Dig In Against Infrastructure Bill as Party Divisions Persist"
## [6] "Successful programs from around the world could guide Congress in designing a paid family leave plan."
```

SelectorGadget: when to use and not to use it

Having learned about a semi-automated approach to generating XPath expressions, you might ask:

Why bother with learning XPath at all?

Well...

- SelectorGadget is not perfect. Sometimes, the algorithm will fail.
- Starting from a different element sometimes (but not always!) helps.
- Often the generated expressions are unnecessarily complex and therefore difficult to debug.
- In my experience, SelectorGadget works 50-60% of the times when scraping from static webpages.
- You are also prepared for the remaining 40-50%!

Scraping HTML tables

Purchased Equipments (June, 2006)			
Item Num#	Item Picture	Item Description	Price
		Shipping Handling, Installation, etc	Expense
1.		IBM Clone Computer.	$ 400.00
		Shipping Handling, Installation, etc	$ 20.00
2.		1GB RAM Module for Computer.	$ 50.00
		Shipping Handling, Installation, etc	$ 14.00
Purchased Equipments (June, 2006)			

Built	Building	City	Country	Roof		Floors	Pinnacle		Current status
1870	Equitable Life Building	New York City	United States	043 m	142 ft	8			Destroyed by fire in 1912
1889	Auditorium Building	Chicago		082 m	269 ft	17	106 m	349 ft	Standing
1890	New York World Building	New York City		094 m	309 ft	20	106 m	349 ft	Demolished in 1955
1894	Philadelphia City Hall	Philadelphia		155.8 m	511 ft	9	167 m	548 ft	Standing
1908	Singer Building	New York City		187 m	612 ft	47			Demolished in 1968
1909	Met Life Tower			213 m	700 ft	50			Standing
1913	Woolworth Building			241 m	792 ft	57			Standing
1930	40 Wall Street					70	283 m	927 ft	Standing
1930	Chrysler Building			282.9 m	927 ft	77	319 m	1,046 ft	Standing
1931	Empire State Building			381 m	1,250 ft	102	443 m	1,454 ft	Standing
1972	World Trade Center (North Tower)			417 m	1,368 ft	110	527.3 m	1,730 ft	Destroyed in 2001 in the September 11 attacks
1974	Willis Tower (formerly Sears Tower)	Chicago		442 m	1,450 ft	108	527 m	1,729 ft	Standing
