Web Scraping

Lecture 13

Dr. Colin Rundel

rvest

www.rstudio.com

Hypertext Markup Language

Most of the data on the web is still largely available as HTML - while it is structured (hierarchical) it often is not available in a form useful for analysis (flat / tidy).

```html
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p align="center">Hello world!</p>
    <br/>
    <div class="name" id="first">John</div>
    <div class="name" id="last">Doe</div>
    <div class="contact">
      <div class="home">555-555-1234</div>
      <div class="home">555-555-2345</div>
      <div class="work">555-555-9999</div>
      <div class="fax">555-555-8888</div>
    </div>
  </body>
</html>
```

rvest

`rvest` is a package from the tidyverse that makes basic processing and manipulation of HTML data straightforward. It provides high level functions for interacting with html via the xml2 library.

Core functions:

- `read_html()` - read HTML data from a url or character string.
- `html_elements()` / ~~`html_nodes()`~~ - select specified elements from the HTML document using CSS selectors (or xpath).
- `html_element()` / ~~`html_node()`~~ - select a single element from the HTML document using CSS selectors (or xpath).
- `html_table()` - parse an HTML table into a data frame.
- `html_text()` / `html_text2()` - extract tag's text content.
- `html_name()` - extract a tag/element's name(s).
- `html_attrs()` - extract all attributes.
- `html_attr()` - extract attribute value(s) by name.

html, rvest, & xml2

```
html =
'<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p align="center">Hello world!</p>
    <br/>
    <div class="name" id="first">John</div>
    <div class="name" id="last">Doe</div>
    <div class="contact">
      <div class="home">555-555-1234</div>
      <div class="home">555-555-2345</div>
      <div class="work">555-555-9999</div>
      <div class="fax">555-555-8888</div>
    </div>
  </body>
</html>'
```

```
read_html(html)
```

```
{html_document}
<html>
[1] <head>\n<meta http-equiv="Content-Type" cont
[2] <body>\n    <p align="center">Hello world!</
```

Selecting elements

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

```
{xml_nodeset (1)}
[1] <p align="center">Hello world!</p>
```

```r
read_html(html) |> html_elements("p") |> html_text()
```

```
[1] "Hello world!"
```

```r
read_html(html) |> html_elements("p") |> html_name()
```

```
[1] "p"
```

```r
read_html(html) |> html_elements("p") |> html_attrs()
```

```
[[1]]
   align
"center"
```

```r
read_html(html) |> html_elements("p") |> html_attr("align")
```

```
[1] "center"
```

More selecting tags

```r
read_html(html) |> html_elements("div")
```

```
{xml_nodeset (7)}
[1] <div class="name" id="first">John</div>
[2] <div class="name" id="last">Doe</div>
[3] <div class="contact">\n      <div class="home">555-555-1234</div>\n      ...
[4] <div class="home">555-555-1234</div>
[5] <div class="home">555-555-2345</div>
[6] <div class="work">555-555-9999</div>
[7] <div class="fax">555-555-8888</div>
```

```r
read_html(html) |> html_elements("div") |> html_text()
```

```
[1] "John"
[2] "Doe"
[3] "\n      555-555-1234\n      555-555-2345\n      555-555-9999\n      555-555-8888\
[4] "555-555-1234"
[5] "555-555-2345"
[6] "555-555-9999"
[7] "555-555-8888"
```

Nesting tags

```r
read_html(html) |> html_elements("body div")
```

```
{xml_nodeset (7)}
[1] <div class="name" id="first">John</div>
[2] <div class="name" id="last">Doe</div>
[3] <div class="contact">\n      <div class="home">555-555-1234</div>\n       ...
[4] <div class="home">555-555-1234</div>
[5] <div class="home">555-555-2345</div>
[6] <div class="work">555-555-9999</div>
[7] <div class="fax">555-555-8888</div>
```

```r
read_html(html) |> html_elements("body>div")
```

```
{xml_nodeset (3)}
[1] <div class="name" id="first">John</div>
[2] <div class="name" id="last">Doe</div>
[3] <div class="contact">\n      <div class="home">555-555-1234</div>\n       ...
```

```r
read_html(html) |> html_elements("body div div")
```

```
{xml_nodeset (4)}
[1] <div class="home">555-555-1234</div>
[2] <div class="home">555-555-2345</div>
[3] <div class="work">555-555-9999</div>
[4] <div class="fax">555-555-8888</div>
```

CSS selectors

We will be using a tool called selector gadget to help us identify the html elements of interest - it does this by constructing a CSS selector which can be used to subset the html document.

