

Introduction to Data Science

Session 6: Web data and technologies

Simon Munzert

Hertie School | GRAD-C11/E1339

Table of contents

1. Web data for data science
2. HTML basics
3. XPath basics
4. CSS basics
5. Regular expressions
6. Summary

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](#)

4164 Accesses | 110 Altmetric | [Metrics](#)

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 Munzert² 

¹Faculty of Social Sciences, Goethe University Frankfurt am Main, Germany; and ²Data 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}

^aDepartment of Politics, Princeton University, Princeton, NJ 08544; ^bSchool of Public and International Affairs, Princeton University, Princeton, NJ 08544; ^cDepartment of Political Science and International Relations, University of Southern California, Los Angeles, CA 90089; ^dData Science Lab, Hertie School, 10117 Berlin, Germany; and ^eDepartment 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 naturally 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?

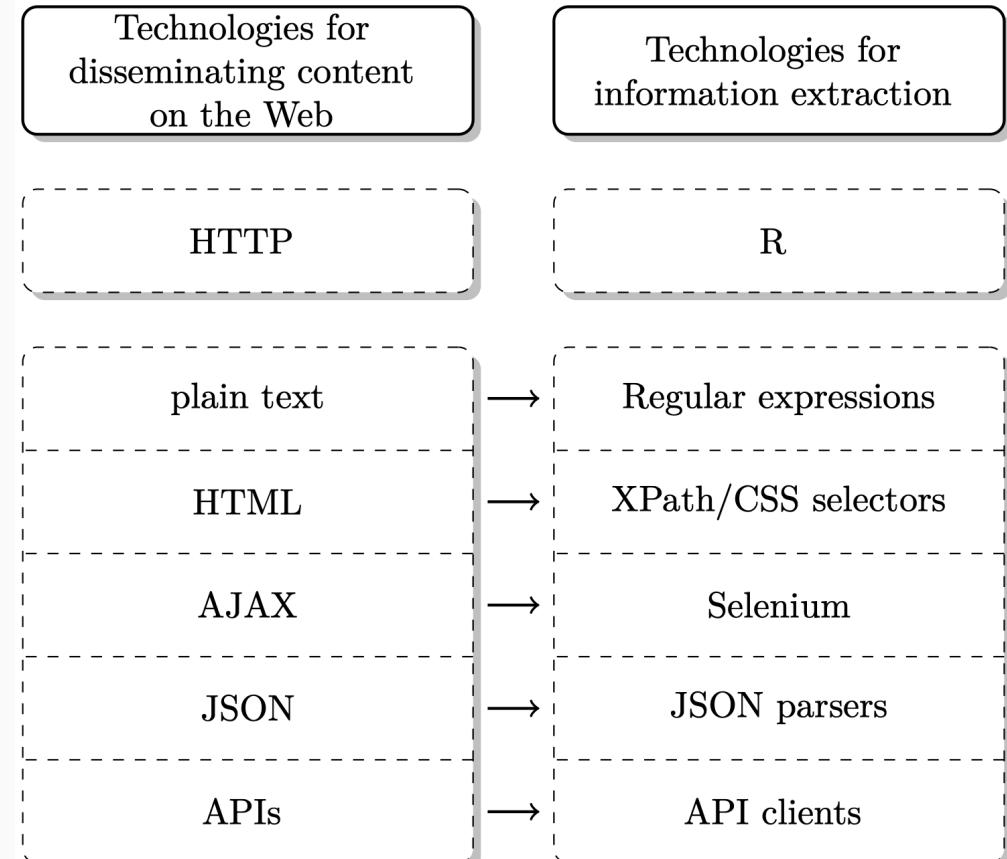
- Not all data you get from the web is "web data".
- Web data is **data that is created on, for, or via the web**. By that definition, a survey dataset that you download from a data repository is not web data.
- On the other hand, survey data collected online (i.e., web/mobile questionnaires) is web 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.

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**)



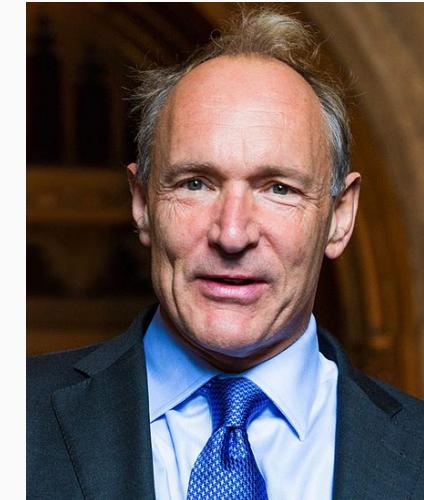
Credit **ADCR**

HTML basics

HTML background

What is HTML?

