


Dungeons and Dragons Web Scraping with rvest and RSelenium

Aug 10, 2018 · 17 min read · 1 Comment (https://lmyint.github.io/post/dnd-scraping-rvest-rselenium/#disqus_thread) ·  R (<https://lmyint.github.io/categories/r/>)

scraping%20with%20rvest%20and%20RSelenium&url=https%3a%2f%2flmyint.github.io%2fpost%2fdnd-

scraping-rvest-rselenium%2f)

&url=https%3a%2f%2flmyint.github.io%2fpost%2fdnd-scraping-rvest-
%20Web%20Scraping%20with%20rvest%20and%20RSelenium)

tps%3a%2f%2flmyint.github.io%2fpost%2fdnd-scraping-rvest-
%20Web%20Scraping%20with%20rvest%20and%20RSelenium)

.0Scraping%20with%20rvest%20and%20RSelenium&body=https%3a%2f%2flmyint.github.io%2fpost%2fdnd-

I love Dungeons and Dragons. I am also a data-loving statistician. At some point, these worlds were bound to collide.

For those unfamiliar with Dungeons and Dragons (DnD), it is a role-playing game that is backed by an extraordinary amount of data. The overall gist is that players create characters that band together with other characters to travel the world and adventure. Essentially, it's collective storytelling aided by dice as vehicles of chance and uncertainty. Where does data come in? Through the world-building content that is released by the official creators and by players. This

content helps players build characters that have a range of characteristics and abilities governed by their past and occupation. This content similarly helps shape the monsters and enemies that the characters may face.

There is a wonderful digital resource for DnD content called DnD Beyond (<https://www.dndbeyond.com/>) that contains information on characters, monsters, and treasures. (No API yet, but it is apparently in the works (<https://twitter.com/dndbeyond/status/909834529736740864?lang=en>).) For a while, I've been interested in playing around with data on monster statistics, and I finally got around to it this week! I had been reluctant to start because I did not have a clear idea of how to scrape pages that required login via redirect to an external authentication service (here, Twitch). I'll go over the specific hurdles and solutions in this post. I'll also give a general tutorial for scraping with `rvest`.

All of the code for this post is available at https://github.com/lmyint/dnd_analysis (https://github.com/lmyint/dnd_analysis).

Table of Contents

1. Structure of the scraping task
2. Step 1: Scrape tables to get individual monster page URLs
 - General structure of `rvest` code
 - `SelectorGadget`
 - Extract URLs
3. Step 2: Use `RSelenium` to access pages behind login
 - What did not work
 - What did work
 - Step 2a: Start automated browsing with `rsDriver`
 - Step 2b: Browser navigation and interaction
 - Step 2c: Extract page source
4. Step 3: Write a function to scrape an individual page
5. Summary

Structure of the scraping task

If you go to <https://www.dndbeyond.com/monsters> (<https://www.dndbeyond.com/monsters>), you will see the first of several tens of pages of monster listings. You will also see that each monster name is a link to an individual monster page that contains more extensive details about that monster's statistics, abilities, and lore. An example that is free to view is the Adult Green Dragon

(<https://www.dndbeyond.com/monsters/adult-green-dragon>). Other monsters that are not part of the Basic Rules set can only be viewed if you are signed in and have purchased the digital book in which that monster is contained.

The first part of the scraping task is to scrape the several pages of tables starting at <https://www.dndbeyond.com/monsters> (<https://www.dndbeyond.com/monsters>) in order to get the links to individual monster pages.

The second part of the scraping task is to scrape the individual monster pages, such as the Adult Green Dragon (<https://www.dndbeyond.com/monsters/adult-green-dragon>).

Throughout, I use the following packages:

- `rvest` (<https://cran.r-project.org/web/packages/rvest/index.html>) for page scraping
- `stringr` (<https://cran.r-project.org/web/packages/stringr/index.html>) for working with strings
- `tibble` (<https://cran.r-project.org/web/packages/tibble/index.html>) for the flexibility over data frames to allow list-columns
- `RSelenium` (<https://github.com/ropensci/RSelenium>) for browser navigation via R. This package was on CRAN but removed in May 2018. I used the development version on GitHub, but the package maintainer is currently working to fix this (<https://github.com/ropensci/RSelenium/issues/172>).

