Scraping Notes - Chapter 6

Data Intake

R can read in data of many different formats. You have seen the read_csv function used in several assignments. We have also loaded data in from other packages in R (or from the base packages). If you have data in a different format, there is probably an R function to help ingest it. In the prep, you learned about the *haven* package which can help with certain data formats.

If you have a data file and aren't sure how to read it in, here are several ways to investigate options that may work:

- In your Environment window, hit the *Import Dataset* button and try options via the drop-down menu. (If it doesn't match any of the options, try the From Text (readr) one anyway.)
- Look up the file format and "how to read into R" via Google or another search engine. This may take you to a blog entry or an R package (where you can look for examples).

When reading in a data file, there are several issues to consider:

- Are variable names appropriate? (Run janitor's cleannames if you suspect you'll have issues once its in R.)
- Are missing values consistent across the data set? Make sure R knows how those are indicated. If it's NA for categorical variables but 9999999 for numbers, you have more work to do once the data is in R.
- Do you want all the rows? Sometimes when reading data in you only want every other row (for example, data that has labels repeat), or maybe you want to start on the 18th row due to some header information.
- Are variables in the format you expect? (If it's supposed to be numeric, is R treating it that way?)

You can set options for skipping rows, formatting variables, etc. in the read-in commands, usually. And if not, you can clean that up in R afterward.

Spend the time to look at your data and make sure everything is working the way you expect.

Sometimes though, we don't have a data file available and are interested in scraping data from a website. That's the focus of the rest of our notes today.

Ethics of Web Scraping

Be sure scraping is allowed on a site before attempting it. If scraping is not allowed directly, sometimes the website has an API that you may use, or be granted access to. There are even R packages that provide interfaces to some APIs.

How to check that scraping is allowed:

- Load the robotstxt package.
- Get the url of the desired webpage for scraping.
- Run the paths_allowed function on the url.
- If it returns TRUE, you are allowed to scrape. If it returns FALSE, you are not allowed to.

Example where you are NOT allowed to scrape.

```r
# ESPN College Football Recruiting Rankings
url <- "http://www.espn.com/college-sports/football/recruiting/playerrankings/_/view/rn300"
```

```
# Check bot permissions
paths_allowed(url) # FALSE means no scraping allowed!
```

```
##  www.espn.com
```

```
## [1] FALSE
```

Example where you are allowed to scrape.

```
# National Weather Service
url <- "https://www.weather.gov/"

# Check bot permissions
paths_allowed(url) # TRUE means you are allowed to scrape
```

```
## [1] TRUE
```

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").

General Structure for 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()
```

Text Example from Wikipedia

The main package used for scraping data from the web in R is *rvest*. The text example demos reading in data from a Wikipedia page, which is fairly well organized and easy to do. You worked with other tables from the same page in the Prep activity for the week.

Briefly, here is the text code again:

```
url <- "http://en.wikipedia.org/wiki/Mile_run_world_record_progression"
# Added check
paths_allowed(url)
```

```
##  en.wikipedia.org
```

```
## [1] TRUE
```

```
tables <- url %>%
  read_html() %>%
  html_nodes("table") #can be replaced with html_elements

# Count number of tables
length(tables)
```

```
## [1] 12
```

```
# Create table for amateur men prior to 1862
amateur <- tables %>%
  purrr::pluck(3) %>%
  html_table()

amateur
```

```
## # A tibble: 5 x 5
##   Time  Athlete             Nationality    Date             Venue
##   <chr> <chr>               <chr>          <chr>            <chr>
## 1 4:52  Cadet Marshall      United Kingdom 2 September 1852 Addiscome
## 2 4:45  Thomas Finch        United Kingdom 3 November 1858  Oxford
## 3 4:45  St. Vincent Hammick United Kingdom 15 November 1858 Oxford
## 4 4:40  Gerald Surman       United Kingdom 24 November 1859 Oxford
## 5 4:33  George Farran       United Kingdom 23 May 1862      Dublin
```

