More on web scraping

Web scraping workflow

In the reading set, you worked in an Rmd file to scrape a table from a Wikipedia page. The Rmd file allowed you to knit your work and upload it to Gradescope, but it is more typical to create a separate R script for web scraping and data wrangling. Within that R script, you would save your final scraped and wrangled dataset into a user-friendly format (e.g., txt, csv, etc.) and start your Rmd document by reading in that cleaned dataset.

Workflow steps

1. Create a folder to begin your project.
2. If you expect to have multiple R scripts, create a subfolder called "code" and place all of your R scripts within that folder (e.g., the "code" folder might include "scrape-mile-records.R", "wrangle-mile-records.R", "scratch-work.R").
3. Similarly, if you expect to have multiple datasets, create a subfolder called "data" in which to place all of your datasets (e.g., the "data" folder might include "mile-records_messy.csv", "mile-records.csv", "mile-records_wide.csv").

   If you only expect to have one R script and/or one dataset, then creating subfolders is unnecessary.
4. Scrape and wrangle your data in the appropriate R script, and then output and save the scraped data in your data folder in a user-friendly format (e.g., as a csv or txt file by using `write_csv()` or as a dataframe in an RData file by using `save()`).
5. Import the csv or R dataframe (e.g., using `read_csv()` or `load()`) into an Rmd (e.g. "analyze-running-records.Rmd") file to analyze or otherwise work with the data.

Workflow logic

Scraping your data in an R script instead of your R Markdown document prevents you from repeating the web-scraping process every time you knit your R Markdown document. This is important for a number of reasons, including:

- *Hit limits*: Some websites have limits on the number of hits you can make. It's usually fine to make multiple hits as you're working and testing your scraping code, but once you have the data you need from a website, you don't want to re-execute the scraping every time you knit a file or you could get blocked from scraping that site.
- *Efficiency*: Depending on what you're scraping, it could take quite some time and computer resources to execute the scraping code.
- *Reproducibility*: You want a permanent record of the information you scraped from the web. Websites can change over time, so saving the data permanently and writing code to analyze that saved dataset is safer than relying on re-scraping every time you need to update your analysis code.

Web scraping tables

When the information you want is stored in a table format on a web page, scraping the data is fairly straightforward thanks to the **rvest** package and the `pluck()` function from **purrr**, e.g., usually following the format:

```
# 1. Identify webpage URL you want to scrape
url <- "INSERT URL HERE"

# 2. Confirm bots are allowed to access the page
paths_allowed(url)

# 3. Scrape tables on the page
tables <- url %>%
  read_html() %>%
  html_elements("table")

# 4. Identify and "pluck" tables you want to work with
table1 <- pluck(tables, 1) %>%
  html_table()
 .
 .
 .
table9 <- pluck(tables, 9) %>%
  html_table()
```

> **Note on checking `paths_allowed()`:** Just because bots are allowed on the root domain of a website, the website could disallow bots on certain subpages (e.g., "en.wikisource.org" is the root domain but a subpage is "https://en.wikisource.org/wiki/September%27s_Baccalaureate"). Make sure you verify the particular subpage or set of subpages you want to scrape allow bots by using `paths_allowed()`. Alternatively, you could check the full "robots.txt" file yourself to see where bots are not allowed by adding "/robots.txt" to the end of any root domain (e.g., "https://www.en.wikisource.org/robots.txt").

Web scraping text

We often want information from a website that is *not* stored in table format. For instance, next week we'll be working with poetry to introduce text analysis, and we'll get those poems from webpages where the text is not stored in tables.

An example of scraping text, rather than tables, from a page is shown below. The code scrapes the text of Emily Dickinson's poem, *September's Baccalaureate* from Wikisource.

This requires two modifications to our previous approach:

1. Identify a different element (instead of `"table"`) within the `html_elements()` function to list, and
2. Use `html_text()` (instead of `html_table()`) to extract the text.

```
# Identify page where poem is listed
sep_bac_url <- "https://en.wikisource.org/wiki/September%27s_Baccalaureate"

# Confirm bots are allowed to access the page
paths_allowed(sep_bac_url)
```

```
[1] TRUE
```

```
# Get poem
sep_bac_text <- sep_bac_url %>%
  read_html() %>%
  # Get list of "div p" elements on the page
  html_elements("div p") %>%
  # `Pluck` poem from list and grab text elements
  pluck(1) %>%
  html_text()
```

If we look at the output of `sep_bac_text`, we'll see that the output is just one very long string (the "`\n`"s are R's way of inserting line breaks in text output).

```
# Print text of poem
print(sep_bac_text)
```

```
[1] "September's Baccalaureate\nA combination is\nOf Crickets - Crows - and Retrospects\nAnd a dis
```

To clean this up, we can display the poem using `cat()` instead:

