Lecture 6: Data Acquisition - Scraping Static Websites

James Sears*
AFRE 891 SS 24
Michigan State University

*Parts of these slides are adapted from **"Data Science for Economists"** by Grant McDermott.

Table of Contents

Prologue

Packages for Today

Today we'll pick up with scraping static websites.

For this we'll need the following packages:

```
pacman::p_load(lubridate, rvest, stringr, tiydverse)
```

Scraping Static Websites

Scraping Static Sites

Last time we saw how to use **rvest** functions to scrape information from **HTML Tables**.

- **Biggest Challenge:** finding the right CSS selectors

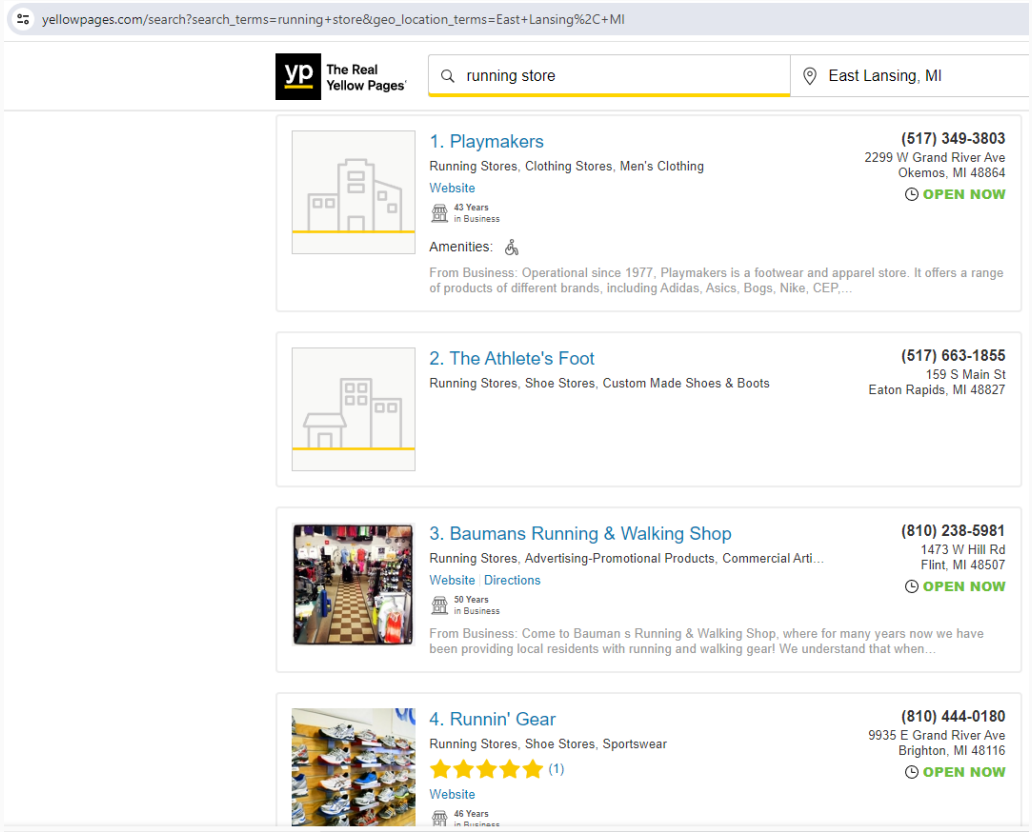
Sometimes, however, the data we want to scrape *aren't* in a nice table format already.

Let's work through an example: **Yellow Pages**.

Application 2: Yellow Pages

Let's say you got excited by all this marathon talk and want to start running.

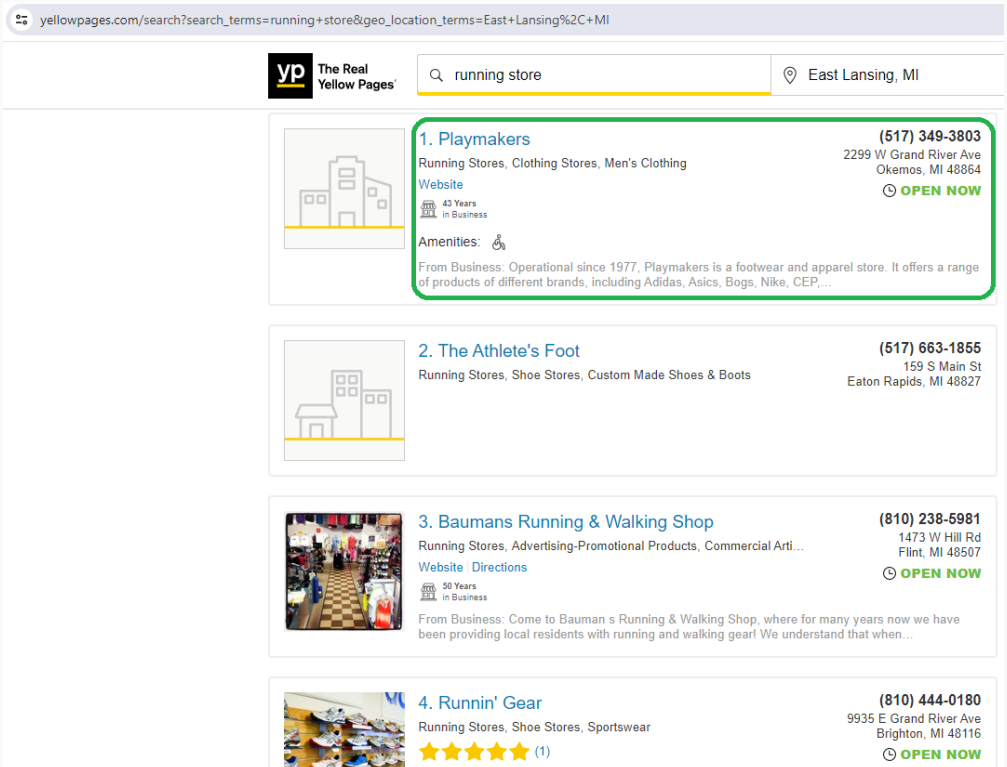
Naturally, you go to **Yellowpages.com** and search for "running stores" in East Lansing



SelectorGadget vs. Source

Let's scrape the descriptive info for each of the stores to help us figure out where to go first.