Step 1: Scrape tables to get individual monster page URLs

By visiting a few different pages of monster results, we can see that the URLs for the page results have a consistent format: `https://www.dndbeyond.com/monsters?page=NUM` where `NUM` ranges from 1 to the last page. We can obtain the last page number programatically with the following:

```
page <- read_html("https://www.dndbeyond.com/monsters")
num_pages <- page %>%
  html_nodes(".b-pagination-item") %>%
  html_text() %>%
  as.integer() %>%
  max(na.rm = TRUE)
```

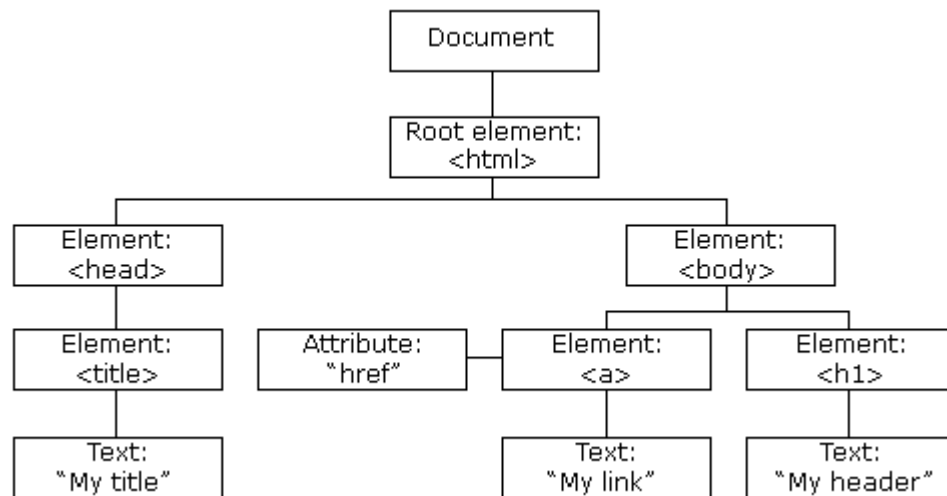
Let's explore the anatomy of this code to better understand how to work with `rvest`.

General structure of `rvest` code

The small example above shows the power of `rvest`. In many cases, the code to scrape content on a webpage really does boil down to something as short as:

```
url %>% read_html() %>% html_nodes("CSS or XPATH selector") %>% html_text() OR html_attr
```

- We start with a URL string that is passed to the `read_html` function. This creates an XML document object which is a tree representation of the content on a webpage. Why a tree? This requires some familiarity with HTML, but essentially text content is nested within enclosing formatting tags to make up a webpage. The following diagram from a W3Schools tutorial (https://www.w3schools.com/js/js_htmlDOM_navigation.asp) illustrates this.



- The `html_nodes` function takes a string specifying the HTML tags that you desire to be selected. The selector string can be a CSS or XPATH selector. I only know about CSS selectors, and that has sufficed for all of my web scraping to date. This function returns a list of nodes that have been selected from the HTML tree. For example, selecting the `<body>` tag is like grabbing the trunk of the HTML tree, and selecting paragraph `<p>` tags is like grabbing only the thinner branches. This list of nodes is still a list of XML objects.
- Usually what we want in scraping is the text that we see on the webpage that is contained within the specific sections extracted with `html_nodes`. We can get this text with `html_text`. Often we will also want attributes of the text on a webpage. For example, we may see text that is actually a link, and we want the URL for that link. In this case `html_text` would not give us what we want because it would give us the link text. However, `html_attr` will allow us to extract the URL. A specific example of this in just a second!