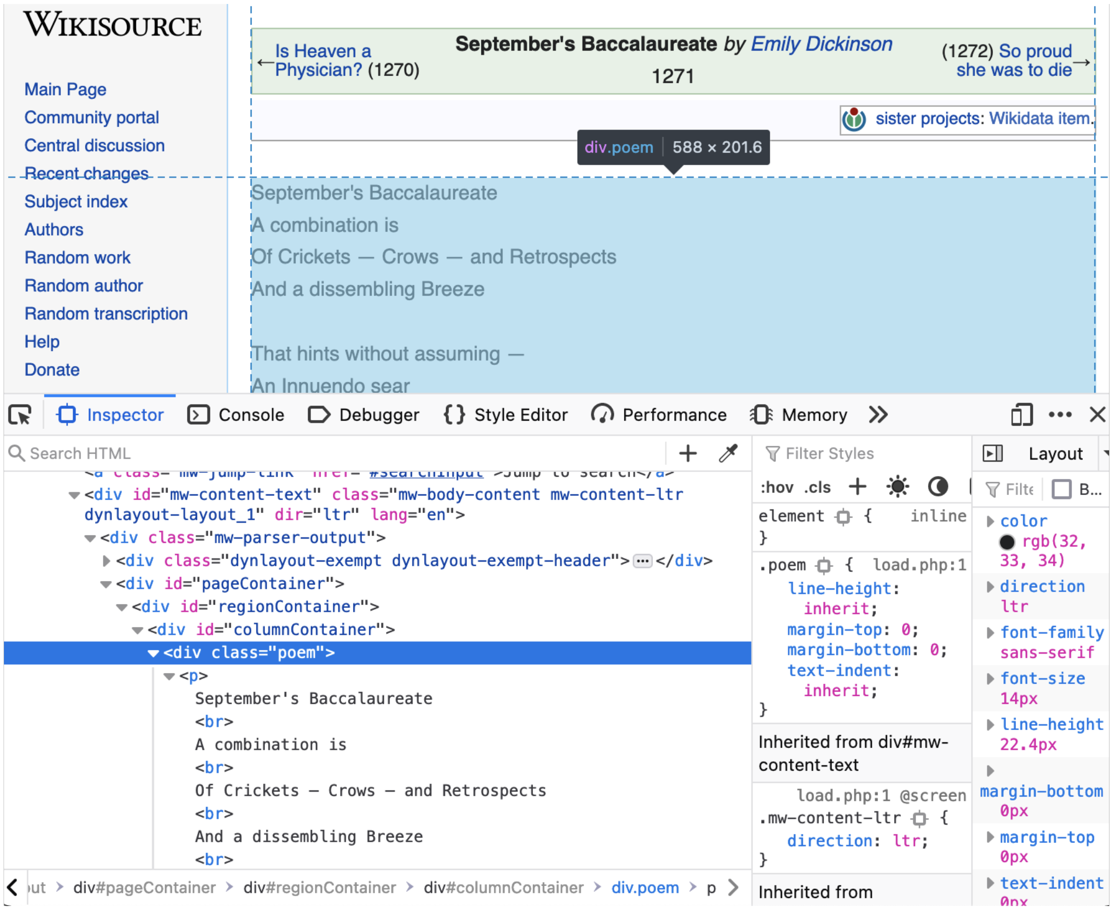
```
# Print clean text of poem
cat(sep_bac_text)
```

```
September's Baccalaureate
A combination is
Of Crickets - Crows - and Retrospects
And a dissembling Breeze
That hints without assuming -
An Innuendo sear
That makes the Heart put up its Fun
And turn Philosopher.
```

CSS selectors

Why `"div p"`? This is an example of a CSS selector, which allows us to identify the HTML elements of a webpage we want to select. The `read_html()` function reads in the HTML source code of a webpage and creates a list of elements based on the CSS selector you specify. In this case, `"div p"` tells R to select all `<p>` elements inside of `<div>` elements.

The screenshot below shows the highlighted source code and the corresponding part of the webpage using Firefox Web Developer Tools. If you look at the CSS Selector reference page linked above and the source code below, you might notice that using `html_elements(".poem")` would allow us to select the poem more directly (there's only one element on the page of `class="poem"` but 8 different `<p>` elements within `<div>` elements).



September's Baccalaureate *by Emily Dickinson*
1271

September's Baccalaureate
A combination is
Of Crickets — Crows — and Retrospects
And a dissembling Breeze

That hints without assuming —
An Innuendo sear

```
<a class="mw-jump-link" href="#searchInput">Jump to search</a>
<div id="mw-content-text" class="mw-body-content mw-content-ltr
dynlayout-layout_1" dir="ltr" lang="en">
  <div class="mw-parser-output">
    <div class="dynlayout-exempt dynlayout-exempt-header">…</div>
    <div id="pageContainer">
      <div id="regionContainer">
        <div id="columnContainer">
          <div class="poem">
            <p>
              September's Baccalaureate
              <br>
              A combination is
              <br>
              Of Crickets – Crows – and Retrospects
              <br>
              And a dissembling Breeze
              <br>
```

ut > div#pageContainer > div#regionContainer > div#columnContainer > div.poem > p

```
element {                 inline
}
.poem {               load.php:1
    line-height:
      inherit;
    margin-top: 0;
    margin-bottom: 0;
    text-indent:
      inherit;
}
Inherited from div#mw-content-text
      load.php:1 @screen
.mw-content-ltr {
    direction: ltr;
}
Inherited from
```

```
color
  ● rgb(32,
  33, 34)
direction
  ltr
font-family
  sans-serif
font-size
  14px
line-height
  22.4px
margin-bottom
  0px
margin-top
  0px
text-indent
  0px
```

While "div p" may often be the CSS selector you're looking for when extracting text from a web page, other CSS selectors will come in handy. I used the Firefox Web Developer Tool in my screenshot above and Safari also has an option to show a Web Inspector in its Develop menu, but SelectorGadget is a Chrome Extension that makes the process of identifying CSS Selectors much easier. With the extension activated, a box will open in the bottom right of the website. Click on a page element that you would like your selector to match, and SelectorGadget will then generate a minimal CSS selector for that element. It will also highlight all other elements that are matched by the selector.

You can click on a highlighted element to remove it from the selector, or click on an unhighlighted element to add it to the selector. Through this process of selection and removal, SelectorGadget helps you come up with the appropriate CSS selector for your needs.

More Info

See Web Scraping 101 for more details.