Try using SelectorGadget to find the selector for the info in **Playmakers**' box.



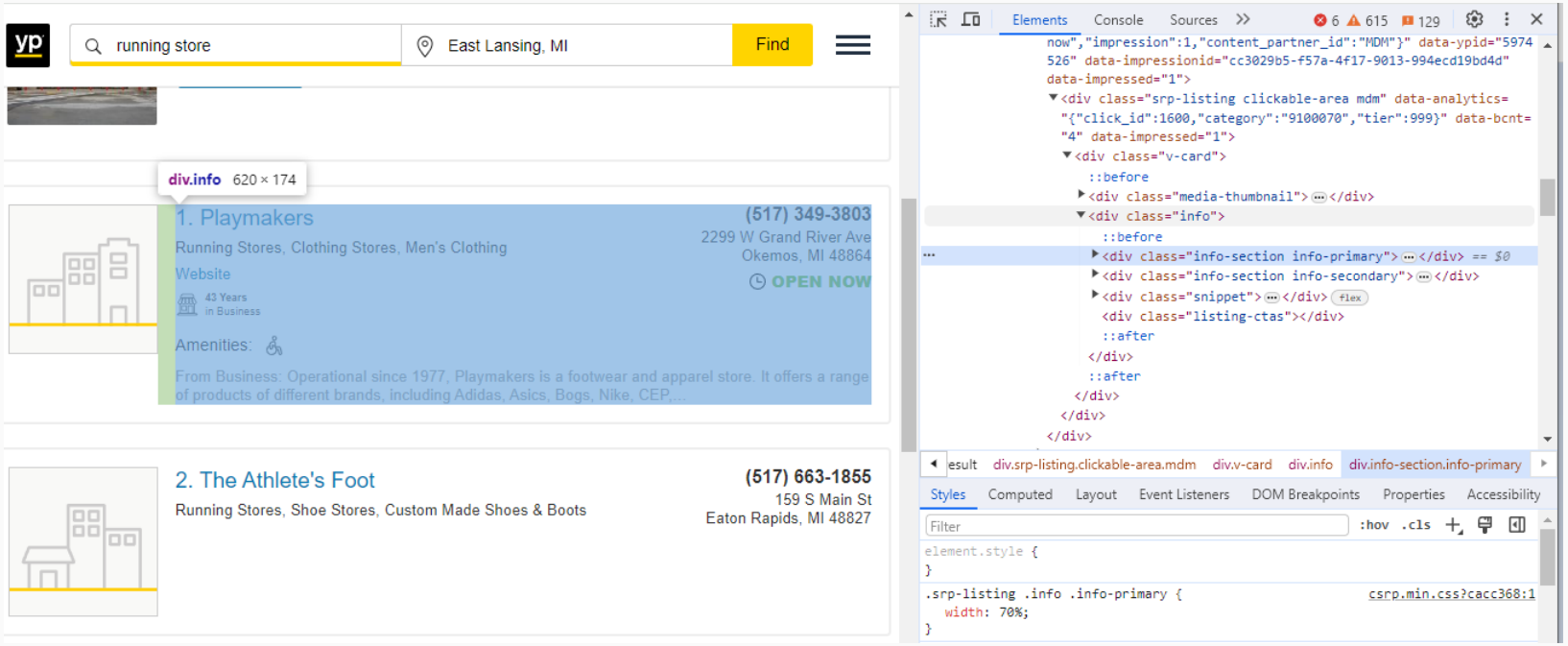
SelectorGadget vs. Source

Were you able to get it? I wasn't. ☹

This is an example of a case where we need to **inspect the source** to find the selector. Give it a try now!

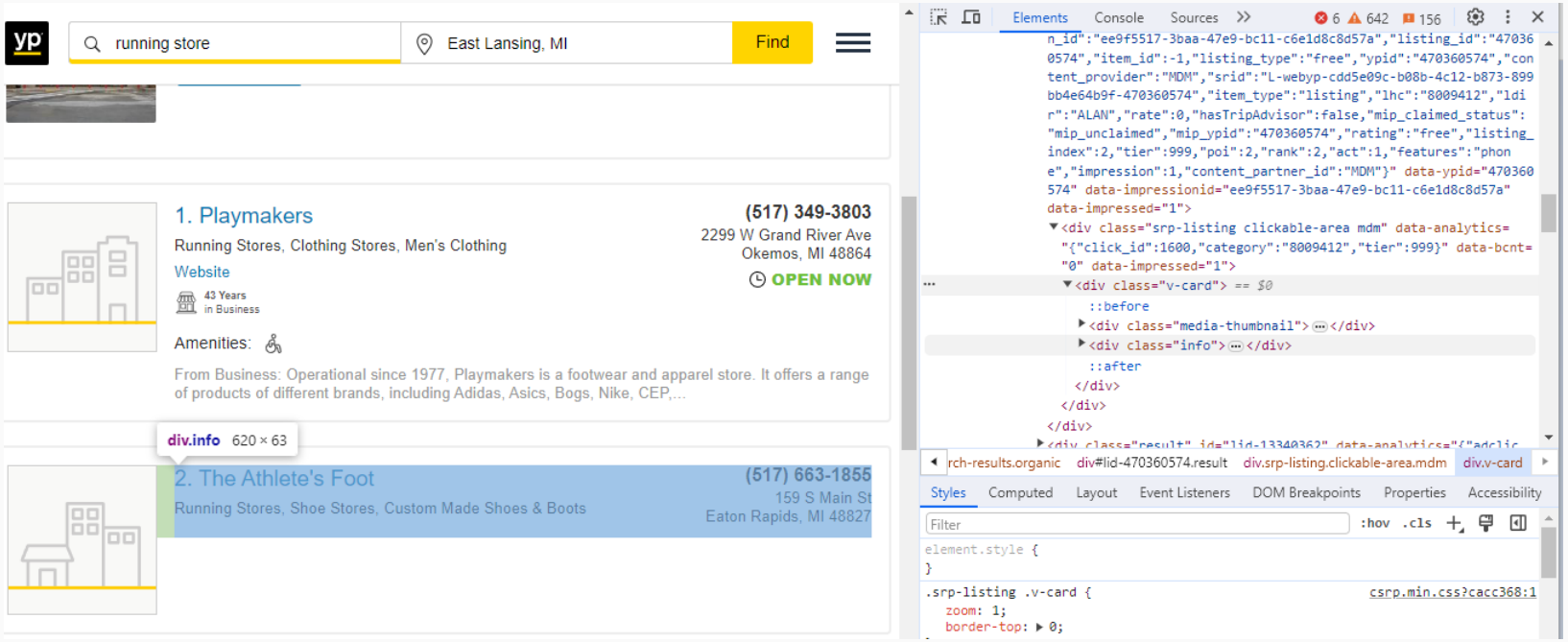
SelectorGadget vs. Source

After a little digging you'll find the selector to be `"div.info"`



SelectorGadget vs. Source

One hiccup: this selector isn't unique!