- **HyperText Markup Language**
- 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 tree structure

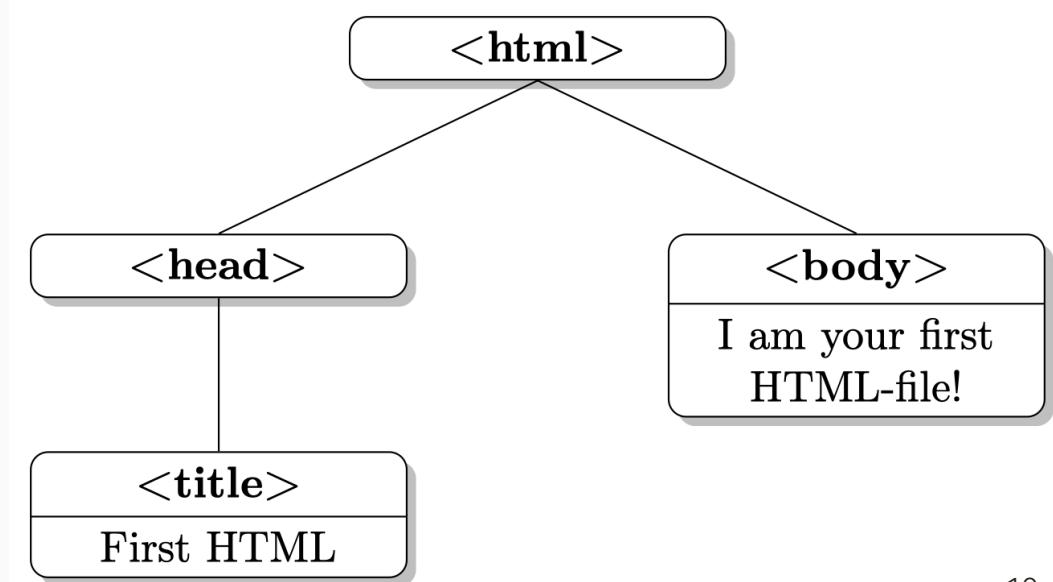
The DOM tree

- 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	<code>title</code>
Start tag	<code><title></code>
End tag	<code></title></code>
Value	<code>First HTML</code>

Attributes

- Describe elements and are stored in the start tag.
- There are specific attributes for specific elements.
- Example: `Link to Homepage`
- 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 ``, ``, and `<dl>`

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

Example:

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

Important tags and attributes

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

- They are used to group content over lines (`<div>`, creating a block-level element) or within lines (``, 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:

```
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:

```
<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:

```
<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.

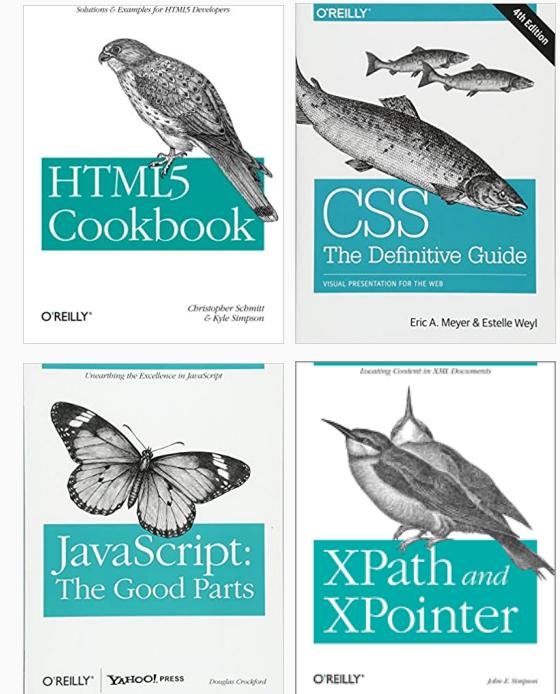
Example:

```
<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".