```
# Create table for records for men
records <- tables %>%
  purrr::pluck(4) %>%
  html_table() %>%
  select(-Auto)  # remove unwanted column

head(records) # earliest records
```

```
## # A tibble: 6 x 5
##   Time   Athlete           Nationality   Date              Venue
##   <chr>  <chr>             <chr>         <chr>             <chr>
## 1 4:14.4 John Paul Jones   United States 31 May 1913[6]    Allston, Mass.
## 2 4:12.6 Norman Taber      United States 16 July 1915[6]   Allston, Mass.
## 3 4:10.4 Paavo Nurmi       Finland       23 August 1923[6] Stockholm
## 4 4:09.2 Jules Ladoumègue France        4 October 1931[6] Paris
## 5 4:07.6 Jack Lovelock     New Zealand   15 July 1933[6]   Princeton, N.J.
## 6 4:06.8 Glenn Cunningham United States 16 June 1934[6]   Princeton, N.J.
```

```
tail(records) # most recent records
```

```
## # A tibble: 6 x 5
##   Time    Athlete           Nationality    Date                Venue
##   <chr>   <chr>             <chr>          <chr>               <chr>
## 1 3:48.53 Sebastian Coe     United Kingdom 19 August 1981[6]   Zürich
## 2 3:48.40 Steve Ovett        United Kingdom 26 August 1981[6]    Koblenz
## 3 3:47.33 Sebastian Coe      United Kingdom 28 August 1981[6]    Brussels
## 4 3:46.32 Steve Cram         United Kingdom 27 July 1985[6]      Oslo
## 5 3:44.39 Noureddine Morceli Algeria        5 September 1993[6] Rieti
## 6 3:43.13 Hicham El Guerrouj Morocco        7 July 1999[6]      Rome
```

Many wikipedia pages are set up in a similar format, so adapting this code to a different Wikipedia url is often pretty easy. But what if we want to do a non-Wikipedia page?

Hold on to your hats, we need to go learn a bit about HTML. (I am not an expert here, but can show you a few things.) There is a bit more detail about this below as well in the example scraping text.

Scraping from National Weather Service

We already checked we are able to scrape from the National Weather service. But let's go check out the specific forecast page for Amherst, so we can get some local weather data.

```r
# National Weather Service - Amherst, MA 01002 Forecast
url <- "https://forecast.weather.gov/MapClick.php?lat=42.377590000000055&lon=-72.51815999999997#.YhZQZJ

# Check bot permissions
paths_allowed(url) # TRUE means you are allowed to scrape
```

```
## [1] TRUE
```

Now, let's try the same code from the text and see what tables we get.

```r
tables <- url %>%
  read_html() %>%
  html_elements("table")

# Count number of tables
length(tables)
```

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

```r
onlytable <- tables %>%
  purrr::pluck(1) %>%
  html_table()

onlytable
```

```
## # A tibble: 7 x 2
##   X1          X2
##   <chr>       <chr>
## 1 Humidity    53%
## 2 Wind Speed  SW 15 G 25 mph
## 3 Barometer   30.01 in (1017.2 mb)
## 4 Dewpoint    22°F (-6°C)
## 5 Visibility  10.00 mi
## 6 Wind Chill  29°F (-2°C)
## 7 Last update 1 Mar 2:56 pm EST
```

There is only one table and it's the local current conditions.

If we visit the page though, we can see other information - like the extended forecast. How do we get that if it's not a table?

We need to learn a little about html and Web Developer Tools (which I learned about from the blog reference below which motivates this example).

I'm going to demo this on the classroom computer in Chrome, but you should be able to find the equivalent in any browser (I used Firefox and Chrome in my office while setting this up. I am unsure where the options would be in Safari, but they should exist.)

(Out to Demo)

So, the extended forecast information, at least, the temperature part of it, seems to be in *temp* objects.

```
url %>%
  read_html() %>%
  html_elements(".temp") %>%
  html_text() #it's not in tables!
```

```
## [1] "High: 39 °F" "Low: 30 °F"  "High: 40 °F" "Low: 28 °F"  "High: 33 °F"
## [6] "Low: 7 °F"   "High: 31 °F" "Low: 11 °F"  "High: 40 °F"
```

This would still need cleaned up some, but we were able to grab the numbers. If we save the object, we can try to parse the numbers out.