Duplicate Elements with

Solution: use `html_elements()` to retrieve **all matching elements at once**

```
page1 ← read_html("https://www.yellowpages.com/east-lansing-mi/running-s
  html_elements("div.info")
page1
```

```
## {xml_nodeset (7)}
## [1] <div class="info">\n<div class="info-section info-primary">\n<h2 class="
## [2] <div class="info">\n<div class="info-section info-primary">\n<h2 class="
## [3] <div class="info">\n<div class="info-section info-primary">\n<h2 class="
## [4] <div class="info">\n<div class="info-section info-primary">\n<h2 class="
## [5] <div class="info">\n<div class="info-section info-primary">\n<h2 class="
## [6] <div class="info">\n<div class="info-section info-primary">\n<h2 class="
## [7] <div class="info">\n<div class="info-section info-primary">\n<h2 class="
```

Application 2: Yellow Pages

We got back a **list** with 7 elements, one for each of the stores listed on the first page of search results.

What does the **Playmakers element** (first list object) contain?

```
page1[[1]]
```

```
## {html_node}
## <div class="info">
## [1] <div class="info-section info-primary">\n<h2 class="n">1. <a class="busi
## [2] <div class="info-section info-secondary">\n<div class="phones phone prim
## [3] <div class="snippet"><p class="body"><span>From Business: Operational si
## [4] <div class="listing-ctas"></div>
```

Application 2: Yellow Pages

Turns out, **more elements!**

1. `info-section info-primary`
2. `info-section info-secondary`
3. `snippet`
4. `listing-ctas`

This makes sense based on the source code:

Application 2: Yellow Pages

Let's start by retrieving the primary info as a table

- Start with section element type and a period (`"div."`)
- Replace spaces in class with periods
- Parse as table with `html_table()`

```
inf_prim ← page1[[1]] %>% # start with first listing box
  html_element("div.info-section.info-primary") %>% # grab primary info
  html_table()

inf_prim
```

```
## # A tibble: 0 × 0
```

Application 2: Yellow Pages

Oops, that doesn't work here. Turns out our primary info box contains more divisions/elements!

```
inf_prim ← page1[[1]] %>% # start with first listing box
  html_element("div.info-section.info-primary")

inf_prim
```

```
## {html_node}
## <div class="info-section info-primary">
## [1] <h2 class="n">1. <a class="business-name" href="/okemos-mi/mip/playmaker
## [2] <div class="categories">\n<a href="https://www.yellowpages.com/east-lans
## [3] <div class="ratings" data-israteable="true"></div>\n
## [4] <div class="links"><a class="track-visit-website" href="http://www.playm
## [5] <div class="badges"><div class="years-in-business">\n<img src="//i3.ypcd
## [6] <div class="amenities">\n<span>Amenities:</span><span class="amenities-i
```

Application 2: Yellow Pages

Let's try retrieving just the **business name** element

```
name ← page1[[1]] %>% # start with first listing box
  html_element("div.info-section.info-primary") %>% # get primary info di
  html_element("a.business-name")

name
```

```
## {html_node}
## <a class="business-name" href="/okemos-mi/mip/playmakers-5974526" data-analy
## [1] <span>Playmakers</span>
```

Application 2: Yellow Pages

Here we've got an `<a>` class, which is the HTML tag for a **hyperlink**.

This means we could do one of two things:

1. Retrieve the Text (Business Name)

2. Retrieve the Link (Business Page)

Retrieve Text with *html_text()*

1. Retrieve the Text (Business Name):

- `html_text()` retrieves the **raw text**
- `html_text2()` retrieves text **as it appears online**
  - Here there's no difference

```
name %>% html_text() # get raw text portion of element
```

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

```
name %>% html_text2() # get text as it appears online
```

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

Retrieve Link with *html_attr()*

2. Retrieve the Link (Business Page):

- `html_attr()` gets an attribute with a particular name (here the hyperlink, or "href")