Some examples of basic selector syntax is below,

Selector	Example	Description
.class	`.title`	Select all elements with class="title"
#id	`#name`	Select all elements with id="name"
element	`p`	Select all <p> elements
element element	`div p`	Select all <p> elements inside a <div> element
element>element	`div > p`	Select all <p> elements with <div> as a parent
[attribute]	`[class]`	Select all elements with a class attribute
[attribute=value]	`[class=title]`	Select all elements with class="title"

There are also a number of additional combinators and pseudo-classes that improve flexibility, see examples here.

`html_elements()` also supports XPath selectors as an alternative to CSS selectors, though we will prefer CSS selectors in

CSS classes and ids

```r
read_html(html) |> html_elements(".name")
```

```
{xml_nodeset (2)}
[1] <div class="name" id="first">John</div>
[2] <div class="name" id="last">Doe</div>
```

```r
read_html(html) |> html_elements("div.name")
```

```
{xml_nodeset (2)}
[1] <div class="name" id="first">John</div>
[2] <div class="name" id="last">Doe</div>
```

```r
read_html(html) |> html_elements("#first")
```

```
{xml_nodeset (1)}
[1] <div class="name" id="first">John</div>
```

Mixing it up

```r
read_html(html) |> html_elements("[align]")
```

```
{xml_nodeset (1)}
[1] <p align="center">Hello world!</p>
```

```r
read_html(html) |> html_elements(".contact div")
```

```
{xml_nodeset (4)}
[1] <div class="home">555-555-1234</div>
[2] <div class="home">555-555-2345</div>
[3] <div class="work">555-555-9999</div>
[4] <div class="fax">555-555-8888</div>
```

`html_text()` vs `html_text2()`

```r
html = read_html(
  "<p>
    This is the first sentence in the paragraph.
    This is the second sentence that should be on the same line as the first sente
  </p>"
)
```

```r
html |> html_text()
```

```
[1] "  \n    This is the first sentence in the paragraph.\n    This is the second sent
```

```r
html |> html_text2()
```

```
[1] "This is the first sentence in the paragraph. This is the second sentence that sho
```

```
html |> html_text() |> cat(sep="\n")
```

```
    This is the first sentence in the paragraph.
    This is the second sentence that should be on the same line a
```

```
html |> html_text2() |> cat(sep="\n")
```

```
This is the first sentence in the paragraph. This is the second s
This third sentence should start on a new line.
```

html tables

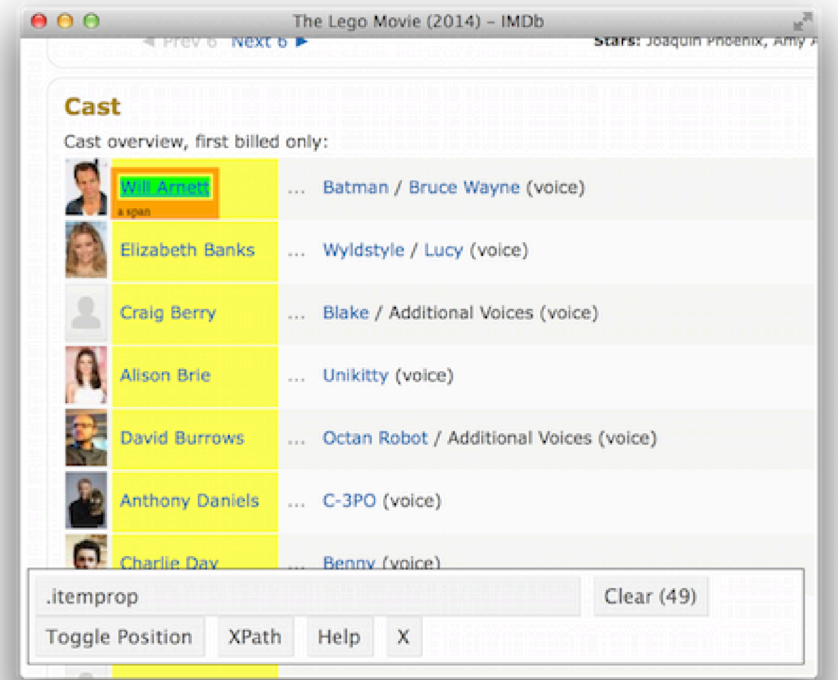
```r
html_table =
'<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <table>
      <tr> <th>a</th> <th>b</th> <th>c</th> </tr>
      <tr> <td>1</td> <td>2</td> <td>3</td> </tr>
      <tr> <td>2</td> <td>3</td> <td>4</td> </tr>
      <tr> <td>3</td> <td>4</td> <td>5</td> </tr>
    </table>
  </body>
</html>'
```

```r
read_html(html_table) |>
  html_elements("table") |>
  html_table()
```

```
[[1]]
# A tibble: 3 × 3
      a     b     c
  <int> <int> <int>
1     1     2     3
2     2     3     4
3     3     4     5
```

SelectorGadget

This is a javascript based tool that helps you interactively build an appropriate CSS selector for the content you are interested in.



selectorgadget.com

Web scraping considerations

“Can you?” vs “Should you?”