```
amherstTemps <- url %>%
  read_html() %>%
  html_elements(".temp") %>%
  html_text() %>%
  parse_number()

amherstTemps
```

```
## [1] 39 30 40 28 33  7 31 11 40
```

You could add the High/Low information in another variable:

```
amherstHL <- url %>%
  read_html() %>%
  html_elements(".temp") %>%
  html_text() %>%
  str_split(":", simplify = TRUE) %>%
  data.frame()
amherstHL
```

```
##     X1     X2
## 1 High   39 °F
## 2  Low   30 °F
## 3 High   40 °F
## 4  Low   28 °F
## 5 High   33 °F
## 6  Low    7 °F
## 7 High   31 °F
## 8  Low   11 °F
## 9 High   40 °F
```

```
amherstData <- bind_cols(amherstHL$X1, amherstTemps) %>%
  rename("High_Low" = "...1", "Temp" = "...2" )
```

```
## New names:
## * NA -> ...1
## * NA -> ...2
```

```
amherstData
```

```
## # A tibble: 9 x 2
##   High_Low  Temp
##   <chr>    <dbl>
## 1 High        39
## 2 Low         30
## 3 High        40
## 4 Low         28
## 5 High        33
## 6 Low          7
## 7 High        31
## 8 Low         11
## 9 High        40
```

There may be better ways to do this. We'll see more text analysis in the text analysis chapter.

For example, we see now that we could have gotten all the information in the amherstHL object and then just parsed the number out from its second column.

What if the data isn't in html?

If you are trying to scrape and the data doesn't seem to be stored in html, it could be in another web format. You can explore options with the *httr* and *jsonlite* packages, as examples of where you could start looking.

Another example

This example is motivated by many past stats projects where I've seen students scrape sports statistics. At times, that data is organized well on the pages and you can copy/paste from websites into .csv files fairly easily. That works, but is not as reproducible as having code that does the scraping for you. In particular, if you were grabbing statistics as a season progressed, you'd have to re-download the data every time a game was played if you were manually downloading the data sets.

While preparing these notes, I took a look at scraping sports statistics from a few different sites - ESPN, NBA.com, etc. For some, bots were NOT allowed. For others, bots were allowed but the data did not appear to be in html format.

For this example, we'll look at basketball-reference.com, and a related blog post that motivates this is cited below. I've updated and cleaned up the scraping code provided by the blog so it looks more like what we are used to. The structure of this page IS similar to the Wikipedia setup, so the reading of tables is pretty easy.

```
# URL of NBA All-Star Career Stats by Player
url <- "https://www.basketball-reference.com/allstar/NBA-allstar-career-stats.html"

# Check bot permissions
paths_allowed(url) # TRUE means you are allowed to scrape
```

```
##  www.basketball-reference.com
```

```
## [1] TRUE
```

```
tables <- url %>%
  read_html %>%
  html_elements("table")

length(tables)
```

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

```
all_star_data <- tables %>%
  purrr::pluck(1) %>%
  html_table()