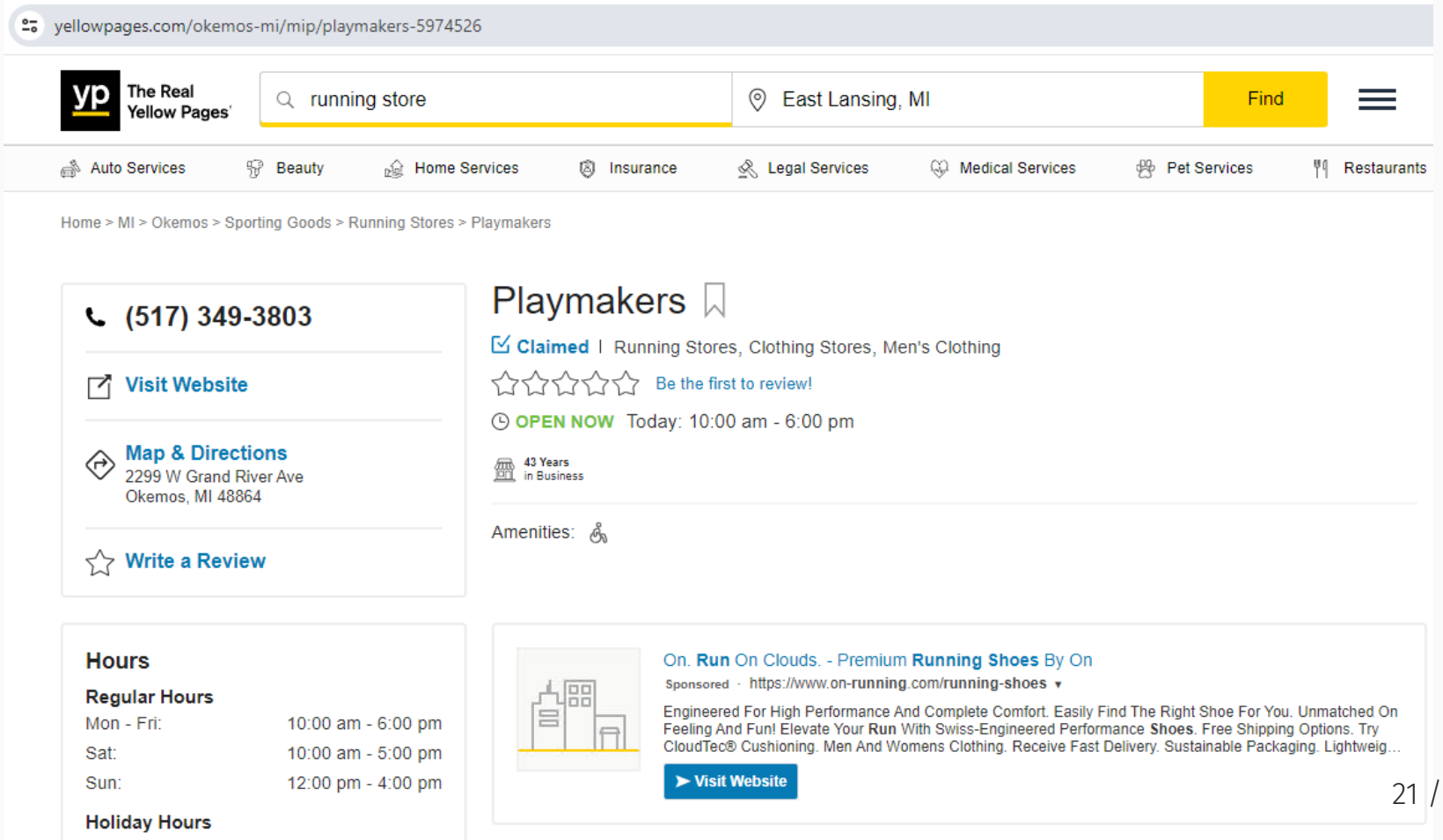
```
name %>%
  html_attr("href") # get link portion of element (link after "href =")
```

```
## [1] "/okemos-mi/mip/playmakers-5974526"
```

This gets us the portion of the url after `yellowpages.com` for the business page.

Application 2: Yellow Pages

Next, let's switch over to the store page which contains a lot more detailed info.



Application 2: Yellow Pages

Let's start by grabbing the **categories**.

- Retrieve the page using the url we just grabbed
- Grab the division we want ("categories")
- Parse as text
- Split on comma/space, trim extra white space

```
link ← paste0("https://yellowpages.com/", name %>% html_attr("href"))
pm ← read_html(link)
pm %>% html_element("div.categories") %>%
  html_text() %>%
  str_split(", ") %>%
  str_squish()
```

```
## [1] "c(\"Running Stores\", \"Clothing Stores\", \"Men's Clothing \")"
```

Application 2: Yellow Pages

Next let's get the "Other Information" contents.

```
pm %>% html_element("dd.other-information") %>%
  html_text()
```

```
## [1] "Parking: Lot, FreeWheelchair Accessible: Yes"
```

Application 2: Yellow Pages

Using `html_text()` gets us non-breaking spaces (` `).

Switching to `html_text2()` looks more like we expect:

```
pm %>% html_element("dd.other-information") %>%
  html_text2()
```

```
## [1] "Parking: Lot, Free\n\nWheelchair Accessible: Yes"
```

Where `\n` is the HTML for line break

Challenge

Challenge: when is Playmakers open?

- Use the page source to find the selector for the contents of the "Hours" box (Regular Hours through Holiday Hours)
- Format it as a table with two variables:
  1. "Days"
  2. "Business Hours"
- Remove the colon from regular hours days

Challenge

Challenge: when is Playmakers open?

- Use the page source to find the selector for the contents of the "Hours" box (Regular Hours through Holiday Hours)
- Format it as a table with two variables:
  1. "Days"
  2. "Business Hours"
- Remove the colon from regular hours days

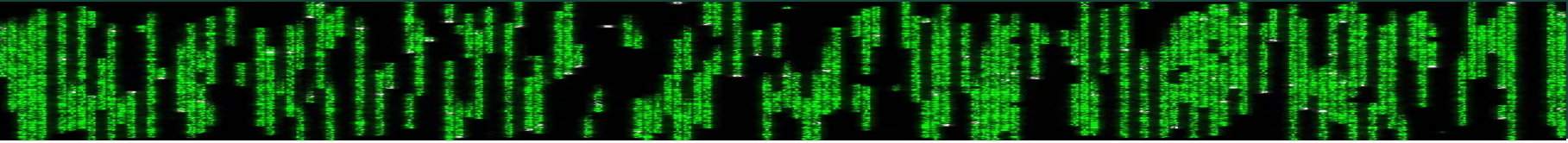
```
hours ← pm %>% html_element("dd.open-hours")
hours_df ← hours %>% html_table()
colnames(hours_df) ← c("Days", "Business Hours")

hours_df ← dplyr::mutate(hours_df,
                  Days = str_replace(Days, ":$", ""))
```

Summary: Static Scraping

When scraping static web content rendered server-side:

- Start by finding the relevant CSS selectors
- Use the **rvest** package to read in the HTML document and parse it
  - .hi-medgrn[Tabular workflow:] `read_html(URL) %>% html_elements(CSS_Selectors) %>% html_table()`
  - Might need other functions depending on content type (e.g. `html_text/text2(), html_attr(), html_children()`)

Interacting with Static Sites



While scraping a **known url** is pretty cool, we can do more than that.

We already saw one way to progress through a url chain, but we can **go deeper**.

We can go **into.the.matrix**.

Interacting with Static Sites

Let's go back to the beginning with our Yellow Pages workflow.

Suppose instead you wanted to

1. Enter a search term (i.e. "running store") and execute a search from the home page
2. Retrieve all business URLs from the first page
3. Navigate to the next page and repeat the process

We can do this with **rvest**.