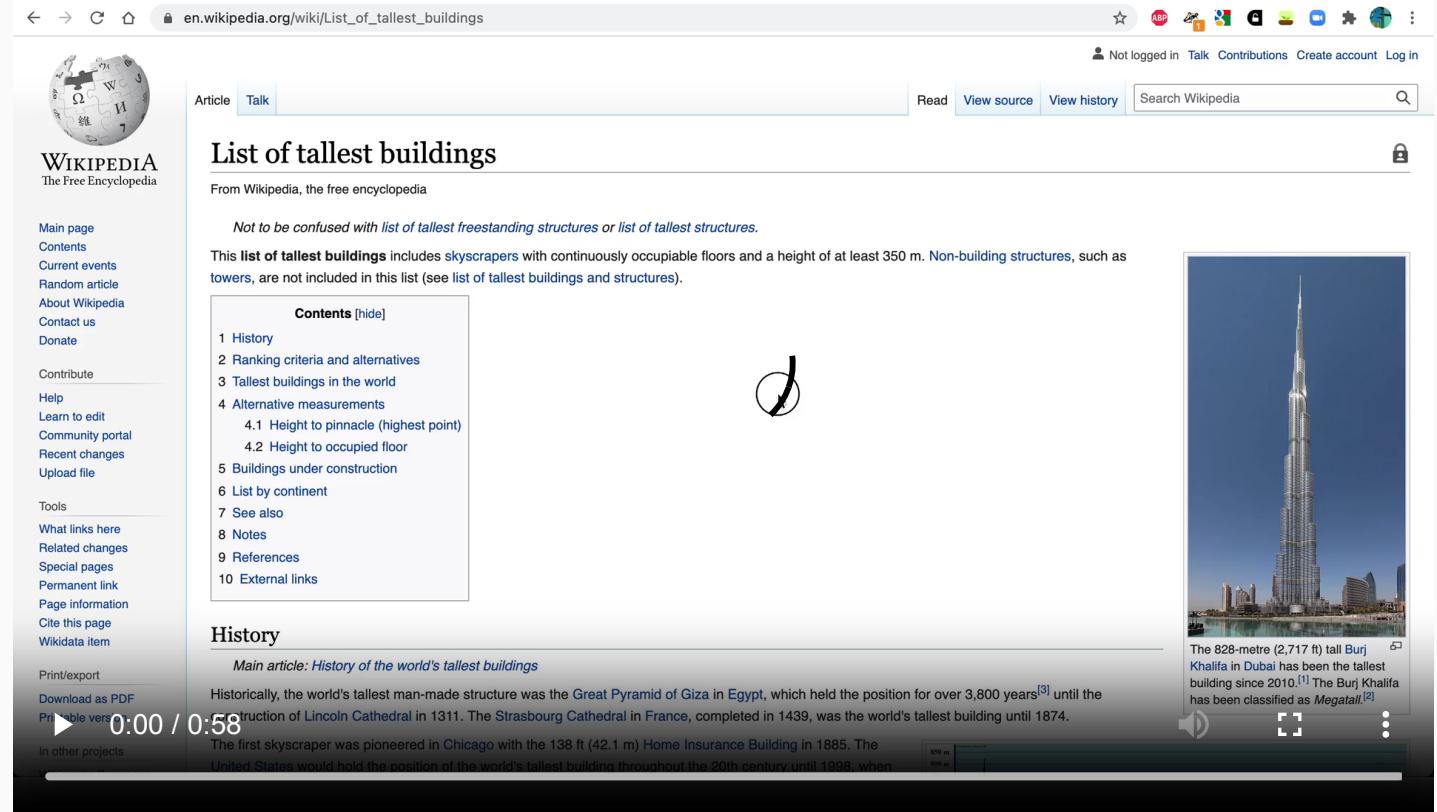
Inspecting HTML source code

- 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

The screenshot shows a web browser displaying the Wikipedia article "List of tallest buildings". The URL in the address bar is en.wikipedia.org/wiki/List_of_tallest_buildings. The page title is "List of tallest buildings" and it is categorized under "Article". The sidebar on the left contains links to "Main page", "Contents", "Current events", "Random article", "About Wikipedia", "Contact us", "Donate", "Contribute", "Help", "Learn to edit", "Community portal", "Recent changes", and "Upload file". Below that is a "Tools" section with links to "What links here", "Related changes", "Special pages", "Permanent link", "Page information", "Cite this page", and "Wikidata item". At the bottom of the sidebar are "Print/export" and "Download as PDF" options. A video player interface at the bottom shows a play button, the time "0:00 / 1:12", and a progress bar. The main content area starts with a note about not being confused with other lists. It then describes the list of tallest buildings, mentioning that it includes skyscrapers with continuously occupiable floors and a height of at least 350 m. Non-building structures, such as towers, are not included. A large image of the Burj Khalifa is prominently displayed on the right side of the page. Below the image, a caption provides details about the building's height (828 metres / 2,717 ft) and its status as the tallest building in Dubai since 2010.

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

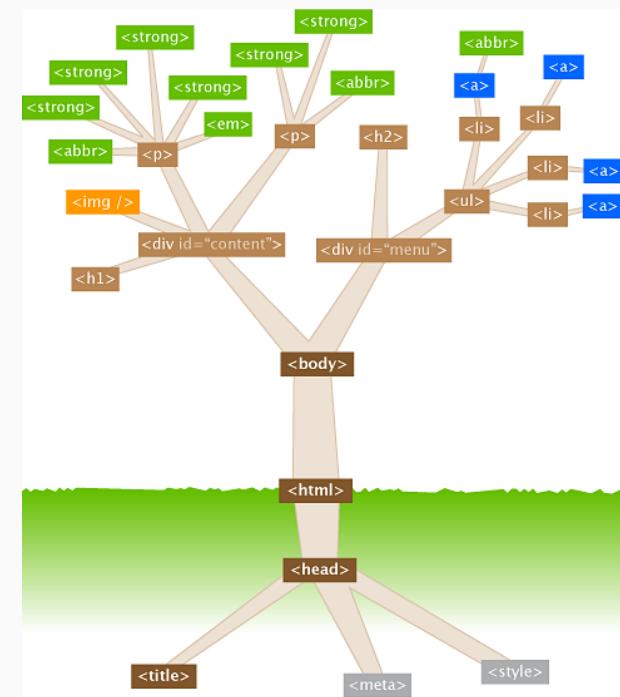
- Check whether data is in static source code (the search function helps!)
- For small HTML files: understand structure