1996	Petronas Towers	Kuala Lumpur	Malaysia	379 m	1,242 ft	88	452 m	1,483 ft	Standing
2004	Taipei 101	Taipei	Taiwan	449 m	1,474 ft	101	509 m	1,671 ft	Standing
2010	Burj Khalifa	Dubai	United Arab Emirates	828 m	2,717 ft	163	829.8 m	2,722 ft	Standing

DATES	POLLSTER	GRADE	SAMPLE	WEIGHT	APPROVE	DISAPPROVE	ADJUSTED	
DEC. 28-30	Gallup	B-	1,500 A	1.03	40%	55%	41%	53%
DEC. 26-28	Rasmussen Reports/Pulse Opinion Research	C+	1,500 LV	0.85	45%	53%	40%	53%
DEC. 24-28	Ipsos	A-	1,519 A	2.01	37%	58%	37%	57%
DEC. 23-27	Gallup	B-	1,500 A	0.58	38%	56%	39%	54%
DEC. 24-26	YouGov	B	1,500 A	1.13	38%	52%	39%	55%

Scraping HTML tables

- HTML tables are everywhere.
- They are easy to spot in the wild - just look for `<table>` tags!
- Exactly because scraping tables is an easy and repetitive task, there is a dedicated `rvest` function for it: `html_table()`.

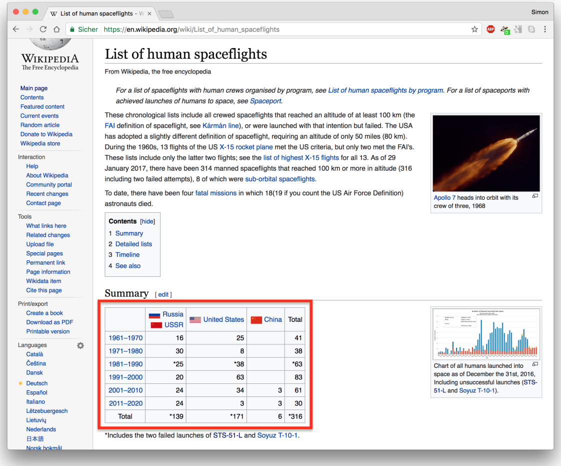
Function definition

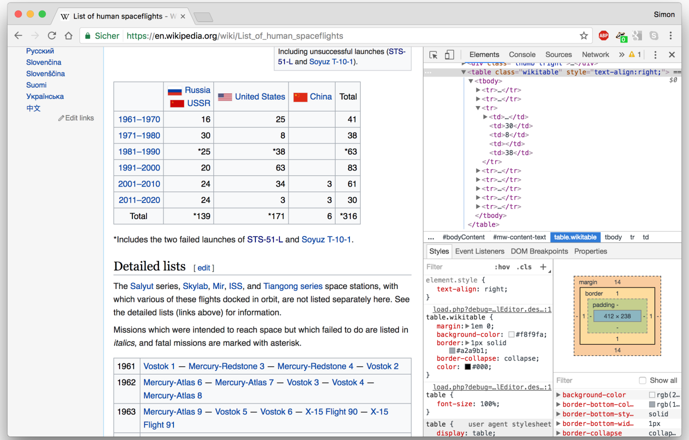
```
R> html_table(x,
+     header = NA,
+     trim = TRUE,
+     dec = ".",
+     na.strings = "NA",
+     convert = TRUE
+   )
```

Argument	Description
`x`	Document (from `read_html()`) or node set (from `html_elements()`).
`header`	Use first row as header? If `NA`, will use first row if it consists of `<th>` tags.
`trim`	Remove leading and trailing whitespace within each cell?
`dec`	The character used as decimal place marker.
`na.strings`	Character vector of values that will be converted to `NA` if `convert` is `TRUE`.
`convert`	If `TRUE`, will run `type.convert()` to interpret texts as int, dbl, or `NA`.

Scraping HTML tables: example

- We are going to scrape a small table from the Wikipedia page https://en.wikipedia.org/wiki/List_of_human_spaceflights.
- (Note that we're actually using an old version of the page (dating back to May 1, 2018), which is accessible here. Wikipedia pages change, but this old revision and associated link won't.))
- The table is not entirely clean: There are some empty cells, but also images and links.
- The HTML code looks straightforward though.