Interacting with Web Forms

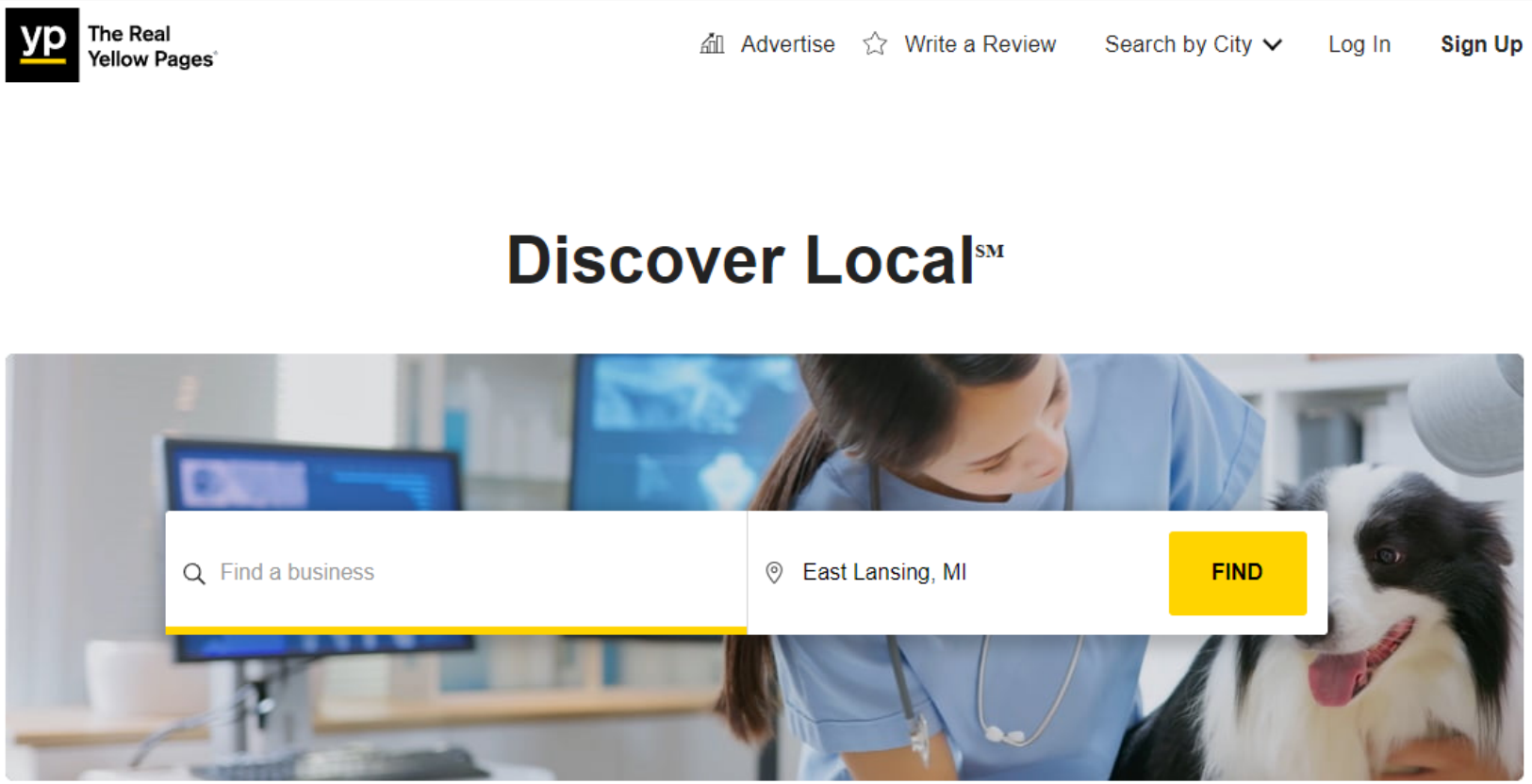
To **interact with web forms**, we'll use the below workflow:

- Retrieve a target form with `html_form()`
- Fill in the form's required elements with `html_form_set()`
- Submit the form with `html_form_submit()`
- Read the resulting url with `html_read()`.

Let's practice this.