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 itemscope="" itemtype="http://schema.org/WebPage" lang="de">
## [1] <head>\n<meta content="text/html; charset=UTF-8" http-equiv="Content-Type" ...
## [2] <body bgcolor="#fff">\n<script nonce="fe3VLTLA97vq3sXVoznX1Q">(function() ...
```

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.

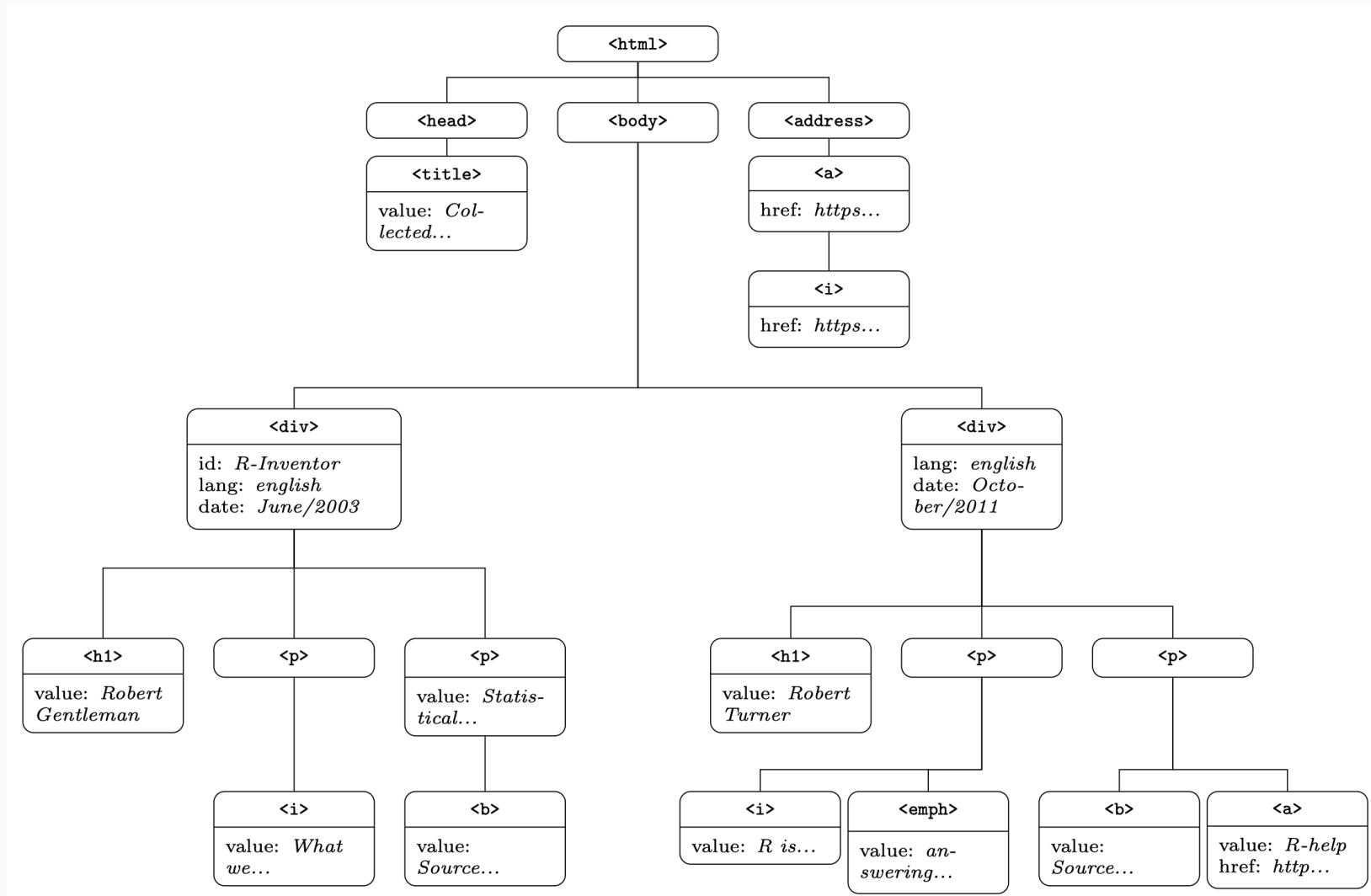
Why XPath for web scraping?

- 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 `` 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