```
▼<table class="wikitable" style="text-align:right;">
  ▼<tbody>
    ▶<tr>…</tr>
    ▶<tr>…</tr>
    ▼<tr>
      ▶<td>…</td>
       <td>30</td>
       <td>8</td>
       <td></td>
       <td>38</td>
     </tr>
    ▶<tr>…</tr>
    ▶<tr>…</tr>
    ▶<tr>…</tr>
    ▶<tr>…</tr>
    ▶<tr>…</tr>
   </tbody>
</table>
```

Scraping HTML tables: example (cont.)

```
R> library(rvest)
R> url <- "https://en.wikipedia.org/wiki/List_of_human_spaceflights"
R> url_p <- read_html(url)
R> tables <- html_table(url_p, header = TRUE)
R> spaceflights <- tables[[1]]
R> spaceflights
```

```
## # A tibble: 7 × 5
##   ``        `Russia  Soviet Union` `United States` China Total
##   <chr>     <chr>                       <chr>           <int> <chr>
## 1 1961–1970 16                          25                 NA 41
## 2 1971–1980 30                          8                  NA 38
## 3 1981–1990 *25                         *38                NA *63
## 4 1991–2000 20                          63                 NA 83
## 5 2001–2010 24                          34                  3 61
## 6 2011–2020 24                          3                   3 30
## 7 Total     *139                        *171                6 *316
```

Web scraping: good practice

Scraping: the rules of the game

1. You take all the responsibility for your web scraping work.
2. Think about the nature of the data. Does it entail sensitive information? Do not collect personal data without explicit permission.
3. Take all copyrights of a country's jurisdiction into account. If you publish data, do not commit copyright fraud.
4. If possible, stay identifiable. Stay polite. Stay friendly. Obey the scraping etiquette.
5. If in doubt, ask the author/creator/provider of data for permission—if your interest is entirely scientific, chances aren't bad that you get data.

Consult robots.txt

What's robots.txt?

- "Robots exclusion standard", informal protocol to prohibit web robots from crawling content
- Located in the root directory of a website (e.g., google.com/robots.txt)
- Documents which bot is allowed to crawl which resources (and which not)
- Not a technical barrier, but a sign that asks for compliance

What's robots.txt?

- Not an official W3C standard
- Rules listed bot by bot
- General rule listed under `User-agent: *` (most interesting entry for R-based crawlers)
- Directories folders listed separately

Example

```
User-agent: Googlebot
Disallow: /images/
Disallow: /private/
```

Universal ban

```
User-agent: *
Disallow: /
```

Allow declaration

```
User-agent: *
Disallow: /images/
Allow: /images/public/
```

Crawl delay (in seconds)

```
User-agent: *
Crawl-delay: 2
```

Downloading HTML files

Stay modest when accessing lots of data

- Content on the web is publicly available.
- But accessing the data causes server traffic.
- Stay polite by querying resources as sparsely as possible.

Two easy-to-implement practices

1. Do not bombard the server with requests - and if you have to, do so at modest pace.
2. Store web data on your local drive first, then parse.

Looping over a list of URLs