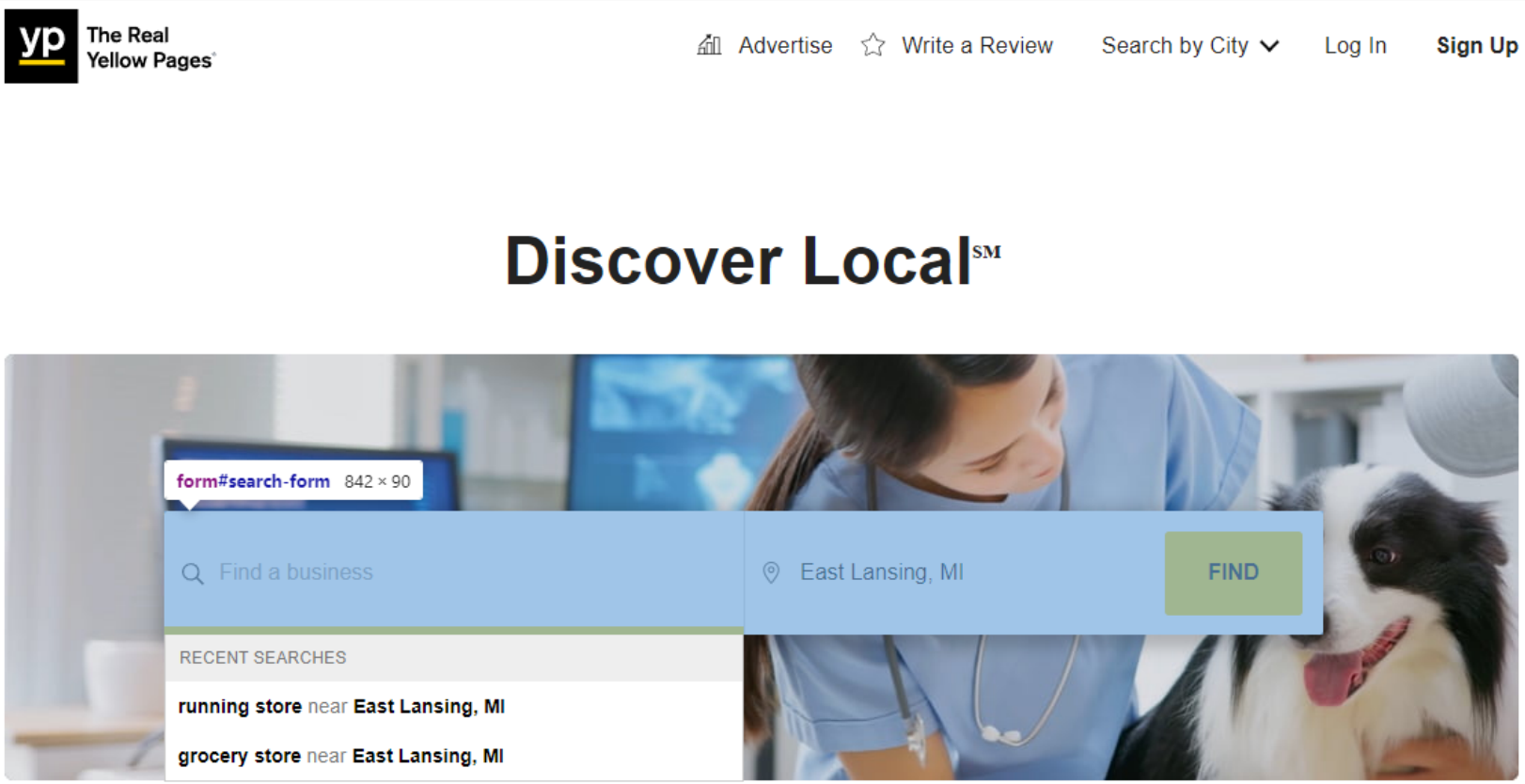
Interacting with Web Forms

Start by navigating to **Yellowpages.com** (the main page).



Interacting with Web Forms

Conveniently, there's just one form here, `search-form`



Interacting with Web Forms

Use `html_form()` to extract the first/only form from the homepage:

```
yp ← read_html("https://www.yellowpages.com/")
# grab the search form
search ← html_form(yp)[[1]]
```

Interacting with Web Forms

Every web form has several different **fields** we'll need to fill or interact with

```
search
```

```
## <form> 'search-form' (GET https://www.yellowpages.com/search)
##   <field> (text) search_terms:
##   <field> (text) geo_location_terms: Los Angeles, CA
##   <field> (button) : Find
```

Here we've got:

- 2 **text fields:** "search_terms" and "geo_location_terms" that need contents
- 1 **button** named "find" we'd click when manually searching

Filling Web Forms with *html_form_set()*

Let's set up our search for "running store" in Grand Rapids, MI

```
search_set ← html_form_set(search,
                           search_terms = "Running Store",
                           geo_location_terms = "Grand Rapids, MI")
```

Submitting Web Forms

Submitting the form and getting the response:

```
resp ← html_form_submit(search_set) # submit the form
resp
```

```
## Response [https://www.yellowpages.com/search?search_terms=Running%20Store&ge
##   Date: 2024-02-19 21:05
##   Status: 200
##   Content-Type: text/html; charset=utf-8
##   Size: 135 kB
## <!DOCTYPE html><html lang="en"><head><script async src="https://www.googleta
##
## function gtag(){dataLayer.push(arguments);}
##
## gtag('js', new Date());
## gtag('config', 'G-0EQTJQH34W');
##
## if (document.cookie.indexOf('optOut') ≠ -1) {
##     gtag('set', 'allow_google_signals', 'false');
##     gtag('set', 'ad_storage', 'denied');
```

Web Form Response

Submitting the form gets us a `response` class object, containing some useful components, including the resulting url:

```
resp$url
```

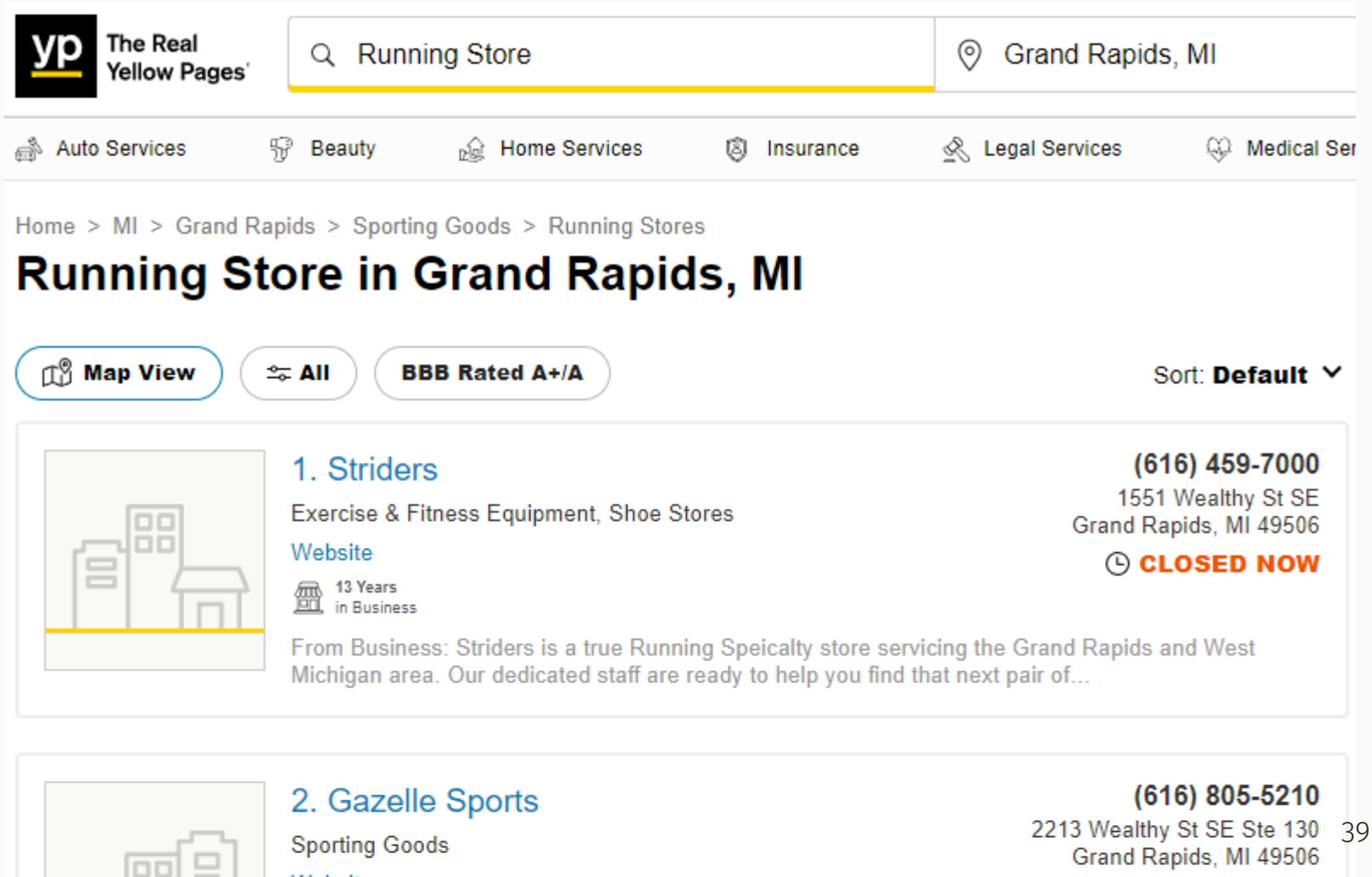
```
## [1] "https://www.yellowpages.com/search?search_terms=Running%20Store&geo_loc
```