SelectorGadget

Back to the code example above:

```
page <- read_html("https://www.dndbeyond.com/monsters")
num_pages <- page %>%
  html_nodes(".b-pagination-item") %>%
  html_text() %>%
  as.integer() %>%
  max(na.rm = TRUE)
```

The most difficult part of this part of code is figuring out the selector to use in `html_nodes`. Luckily, the `rvest` package page on CRAN has a link to a vignette on a tool called SelectorGadget (<https://cran.r-project.org/web/packages/rvest/vignettes/selectorgadget.html>). I love this tool for its playful homage to childhood memories (<https://www.youtube.com/watch?v=e-JHfXVlkik>), and it also greatly helps in determining the CSS selectors needed to select desired parts of a webpage. Once you have dragged the tool link to the bookmark bar, you can click the bookmark while viewing any page to get a hover tool that highlights page elements as you mouse over them. Clicking on an element on the page displays the text for the CSS selector in the tool panel.

Using the SelectorGadget tool, we can determine that the page number buttons on the main monster page (<https://www.dndbeyond.com/monsters>) all have the class `b-pagination-item`. The CSS selector for a class always starts with a period followed by the class name. The last page was the maximum of these numbers. (We need to remove `NA`'s created by integer coercion of the "Next" button.)

Extract URLs

Now that we know how many pages (`num_pages`) to loop through, let's write a function that will extract the URLs for the individual monster pages that are present on a single page of results.

```
get_monster_links <- function(url) {
  page <- read_html(url)
  rel_links <- page %>%
    html_nodes(".link") %>%
    html_attr(name = "href")
  keep <- str_detect(rel_links, "/monsters/")
  rel_links <- rel_links[keep]
  abs_links <- paste0("https://www.dndbeyond.com", rel_links)
  abs_links
}
```

The `get_monster_links` function takes as input a URL for a page of results (like <https://www.dndbeyond.com/monsters?page=2> (<https://www.dndbeyond.com/monsters?page=2>)). Let's work through the function body:

- We first read the HTML source of a page with `read_html`.
- We can then use a combination of `SelectorGadget` with the "View page source" functionality of your browser to select the links on the page. Here we find that they belong to class `link`.
- We use the `html_attr` function here to extract the link path rather than the link text. The `name = "href"` specifies that we want the path attribute. (Anatomy of an HTML link: `Link text seen on page`).
- The remainder of the function subsets the extracted links to only those that pertain to the monster pages (removing links like the home page). Printing the output indicates that these links are only relative links, so we append the base URL to create absolute links (`abs_links`).

Finally, we can loop through all pages of results to get the hundreds of pages for the individual monsters:

```
## Loop through all pages
all_monster_urls <- lapply(seq_len(num_pages), function(i) {
  url <- paste0("https://www.dndbeyond.com/monsters?page=", i)
  get_monster_links(url)
}) %>% unlist
```

Step 2: Use RSelenium to access pages behind login

In Step 1, we looped through pages of tables to get the URLs for pages that contain detailed information on individual monsters. Great! We can visit each of these pages and just do some more `rvest` work to scrape the details! Well... not immediately. Most of these monster pages can only be seen if you have paid for the corresponding digital books and are logged in. DnD Beyond uses Twitch (<https://www.twitch.tv/>) for authentication which involves a redirect. This redirect made it way harder for me to figure out what to do. It was like I had been thrown into the magical, mysterious, and deceptive realm of the Feywild (<http://forgottenrealms.wikia.com/wiki/Feywild>) where I frantically invoked Google magicks to find many dashed glimmers of hope but luckily a solution in the end.

What did not work

It's helpful for me to record what things I tried and failed so I can remember my thought process. Hopefully, it saves you wasted effort if you're ever in a similar situation.