```
<!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 an 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")  
  
## [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")  
  
## [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 preferable. 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/*/*")  
  
## {xml_nodeset (0)}
```

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 ...
```

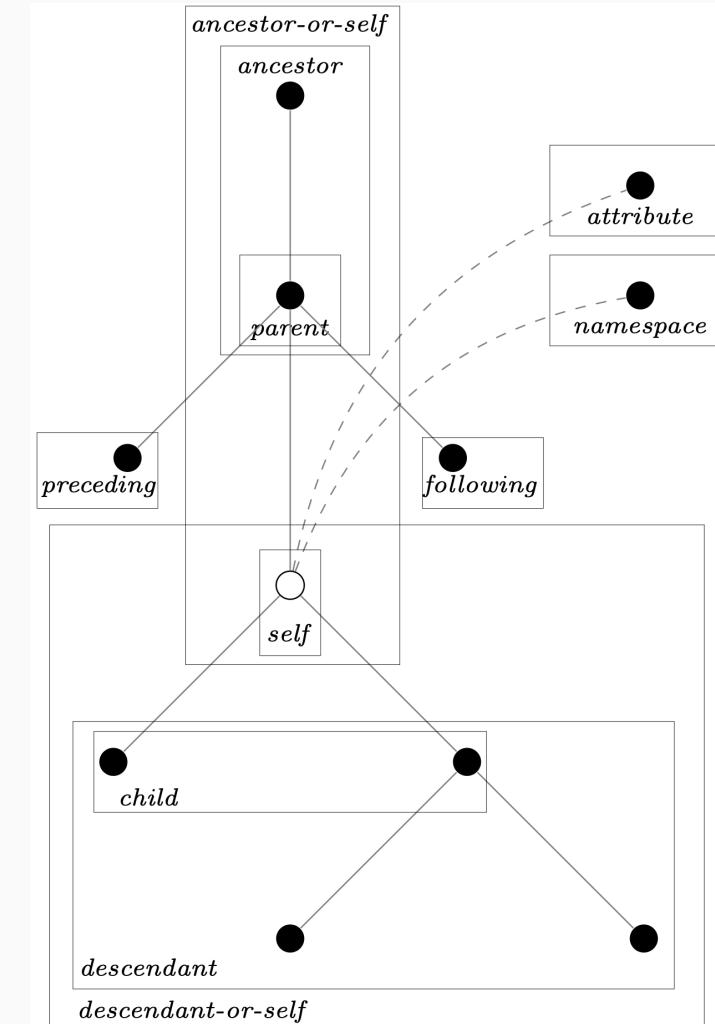
```
R> html_elements(parsed_doc, xpath = "//div[starts-with(./@id, 'R')]")
```

```
## {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]")

## [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

- **Cascading Style Sheets** (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

- Cascading **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`.

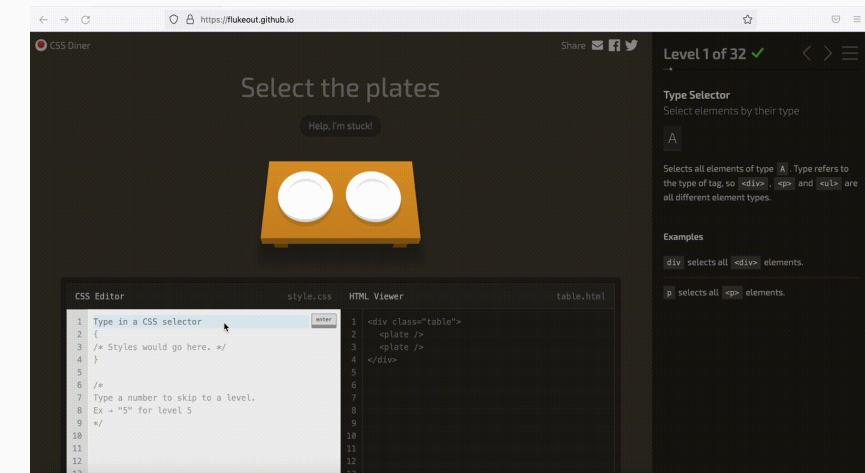
CSS selectors

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.



Regular expressions

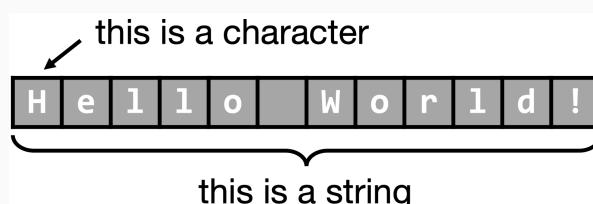
What are regular expressions?

Definition

Regular expressions a.k.a. *regex* or *RegExp* is a tool - a little language of its own really - that lets you describe patterns in text/strings.

Funnily, a regular expression itself is a sequence of characters, some with special, some with literal meaning.

Regular expressions are widely applicable and implemented in many programming languages, including R, as well as search engines, search and replace dialogs, etc.



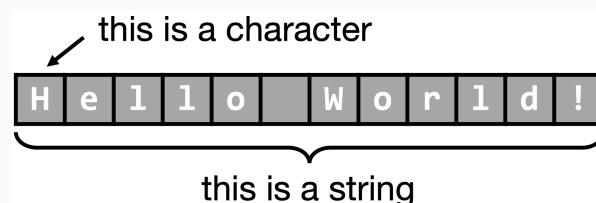
What are regular expressions?

Definition

Regular expressions a.k.a. *regex* or *RegExp* is a tool - a little language of its own really - that lets you describe patterns in text/strings.

Funnily, a regular expression itself is a sequence of characters, some with special, some with literal meaning.

Regular expressions are widely applicable and implemented in many programming languages, including R, as well as search engines, search and replace dialogs, etc.



Why is this useful for web scraping?

Information on the web can often be described by patterns (think email addresses, numbers, cells in HTML tables, ...).

If the data of interest follow specific patterns, we can match and extract them - regardless of page layout and HTML overhead.

Whenever the information of interest is (stored in) text, regular expressions are useful for extraction and tidying purposes.

Regular expressions: example

Below you see a string that contains unstructured phone book entries. The goal is to clean it up and extract the entries. The problem is that the text is really messy, and to find a pattern that helps us describe names on the one hand and phone numbers on the other is difficult. But: regular expressions FTW!