And we can read the response url as we usually do

```
# get the result url's html
page1 ← read_html(resp)
```

Web Form Response

And we're now back to the standard search results first page:



Static Scraping

We could now combine what we've learned to

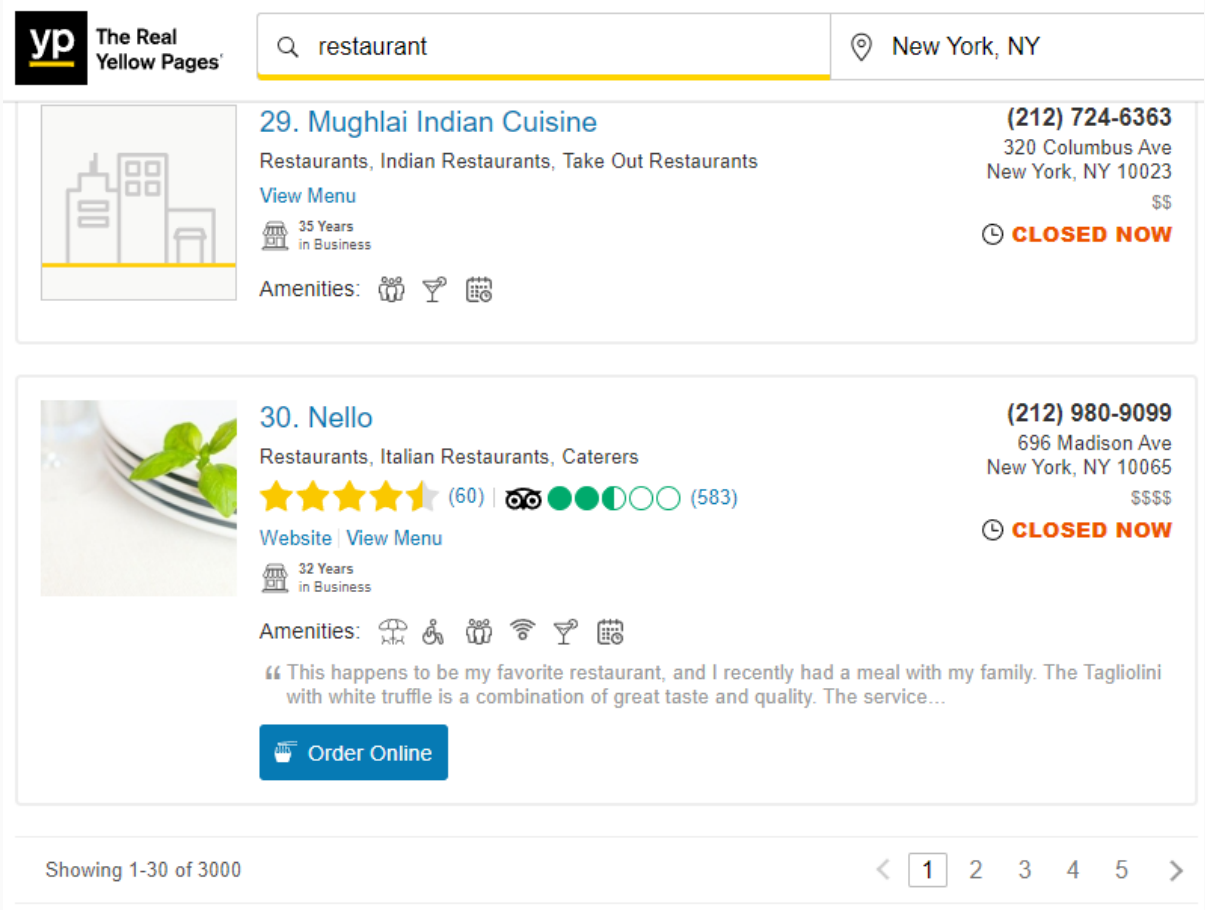
1. Scrape info for each business on the page
2. Scrape urls for each of the business pages
3. Iteratively navigate to each business page and scrape more detailed info

We'll revisit this idea when we talk about **writing custom functions**, but for now...

Let's chat about navigating pages.

Navigating Search Pages

First we'll need a search/geography with a *lot* of results: **"restaurant" in New York, NY**



Navigating Search Pages

Let's take a look at the page navigation menu at the bottom.