- Using `rvest` 's page navigation abilities did not work. I tried the following code:

```
url <- "https://www.dndbeyond.com/login"
session <- html_session(url)
url <- follow_link(session, "Login")
```

But I ran into an error:

```
Error in curl::curl_fetch_memory(url, handle = handle) :  
  Could not resolve host: NA
```

- Using `rvest` 's basic authentication abilities did not work. I found this tutorial (<https://github.com/rstudio/webinars/blob/master/32-Web-Scraping/navigation-and-authentication.md>) on how to send a username and password to a form with `rvest` . I tried hardcoding the extremely long URL that takes you to a Twitch authentication page, sending my username and password as described in the tutorial, and following [this Stack Overflow suggestion] to create a fake login button since the authentication page had an unnamed, unlabeled "Submit" input that did not seem to conform to `rvest` 's capabilities. I got a 403 error.

What did work

Only when I stumbled upon this Stack Overflow post

(<https://stackoverflow.com/questions/40198182/403-error-when-using-rvest-to-log-into-website-for-scraping>) did I learn about the `RSelenium` package. Selenium (<https://www.seleniumhq.org/>) is a tool for automating web browsers, and the `RSelenium` package is the R interface for it.

I am really grateful to the posters on that Stack Overflow question and this blog post (<https://abdallaabdi.com/2016/02/13/navigating-scraping-job-sites-rvest-rselenium/>) for getting me started with `RSelenium` . The only problem is that the `startServer` function used in both posts is now defunct. When calling `startServer` , the message text informs you of the `rsDriver` function.

Step 2a: Start automated browsing with `rsDriver`

The amazing feature of the `rsDriver` function is that you do not need to worry about downloading and installing other software like Docker or phantomjs. This function works right out of the box! To start the automated browsing, use the following:

```
rd <- rsDriver(browser = "chrome")
rem_dr <- rd[["client"]]
```

When you first run `rsDriver`, status messages will indicate that required files are being downloaded. After that you will see the status text “Connecting to remote server” and a Chrome browser window will pop open. The browser window will have a message beneath the search bar saying “Chrome is being controlled by automated test software.” This code comes straight from the example in the `rsDriver` help page.

Step 2b: Browser navigation and interaction

The `rem_dr` object is what we will use to navigate and interact with the browser. This navigation and interaction is achieved by accessing and calling functions that are part of the `rem_dr` object. We can navigate to a page using the `$navigate()` function. We can select parts of the webpage with the `$findElement()` function. Once these selections are made, we can interact with the selections by

- Sending text to those selections with `$sendKeysToElement()`
- Sending key presses to those selections with `$sendKeysToElement()`
- Sending clicks to those selections with `$clickElement()`

All of these are detailed in the RSelenium Basics vignette (<http://rpubs.com/johndharrison/RSelenium-Basics>), and further examples are in the Stack Overflow (<https://stackoverflow.com/questions/40198182/403-error-when-using-rvest-to-log-into-website-for-scraping>) and blog post (<https://abdallaabdi.com/2016/02/13/navigating-scraping-job-sites-rvest-rselenium/>) I mentioned above.

The code below shows this functionality in action:

```
url <- "https://www.dndbeyond.com/login"
rem_dr$navigate(url) # Navigate to login page
rem_dr$findElement(using = "css selector", value = ".twitch-button")$clickElement() # (
## Manually enter username and password here
rem_dr$findElement(using = "css selector", value = ".js-authorize-text")$clickElement()
```

Note: Once the Chrome window opens, you can finish the login process programatically as above or manually interface with the browser window as you would normally. This can be safer if you don't want to have a file with your username and password saved anywhere.

Step 2c: Extract page source

Now that we have programatic control over the browser, how do we interface with `rvest` ? Once we navigate to a page with `$navigate()` , we will need to extract the page's HTML source code to supply to `rvest::read_html` . We can extract the source with `$getPageSource()` :

```
rem_dr$navigate(url)
page <- read_html(rem_dr$getPageSource()[[1]])
```

The subset `[[1]]` is needed after calling `rem_dr$getPageSource()` because `$getPageSource()` returns a list of length 1. The HTML source that is read in can be directly input to `rvest::read_html` .

Excellent! Now all we need is a function that scrapes the details of a monster page and loop! In the following, we put everything together in a loop that iterates over the vector of URLs (`all_monster_urls`) generated in Step 1.