```
R> phone_vec <-  
+ "555-1239Moe Szyslak(636) 555-0113Burns, C. Montgomery  
+ 555-6542Rev. Timothy Lovejoy555 8904Ned Flanders636-555-3226  
+ Simpson,Homer5553642Dr. Julius Hibbert"
```

Regular expressions: example

Below you see a string that contains unstructured phone book entries. The goal is to clean it up and extract the entries. The problem is that the text is really messy, and to find a pattern that helps us describe names on the one hand and phone numbers on the other is difficult. But: regular expressions FTW!

```
R> phone_vec <-  
+ "555-1239Moe Szyslak(636) 555-0113Burns, C. Montgomery  
+ 555-6542Rev. Timothy Lovejoy555 8904Ned Flanders636-555-3226  
+ Simpson,Homer5553642Dr. Julius Hibbert"
```

We're loading the `stringr` package that provides us with tidyverse functionality to operate with string data and apply regular expressions. Then, we construct a regular expression each for the names and the phone numbers (this is the tricky part!). Finally, we apply the regular expressions on the raw vector to extract the information of interest.

Regular expressions: example

```
R> phone_vec <-  
+ "555-1239Moe Szyslak(636) 555-0113Burns, C. Montgomery  
+ 555-6542Rev. Timothy Lovejoy555 8904Ned Flanders636-555-3226  
+ Simpson,Homer5553642Dr. Julius Hibbert"
```

Regular expressions: example

```
R> phone_vec ←  
+ "555-1239Moe Szyslak(636) 555-0113Burns, C. Montgomery  
+ 555-6542Rev. Timothy Lovejoy555 8904Ned Flanders636-555-3226  
+ Simpson,Homer5553642Dr. Julius Hibbert"  
  
R> library(stringr)  
R> names_vec ← unlist(str_extract_all(phone_vec, "[[:alpha:].[, ]{2,}"))  
R> names_vec  
  
## [1] "Moe Szyslak"           "Burns, C. Montgomery" "Rev. Timothy Lovejoy"  
## [4] "Ned Flanders"          "Simpson,Homer"        "Dr. Julius Hibbert"
```

Regular expressions: example

```
R> phone_vec ←  
+ "555-1239Moe Szyslak(636) 555-0113Burns, C. Montgomery  
+ 555-6542Rev. Timothy Lovejoy555 8904Ned Flanders636-555-3226  
+ Simpson,Homer5553642Dr. Julius Hibbert"  
  
R> library(stringr)  
R> names_vec ← unlist(str_extract_all(phone_vec, "[[:alpha:].[ ]{2,}"))  
R> names_vec  
  
## [1] "Moe Szyslak"           "Burns, C. Montgomery" "Rev. Timothy Lovejoy"  
## [4] "Ned Flanders"          "Simpson,Homer"        "Dr. Julius Hibbert"  
  
R> numbers_vec ← unlist(str_extract_all(phone_vec,  
+                           "\\\\"(?([:digit:]{3})?\\)?(-| )?[:digit:]{3}(-| )?[:digit:]{4}"))  
R> numbers_vec  
  
## [1] "555-1239"      "(636) 555-0113" "555-6542"      "555 8904"  
## [5] "636-555-3226" "5553642"
```

Regular expressions: example