Researchers just released profile data on 70,000 OkCupid users without permission

By Brian Resnick | @B_resnick | brian@vox.com | May 12, 2016, 6:00pm EDT

A group of researchers has released a data set on nearly 70,000 users of the online dating site OkCupid. The data dump breaks the cardinal rule of social science research ethics: It took identifiable personal data without permission.

The information — while publicly available to OkCupid users — was collected by Danish researchers who never contacted OkCupid or its clientele about using it.

The data, collected from November 2014 to March 2015, includes user names, ages, gender, religion, and personality traits, as well as answers to the personal questions the site asks to help match potential mates. The users hail from a few dozen countries around the world.

The data dump did not reveal anyone's real name. But it's entirely possible to use clues from a user's location, demographics, and OkCupid user name to determine their identity.

> If your OkC username is one you've used anywhere else, I now know your sexual preferences & kinks, your answers to thousands of questions.
>
> — Scott B. Weingart (@scott_bot) May 11, 2016

“Can you?” vs “Should you?”

Scraping permission & `robots.txt`

There is a standard for communicating to users if it is acceptable to automatically scrape a website via the robots exclusion standard or `robots.txt`.

You can find examples at all of your favorite websites: google, facebook, etc.

These files are meant to be machine readable, but the `polite` package can handle this for us (and much more).

```r
polite::bow("http://google.com")
```

```
<polite session> http://google.com
    User-agent: polite R package
    robots.txt: 473 rules are defined for 5 bots
   Crawl delay: 5 sec
  The path is scrapable for this user-agent
```

```r
polite::bow("http://facebook.com")
```

```
<polite session> http://facebook.com
    User-agent: polite R package
    robots.txt: 720 rules are defined for 34 bots
   Crawl delay: 5 sec
  The path is not scrapable for this user-agent
```

Scraping with polite

Beyond the `bow()` function, `polite` also has a `scrape()` function that helps you scrape a website while maintaining the three pillars of a polite session:

- seek permission,
- take it slowly
- never ask twice.

This is achieved by using the session object from `bow()` within the `scrape()` function to make the request (this is equivalent to `rvest`'s `read_html()` and returns a parsed html object).

New paths within the website can be accessed by using the `nod()` function before using `scrape()`.

Rate limiting

Making requests too quickly can overload a server, get your IP blocked, or violate a site's terms of service. Adding delays between requests is essential for responsible scraping.

The simplest approach is `Sys.sleep()` between requests:

```
for (url in urls) {
  read_html(url) |> html_elements(".title")
  Sys.sleep(1)  # wait 1 second between requests
}
```

`polite` handles this automatically — `scrape()` respects the crawl delay specified in `robots.txt` and defaults to a 5-second delay if none is specified:

```
session = polite::bow("https://example.com")

polite::nod(session, "page/1") |> polite::scrape()
polite::nod(session, "page/2") |> polite::scrape()
```

JavaScript rendered pages

`read_html()` only sees the static HTML delivered by the server — pages that use JavaScript to render content will appear empty or incomplete.

`rvest::read_html_live()` handles this by running a headless Chrome browser (via the `chromote` package) that fully renders the page before returning the HTML:

```r
page = read_html_live("https://example.com")

page |> html_elements(".some-js-rendered-class")
```

The returned object also supports interacting with the page like a user would:

```r
page$click("#load-more-button")
page$scroll_to(top = 1000)
page$type("#search-box", "query text")
page$view()
page$session$screenshot("screenshot.png")
```

Graceful error handling

When scraping many pages, individual requests will often fail (missing elements, network errors, changed page structure, etc.). Rather than stopping entirely, we can handle errors gracefully.

`purrr::possibly()` wraps a function so that errors return a default value instead of stopping:

```r
safe_read = purrr::possibly(read_html, otherwise = NULL)

urls |>
  purrr::map(safe_read)
```

`purrr::safely()` is similar but returns a list with `result` and `error` components, letting you inspect what went wrong:

```r
safe_read = purrr::safely(read_html)

results = urls |> purrr::map(safe_read)

results |> purrr::map("result")  # successful results (NULL on failure)
results |> purrr::map("error")   # error messages (NULL on success)
```

Example - Rotten Tomatoes

For the movies listed in the **Popular Streaming Movies** list on `rottentomatoes.com` create a data frame with the movies' titles, their tomatometer score, and whether the movie is fresh or rotten, and the movie's url.

Exercise 1

Using the url for each movie, now go out and grab the mpaa rating, the runtime and number of user ratings.

If you finish that you can then try to scrape the Tomatometer and audience scores for each movie.