glimpse(all_star_data)
```

```
## Rows: 481
## Columns: 30
## $ ``         <chr> "Player", "Kareem Abdul-Jabbar", "LeBron James", "Kobe Brya~
## $ ``         <chr> "G", "18", "18", "15", "15", "14", "14", "13", "13", "13", ~
## $ ``         <chr> "GS", "13", "18", "15", "12", "11", "2", "13", "9", "10", "~
## $ ``         <chr> "MP", "449", "509", "415", "311", "287", "227", "", "388", ~
## $ ``         <chr> "FG", "105", "172", "119", "63", "72", "49", "52", "72", "7~
## $ ``         <chr> "FGA", "213", "334", "238", "115", "141", "109", "158", "12~
## $ ``         <chr> "FG%", ".493", ".515", ".500", ".548", ".511", ".450", ".32~
## $ ``         <chr> "3P", "0", "40", "22", "1", "0", "10", "", "", "", "3", "",~
## $ ``         <chr> "3PA", "1", "130", "68", "4", "4", "34", "", "", "", "11", ~
## $ ``         <chr> "3P%", ".000", ".308", ".324", ".250", ".000", ".294", "", ~
## $ ``         <chr> "2P", "105", "132", "97", "62", "72", "39", "52", "72", "74~
## $ ``         <chr> "2PA", "212", "204", "170", "111", "137", "75", "158", "122~
## $ ``         <chr> "2P%", ".495", ".647", ".571", ".559", ".526", ".520", ".32~
## $ ``         <chr> "FT", "41", "29", "30", "13", "14", "14", "43", "47", "31",~
## $ ``         <chr> "FTA", "50", "40", "38", "17", "16", "16", "51", "94", "41"~
## $ ``         <chr> "FT%", ".820", ".725", ".789", ".765", ".875", ".875", ".84~
## $ ``         <chr> "ORB", "33", "14", "28", "38", "25", "15", "", "", "2", "22~
## $ ``         <chr> "DRB", "84", "95", "47", "98", "63", "37", "", "", "10", "3~
## $ ``         <chr> "TRB", "149", "109", "75", "136", "88", "52", "78", "197", ~
## $ ``         <chr> "AST", "51", "106", "70", "31", "40", "16", "86", "36", "31~
## $ ``         <chr> "STL", "6", "22", "38", "13", "16", "10", "", "", "4", "37"~
## $ ``         <chr> "BLK", "31", "8", "6", "8", "11", "5", "", "", "0", "6", ""~
## $ ``         <chr> "TOV", "28", "58", "35", "31", "20", "14", "", "", "4", "42~
## $ ``         <chr> "PF", "57", "18", "35", "16", "10", "8", "27", "23", "20", ~
## $ ``         <chr> "PTS", "251", "413", "290", "140", "158", "122", "147", "19~
## $ ``         <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,~
## $ `Per Game` <chr> "MP", "24.9", "28.3", "27.7", "20.7", "20.5", "16.2", "", "~
## $ `Per Game` <chr> "PTS", "13.9", "22.9", "19.3", "9.3", "11.3", "8.7", "11.3"~
## $ `Per Game` <chr> "TRB", "8.3", "6.1", "5.0", "9.1", "6.3", "3.7", "6.0", "15~
## $ `Per Game` <chr> "AST", "2.8", "5.9", "4.7", "2.1", "2.9", "1.1", "6.6", "2.~
```

However, we can see that the table we get needs some cleaning up.

The variable names are in the first row! Also, some of the variable names look like they might be hard to work with in R. The blog shows how to fix this (you already know how to fix the names!). (Again, I've cleaned up comments and formatting a bit.)

```
# put variable names in appropriately
all_star_data <- row_to_names(all_star_data,
                              1,
                              remove_row = TRUE,
                              remove_rows_above = FALSE)
```

```
## Warning in row_to_names(all_star_data, 1, remove_row = TRUE, remove_rows_above
## = FALSE): Row 1 does not provide unique names. Consider running clean_names()
## after row_to_names().
```

```r
# shows names are not clean
names(all_star_data)
```

```
##  [1] "Player" "G"      "GS"     "MP"     "FG"     "FGA"    "FG%"    "3P"
##  [9] "3PA"    "3P%"    "2P"     "2PA"    "2P%"    "FT"     "FTA"    "FT%"
## [17] "ORB"    "DRB"    "TRB"    "AST"    "STL"    "BLK"    "TOV"    "PF"
## [25] "PTS"    NA       "MP"     "PTS"    "TRB"    "AST"
```

```r
# clean names
all_star_data <- clean_names(all_star_data)
names(all_star_data)
```

```
##  [1] "player"      "g"           "gs"          "mp"          "fg"
##  [6] "fga"         "fg_percent"  "x3p"         "x3pa"        "x3p_percent"
## [11] "x2p"         "x2pa"        "x2p_percent" "ft"          "fta"
## [16] "ft_percent"  "orb"         "drb"         "trb"         "ast"
## [21] "stl"         "blk"         "tov"         "pf"          "pts"
## [26] "na"          "mp_2"        "pts_2"       "trb_2"       "ast_2"
```

You could explore more with the data set now that it's in R and cleaned up some. The blog continues doing some other cleaning and analysis, but this was the part about getting the data into R.

(P.S. If you decide to go read the blog entry, the blog data cleaning could be improved. Please follow our style guide and use the tidyverse verbs and pipes whenever possible.)

Web scraping workflow

In the prep for this week, 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.

What would this workflow look like?

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.

Why would we do this?

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.

Example - Web scraping text

We often want information from a website that is *not* stored in table format. We want to get some poetry data so that in a few weeks we can 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.

These were both demo-ed in the Weather example above as well.