```
R> phone_vec ←  
+ "555-1239Moe Szyslak(636) 555-0113Burns, C. Montgomery  
+ 555-6542Rev. Timothy Lovejoy555 8904Ned Flanders636-555-3226  
+ Simpson,Homer5553642Dr. Julius Hibbert"  
  
R> library(stringr)  
R> names_vec ← unlist(str_extract_all(phone_vec, "[[:alpha:].[ ]{2,}"))  
R> names_vec  
  
## [1] "Moe Szyslak"           "Burns, C. Montgomery" "Rev. Timothy Lovejoy"  
## [4] "Ned Flanders"          "Simpson,Homer"        "Dr. Julius Hibbert"  
  
R> numbers_vec ← unlist(str_extract_all(phone_vec,  
+                           "\\\\"(?([[:digit:]]{3})?\\)?(-| )?[:digit:]]{3}(-| )?[:digit:]]{4}"))  
R> numbers_vec  
  
## [1] "555-1239"      "(636) 555-0113"  "555-6542"      "555 8904"  
## [5] "636-555-3226" "5553642"
```

Wait, wait?! 😱

🎵 Regex superherooo 🎵



Regular expressions in R

Here's an example string we're going to work with:

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

Regular expressions in R

Here's an example string we're going to work with:

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

We are going to use the `str_extract()` and the `str_extract_all()` functions from the `stringr` package to apply regular expressions to strings. The generic syntax is:

- `str_extract(string, pattern)`
- `str_extract_all(string, pattern)`

Here's the difference: `str_extract()` returns the first match, `str_extract_all()` returns all matches.

Basic regex syntax

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

Strings match themselves

```
R> str_extract(example.obj, "small")
```

```
## [1] "small"
```

```
R> str_extract(example.obj, "banana")
```

```
## [1] NA
```

Basic regex syntax

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

Strings match themselves

```
R> str_extract(example.obj, "small")  
## [1] "small"  
  
R> str_extract(example.obj, "banana")  
## [1] NA
```

Multiple matches are returned as a list

```
R> multi_vec <- c("text", "manipulation", "basics")  
R> str_extract_all(multi_vec, "a")  
## [[1]]  
## character(0)  
##  
## [[2]]  
## [1] "a" "a"  
##  
## [[3]]  
## [1] "a"
```

Basic regex syntax *cont.*

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

Character matching is case sensitive

```
R> str_extract(example.obj, "small")
```

```
## [1] "small"
```

```
R> str_extract(example.obj, "SMALL")
```

```
## [1] NA
```

```
R> str_extract(example.obj,  
+               regex("SMALL", ignore_case = TRUE))
```

```
## [1] "small"
```

Basic regex syntax cont.

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

Character matching is case sensitive

```
R> str_extract(example.obj, "small")
```

```
## [1] "small"
```

```
R> str_extract(example.obj, "SMALL")
```

```
## [1] NA
```

```
R> str_extract(example.obj,  
+               regex("SMALL", ignore_case = TRUE))  
  
## [1] "small"
```

We can match arbitrary combinations of characters

```
R> str_extract(example.obj, "mall sent")
```

```
## [1] "mall sent"
```

Basic regex syntax *cont.*

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

Matching the beginning of a string

```
R> str_extract(example.obj, "^1")
```

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

```
R> str_extract(example.obj, "^2")
```

```
## [1] NA
```

Basic regex syntax *cont.*

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

Matching the beginning of a string

```
R> str_extract(example.obj, "^1")
```

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

```
R> str_extract(example.obj, "^2")
```

```
## [1] NA
```

Matching the end of a string

```
R> str_extract(example.obj, "sentence$")
```

```
## [1] NA
```

```
R> str_extract(example.obj, "sentence.\$")
```

```
## [1] "sentence."
```

Basic regex syntax *cont.*

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

Express an "or" with the pipe operator

```
R> unlist(str_extract_all(example.obj, "tiny|sentence"))  
## [1] "sentence" "tiny"      "sentence"
```

Basic regex syntax *cont.*

```
R> example.obj <- "1. A small sentence. - 2. Another tiny sentence."
```

Express an "or" with the pipe operator

```
R> unlist(str_extract_all(example.obj, "tiny|sentence"))  
## [1] "sentence" "tiny"      "sentence"
```

The dot: the ultimate wildcard

```
R> str_extract(example.obj, "sm.ll")  
## [1] "small"
```

Meta-characters

Matching of meta-characters

- Some symbols have a special meaning in the regex syntax: ., |, (,), [,], {, }, ^, \$, *, +, ?, and -.
- If we want to match them literally, we have to use an escape sequence: \symbol
- As \ is a meta character itself, we have to escape it with \, so we always write \\. 😱
- Alternatively, use fixed("symbols") to let the parser interpret a chain of symbols literally.

```
R> unlist(str_extract_all(example.obj, "\\\n."))