Within the loop we call the custom `scrape_monster_page` function to be discussed below in Step 3. We also include a check for purchased content. If you try to access a monster page that is not part of books that you have paid for, you will be redirected to a new page. We perform this check with the `$getCurrentUrl()` function, filling in a missing value for the monster information if we do not have access. The `Sys.sleep` at the end can be useful to avoid overloading your computer or if rate limits are a problem.

```
monster_info <- vector("list", length(all_monster_urls))
for (i in seq_along(all_monster_urls)) {
  url <- all_monster_urls[i]
  rem_dr$navigate(url)
  page <- read_html(rem_dr$getPageSource()[[1]])
  ## If content has not been unlocked, the page will redirect
  curr_url <- rem_dr$getCurrentUrl()[[1]]
  if (curr_url == url) {
    monster_info[[i]] <- scrape_monster_page(page)
  } else {
    monster_info[[i]] <- NA
  }
  Sys.sleep(2)
  cat(i, " ")
}
```

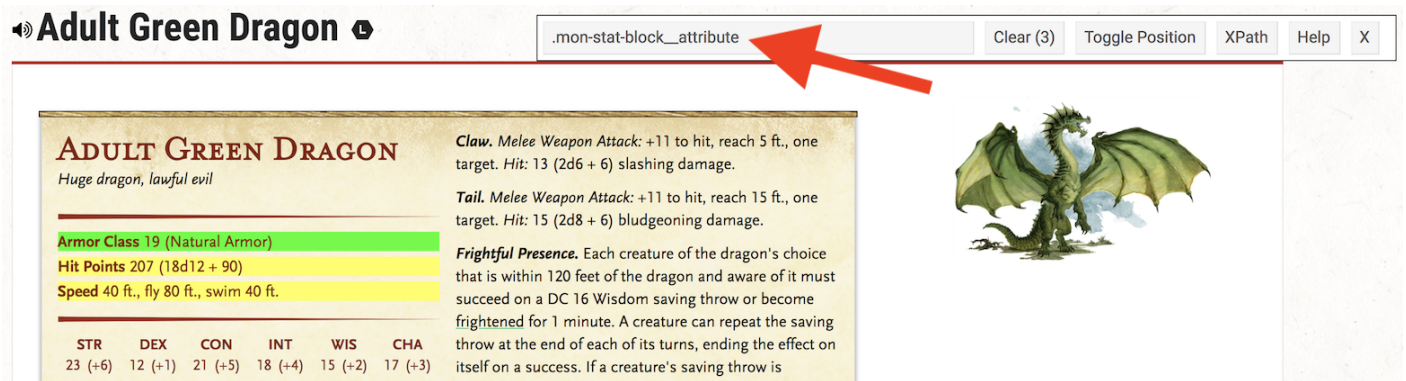
Step 3: Write a function to scrape an individual page

The last step in our scraping endeavor is to write the `scrape_monster_page` function to scrape data from an individual monster page. You can view the full function on GitHub (https://github.com/lmyint/dnd_analysis/blob/master/scrape_monster_stats.R). I won't go through

every aspect of this function here, but I'll focus on some principles that appear in this function that I've found to be useful in general when working with `rvest`.

Principle 1: Use SelectorGadget AND view the page's source

As useful as SelectorGadget is for finding the correct CSS selector, I never use it alone. I always open up the page's source code and do a lot of Ctrl-F to quickly find specific parts of a page. For example, when I was using SelectorGadget to get the CSS selectors for the Armor Class, Hit Points, and Speed attributes, I saw the following:



I wanted to know if there were further subdivisions of the areas that the `.mon-stat-block__attribute` selector had highlighted. To do this, I searched the source code for "Armor Class" and found the following:

```

<div class="mon-stat-block__attribute">
  <span class="mon-stat-block__attribute-label">Armor Class</span>
  <span class="mon-stat-block__attribute-value">
    <span class="mon-stat-block__attribute-data-value">
      19
    </span>

    <span class="mon-stat-block__attribute-data-extra">
      (Natural Armor)
    </span>

  </span>
</div>
<div class="mon-stat-block__attribute">
  <span class="mon-stat-block__attribute-label">Hit Points</span>
  <span class="mon-stat-block__attribute-data">
    <span class="mon-stat-block__attribute-data-value">
      207
    </span>
    <span class="mon-stat-block__attribute-data-extra">
      (18d12 + 90)
    </span>
  </span>
</div>
<div class="mon-stat-block__attribute">
  <span class="mon-stat-block__attribute-label">Speed</span>
  <span class="mon-stat-block__attribute-data">
    <span class="mon-stat-block__attribute-data-value">
      40 ft., fly 80 ft., swim 40 ft.

    </span>
  </span>
</div>

```

Looking at the raw source code allowed me to see that each line was subdivided by spans with classes `mon-stat-block__attribute-label`, `mon-stat-block__attribute-data-value`, and sometimes `mon-stat-block__attribute-data-extra`.

With SelectorGadget, you can actually type a CSS selector into the text box to highlight the selected parts of the page. I did this with the `mon-stat-block__attribute-label` class to verify that there should be 3 regions highlighted.

Adult Green Dragon

.mon-stat-block__attribute-label

Clear (3) Toggle Position XPath Help X

ADULT GREEN DRAGON
Huge dragon, lawful evil


Armor Class 19 (Natural Armor)
Hit Points 207 (18d12 + 90)
Speed 40 ft., fly 80 ft., swim 40 ft.

STR	DEX	CON	INT	WIS	CHA
23 (+6)	12 (+1)	21 (+5)	18 (+4)	15 (+2)	17 (+3)

Claw. *Melee Weapon Attack:* +11 to hit, reach 5 ft., one target. *Hit:* 13 (2d6 + 6) slashing damage.

Tail. *Melee Weapon Attack:* +11 to hit, reach 15 ft., one target. *Hit:* 15 (2d8 + 6) bludgeoning damage.

Frightful Presence. Each creature of the dragon's choice that is within 120 feet of the dragon and aware of it must succeed on a DC 16 Wisdom saving throw or become frightened for 1 minute. A creature can repeat the saving throw at the end of each of its turns, ending the effect on itself on a success. If a creature's saving throw is successful on the effect ends for that creature.



Because SelectorGadget requires hovering your mouse over potentially small regions, it is best to verify your selection by looking at the source code.

Principle 2: Print often

Continuing from the above example of desiring the Armor Class, Hit Points, and Speed attributes, I was curious what I would obtain if I simply selected the whole line for each attribute (as opposed to the three subdivisions). The following is what I saw when I printed this to the screen:

```
> page %>% html_nodes(".mon-stat-block__attribute") %>% html_text()
[1] "\n          Armor Class\n          \n          19'
[2] "\n          Hit Points\n          \n          207'
[3] "\n          Speed\n          \n          40 ft., "
```

A mess! A length-3 character vector containing the information I wanted but not in a very tidy format. Because I want to visualize and explore this data later, I want to do a little tidying up front in the scraping process.

What if I just access the three subdivisions separately and `rbind` them together? This is not a good idea because of missing elements as shown below:

```
> page %>% html_nodes(".mon-stat-block__attribute-label") %>% html_text()
[1] "Armor Class" "Hit Points" "Speed"
> page %>% html_nodes(".mon-stat-block__attribute-data-value") %>% html_text() %>% trimws()
[1] "19" "207"
[3] "40 ft., fly 80 ft., swim 40 ft."
> page %>% html_nodes(".mon-stat-block__attribute-data-extra") %>% html_text() %>% trimws()
[1] "(Natural Armor)" "(18d12 + 90)"
```

For `attribute-label`, I get a length-3 vector. For `attribute-data-value`, I get a length-3 vector. For `attribute-data-value`, I only get a length-2 vector! Through visual inspection, I know that the third line "Speed" is missing the span with the `data-extra` class, but I don't want to rely on visual inspection for these hundreds of monsters! **Printing these results warned me directly that this could happen!** Awareness of these missing items motivates the third principle.