```
# 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)
```

```
##  en.wikisource.org
```

```
## [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 disse
```

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

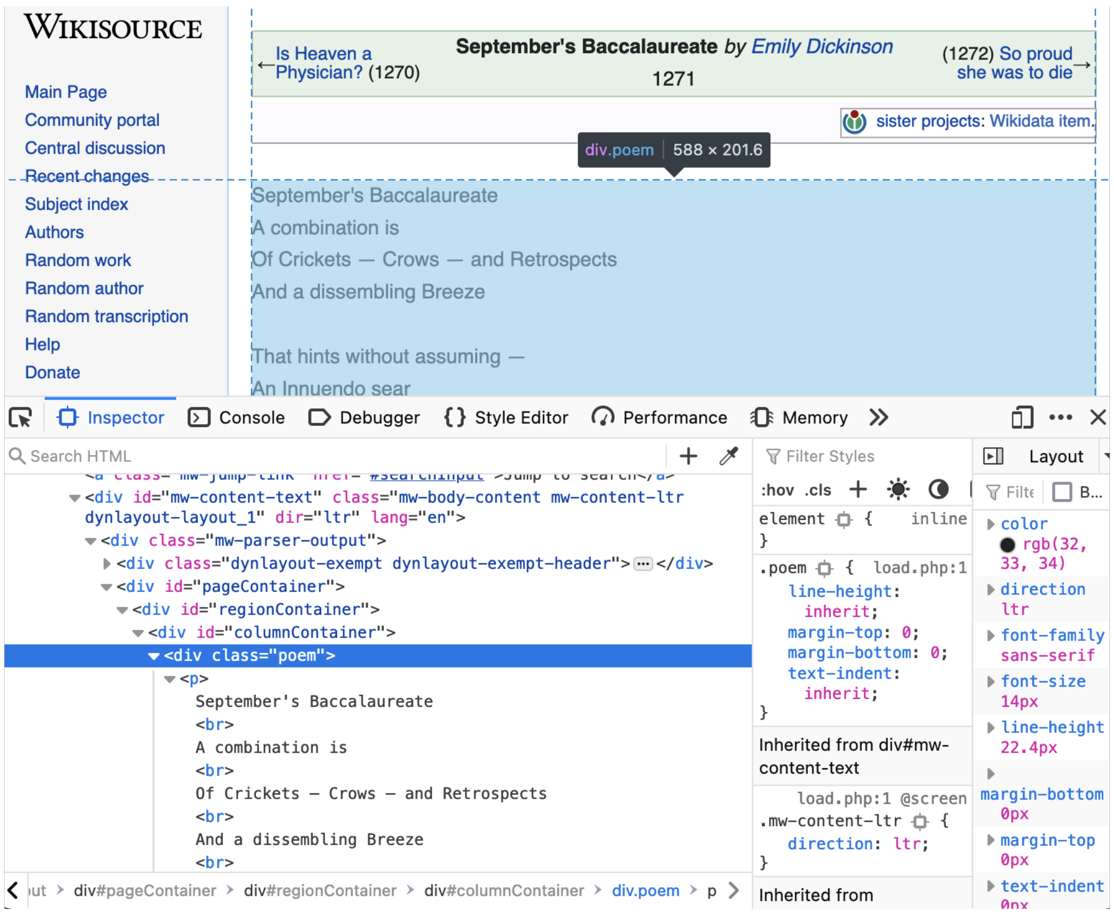
```
# Print clean text of poem
cat(sep_bac_text)
```

```
## 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

```
<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>
```

```
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
```

References

Blog entry - https://www.dataquest.io/blog/web-scraping-in-r-rvest/

Sports Scraping - https://medium.com/analytics-vidhya/web-scraping-nba-all-star-data-8788a7136727

Thanks to prior Stat 231 instructors for the poem text scraping and CSS selector details in the examples.