## [1] "."

R> unlist(str_extract_all(example.obj, fixed(".")))

## [1] "."
```

Character classes

Square brackets [] define character classes

- Character classes help define special wild cards.
- The idea is that any of the characters within the brackets can be matched.

```
R> str_extract(example.obj, "sm[abc]ll")  
## [1] "small"
```

- The hyphen defines a range of characters.

```
R> str_extract(example.obj, "sm[a-p]ll")  
## [1] "small"
```

Character classes *cont.*

Some character classes are pre-defined. They are very convenient to efficiently describe specific string patterns.

Specification	Meaning
<code>[:digit:]</code>	Digits: 0 1 2 3 4 5 6 7 8 9
<code>[:lower:]</code>	Lower-case characters: a-z
<code>[:upper:]</code>	Upper-case characters: A-Z
<code>[:alpha:]</code>	Alphabetic characters: a-z and A-Z
<code>[:alnum:]</code>	Digits and alphabetic characters
<code>[:punct:]</code>	Punctuation characters: . , , ; , etc.
<code>[:graph:]</code>	Graphical characters: <code>[:alnum:]</code> and <code>[:punct:]</code>
<code>[:blank:]</code>	Blank characters: Space and tab
<code>[:space:]</code>	Space characters: Space, tab, newline, and others
<code>[:print:]</code>	Printable characters: <code>[:alnum:]</code> , <code>[:punct:]</code> and <code>[:space:]</code>

Character classes in action

Pre-defined character classes are useful because they are efficient and let us

- combine different kinds of characters
 - facilitate reading of an expression
 - include special characters, e.g., ß, ö, æ, ...
 - can be extended

```
R> unlist(str_extract_all(example.obj, "[[:punct:]ABC]"))
```

```
## [1] ". " "A" ". " "- " ". " "A" ". "
```

```
R> unlist(str_extract_all(example.obj, "[^[:alnum:]]"))
```

Meta symbols in character classes

Within a character class, most meta-characters lose their special meaning. There are exceptions though:

- `^` becomes "not": `[^abc]` matches any character other than "a", "b", or "c".
- `-` becomes a range specifier: `[a-d]` matches any character from a to d. However, `-` at the beginning or the end of a character class matches the hyphen.

```
R> unlist(str_extract_all(example.obj, "[1-2]"))
```

```
## [1] "1" "2"
```

```
R> unlist(str_extract_all(example.obj, "[12-]"))
```

```
## [1] "1" "-" "2"
```

Quantifiers

Quantifiers are meta-characters that allow you to specify how often a certain string pattern should be allowed to appear.

Quantifier	Meaning
?	The preceding item is optional and will be matched at most once
*	The preceding item will be matched zero or more times
+	The preceding item will be matched one or more times
{n}	The preceding item is matched exactly n times
{n, }	The preceding item is matched n or more times
{n, m}	The preceding item is matched between n and m times

```
R> str_extract(example.obj, "s[:alpha:]{3}l")
```

```
## [1] "small"
```

```
R> str_extract(example.obj, "A.+sentence")
```

```
## [1] "A small sentence. - 2. Another tiny sentence"
```

Greedy quantification

The use of `.+` results in "greedy" matching, i.e. the parser tries to match as many characters as possible. This is not always desired. However, the meta-character `?` helps avoid greedy quantification. More generally, it re-interprets the quantifiers `*`, `+`, `?` or `{m,n}` to match as few times as possible.

```
R> str_extract(example.obj, "A.+sentence")  
## [1] "A small sentence. - 2. Another tiny sentence"  
  
R> str_extract(example.obj, "A.+?sentence")  
## [1] "A small sentence"
```

Backreferencing

Sometimes it's useful to induce some "memory" into regex, as in: "Find something that matches a certain pattern, and then again a repeated match of previously matched pattern."

The first pattern is defined with round brackets, as in `(pattern)`. We then refer to the it using `\1` (or with `\2` for the second pattern, etc.).

Example: Match the first letter, then anything until you find the first letter again (not greedy).

```
R> str_extract(example.obj, "([[:alpha:]])\\.+?\\1")  
## [1] "A small sentence. - 2. A"
```

Backreferencing cont.

Goal: Match a word that does not include "a" until the word appears the second time.

Solution:

```
R> str_extract(example.obj, "([[:punct:]][b-z]+[[:punct:]])\\.+?\\\\1")  
## [1] " sentence. - 2. Another tiny sentence."
```

How it works:

- Match all letters without a, therefore: [b-z]
- Match complete words with beginning/end: [[:punct:]]
- Define first word pattern* (...)
- Match anything between occurrences of both words: .+?
- Refer to original word \\1

Coming up

Assignment

No assignment due, but web technologies will be featured in the next assignment.

Next lecture

We'll get serious about scraping data from the web and tapping APIs.