Principle 3: You will need loops

For the Armor Class, Hit Points, and Speed attributes, I wanted to end up with a data frame that looks like this:

```
> attrs
# A tibble: 3 x 3
  label      value      extra
  <chr>    <chr>    <chr>
1 Armor Class 19      (Natural Armor)
2 Hit Points  207      (18d12 + 90)
3 Speed      40 ft., fly 80 ft., swim 40 ft. NA
```

This data frame has properly encoded missingness. To do this, I needed to use a loop as shown below.

```
## Attributes: AC, HP, speed
attr_nodes <- page %>%
  html_nodes(".mon-stat-block__attribute")
attrs <- do.call(rbind, lapply(attr_nodes, function(node) {
  label <- node %>%
    select_text(".mon-stat-block__attribute-label")
  data_value <- node %>%
    select_text(".mon-stat-block__attribute-data-value")
  data_extra <- node %>%
    select_text(".mon-stat-block__attribute-data-extra") %>%
    replace_if_empty(NA)
  tibble(label = label, value = data_value, extra = data_extra)
}))
```

The code below makes use of two helper functions that I wrote to cut down on code repetition:

- `select_text` to cut down on the repetitive `page %>% html_nodes %>% html_text`

```
select_text <- function(xml, selector, trim = TRUE) {
  text <- xml %>%
    html_nodes(selector) %>%
    html_text
  if (trim) {
    text <- text %>%
      trimws
  }
  text
}
```

- `replace_if_empty` to replace empty text with NA

```
replace_if_empty <- function(text, to) {
  if (length(text)==0) {
    text <- to
  }
  text
}
```

I first select the three lines corresponding to these three attributes with

```
attr_nodes <- page %>%
  html_nodes(".mon-stat-block__attribute")
```

This creates a list of three nodes (pieces of the webpage/branches of the HTML tree) corresponding to the three lines of data:

```
> attr_nodes
{xml_nodeset (3)}
[1] <div class="mon-stat-block__attribute">\n      <span class="mon-sta ...
[2] <div class="mon-stat-block__attribute">\n      <span class="mon-sta ...
[3] <div class="mon-stat-block__attribute">\n      <span class="mon-sta ...
```

We can chain together a series of calls to `html_nodes` . I do this in the subsequent `lapply` statement. I know that each of these nodes contains up to three further subdivisions (label, value, and extra information). In this way I can make sure that these three pieces of information are aligned between the three lines of data.

Nearly all of the code in the `scrape_monster_page` function repeats these three principles, and I've found that I routinely use similar ideas in other scraping I've done with `rvest` .

Summary

This is a long post, but a few short take-home messages suffice to wrap ideas together:

- `rvest` is remarkably effective at scraping what you need with fairly concise code. Following the three principles above has helped me a lot when I've used this package.
- `rvest` can't do it all. For scraping tasks where you wish that you could automate clicking and typing in the browser (e.g. authentication settings), `RSelenium` is the package for you. In particular, the `rsDriver` function works right out of the box (as far as I can tell) and is great for people like me who are loath to install external dependencies.

Happy scraping!

[rvest](https://lmyint.github.io/tags/rvest/) (https://lmyint.github.io/tags/rvest/)

[rselenium](https://lmyint.github.io/tags/rselenium/) (https://lmyint.github.io/tags/rselenium/)

[dnd](https://lmyint.github.io/tags/dnd/) (https://lmyint.github.io/tags/dnd/)

1 Comment **lesliemyint**

 Login ▾

 Recommend 1

 Tweet

 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



Wawv • a month ago

Thanks for the tutorial.

I finally managed to automate my data collection process from a website that automatically authenticated me from the browser (I couldn't find how to do it with simple http GET/authenticate request).

^ | ▾ • Reply • Share ›

 Subscribe  Add Disqus to your siteAdd DisqusAdd  Disqus' Privacy PolicyPrivacy PolicyPrivacy