```
R> for (i in 1:length(list_of_urls)) {
+   if (!file.exists(paste0(folder, file_names[i])))
+     download.file(list_of_urls[i],
+                   destfile = paste0(folder, file_n
+                   )
+     Sys.sleep(runif(1, 1, 2))
+     }
+ }
```

- `!file.exists()` checks whether a file does not exist in the specified location.
- `download.file()` downloads the file to a folder. The destination file (location + name) has to be specified.
- `Sys.sleep()` suspends the execution of R code for a given time interval (in seconds).

Staying identifiable

Don't be a phantom

- Downloading massive amounts of data may arouse attention from server administrators.
- Assuming that you've got nothing to hide, you should stay identifiable beyond your IP address.

Two easy-to-implement practices

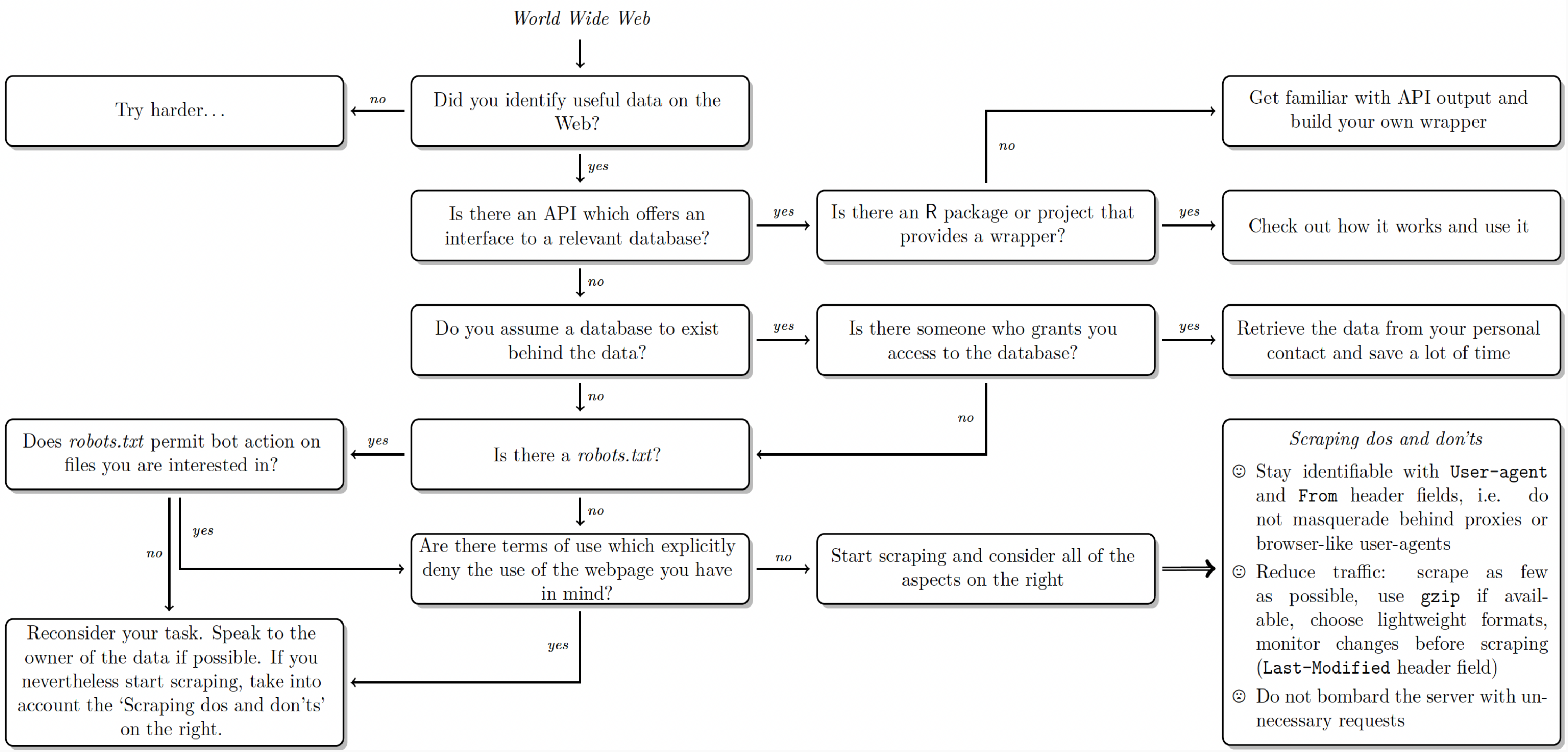
1. Get in touch with website administrators / data owners.
2. Use HTTP header fields `From` and `User-Agent` to provide information about yourself.

Staying identifiable in practice

```
R> url ← "http://a-totally-random-website.com"
R> rvest_session ← session(url,
+   add_headers(From = "my@email.com",
+               `UserAgent` =
+                 R.Version()$version.string
+               )
+ )
R> headlines ← rvest_session %>%
+               html_elements(xpath = "p//a") %>%
+               html_text()
```

- `rvest`'s `session()` creates a session object that responds to HTTP and HTML methods.
- Here, we provide our email address and the current R version as `User-Agent` information.
- This will pop up in the server logs: The webpage administrator has the chance to easily get in touch with you.

Scraping etiquette (cont.)



Summary

Outlook

Until now, the toy examples were limited to single HTML pages. However, often we want to **scrape data from multiple pages**. You might think of newspaper articles, Wikipedia pages, shopping items and the like. In such scenarios, automating the scraping process becomes really powerful. Also, principles of polite scraping are more relevant then.

In other cases, you might be confronted with

- forms,
- authentication,
- dynamic (JavaScript-enriched) content, or want to
- automatically navigate through pages interactively.

Moreover, we've ignored a major alternative way to collect data from the web so far which goes beyond scraping: accessing web APIs. Be sure to check out the respective sessions in the workshop.

There's only so much we can cover in one session. Check out more material online here and there to learn about solutions to some of these problems.

Coming up

Assignment

Assignment 3 is about to go online on GitHub Classroom. Check it out and start scraping the web (politely).

Next lecture

Model fitting and simulation. Now that we know how to retrieve data, let's learn how to run and learn from them.