The main element is an **unordered list** `ul`

With each **list element** denoted by `li`

And each element containing a **hyperlink** `a` to that page

Navigating Search Pages

Let's retrieve the search page url.

```
ny_rest ← read_html("https://www.yellowpages.com/search?search_terms=res
```

How many unordered lists are on the page?

```
ny_rest %>% html_elements("ul") %>% length()
```

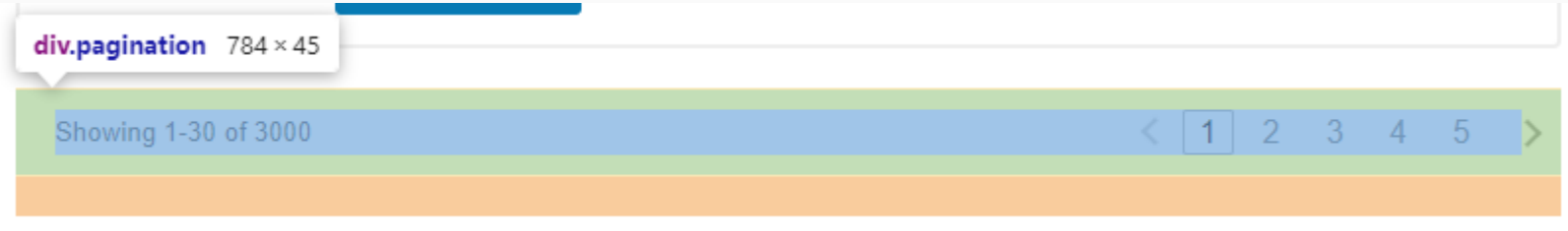
```
## [1] 9
```

Okay, so not just the one that we want.

Navigating Search Pages

While we could look at all the unordered list elements' contents and figure out which is the one we want, a more effective way is to take advantage of **parent/child relationships and nesting of elements.**

If we explore the source code, we can learn that the `ul` element we want is contained within the `div.pagination` element:



Navigating Search Pages

As a result, stringing together two element selection steps will get us what we want:

```
# checking to see if "div.pagination" is unique
ny_rest %>% html_elements("div.pagination") %>% length()
```

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

```
# and the unorder list within that container:
ny_rest %>% html_elements("div.pagination") %>% html_elements("ul") %>% le
```

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

```
# now retrieve the page list
pages ← ny_rest %>%
  html_elements("div.pagination") %>%
  html_elements("ul")
```

Navigating Search Pages

We can then use the **nth child selector** to figure out which of the buttons has the links we want:

```
pages %>% html_element(":nth-child(1)")
```

```
## {xml_nodeset (1)}
## [1] <li><div class="prev"></div></li>\n
```

First child (`:nth-child(1)`) is the "back" arrow

- Link currently disabled since we're on the first page

Navigating Search Pages

We can then use the **nth child selector** to figure out which of the buttons has the links we want:

```
pages %>% html_element(":nth-child(2)")
```

```
## {xml_nodeset (1)}
## [1] <li><span class="disabled">1</span></li>\n
```

Second child (`:nth-child(2)`) is the first page (1)

- Link currently disabled since we're already on that page

Navigating Search Pages

We can then use the **nth child selector** to figure out which of the buttons has the links we want:

```
pages %>% html_element(":nth-child(3)")
```

```
## {xml_nodeset (1)}
## [1] <li><a href="/search?search_terms=restaurant&amp;geo_location_terms=New%
```

Third child (`:nth-child(3)`) is the next page (2)

- Link active!

Navigating Search Pages

Retrieving the attributes for the second search page link:

```
pages %>%
  html_element(":nth-child(3)") %>%
  html_element("a") %>%
  html_attrs()
```

```
## [[1]]
##                                                                                 hr
## "/search?search_terms=restaurant&geo_location_terms=New%20York%2C%20NY&page=
##                                                                               data-pa
##                                                                                    "
##                                                                          data-analyti
##                              "{\"click_id\":132,\"module\":1,\"listing_page\":2
##                                                                          data-remo
##                                                                               "tru
```

Navigating Search Pages

And scraping the second page url:

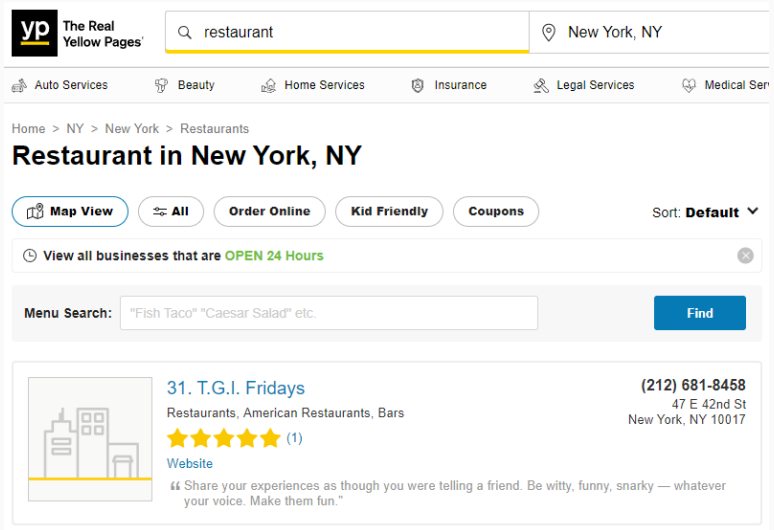
```
page2_url ← pages %>%
  html_element(":nth-child(3)") %>%
  html_element("a") %>%
  html_attr("href")

page2 ← read_html(paste0("https://www.yellowpages.com", page2_url))
```

Navigating Search Pages

Checking to make sure we made it to page 2:

```
page2 %>%
  # grab first business name
  html_element("a.business-name") 
  html_text()
```

```
## [1] "T.G.I. Fridays"
```



Navigation by URL Patterns

Sometimes we can **identify shortcuts** to our scraping process - in this case, that's with a **url pattern**.

Let's look closer at the search response url and the page 2 url:

```
resp$url
```

```
## [1] "https://www.yellowpages.com/search?search_terms=Running%20Store&geo_loc
```

```
page2_url
```

```
## [1] "/search?search_terms=restaurant&geo_location_terms=New%20York%2C%20NY&p
```

Navigation by URL Patterns

First let's decompose the search url (leaving out the yellowpages.com part):

`/search?`
`search_terms=` **`Running%20Store`** `&geo_location_terms=` **`Grand%20Rapids%2C%20MI`**

This reveals a pattern: if we know the search term and geography we want to search, we can **bypass the search form entirely.**

- `%20` the ASCII symbol for space
- `%2C` the ASCII symbol for comma

For example, Indian restaurants in Ann Arbor:

`search?`
`search_terms=` **`Indian%20Restaurant`** `&geo_location_terms=` **`Ann%20Arbor%2C%20MI`**

Navigation by URL Patterns

A similar pattern exists in the search page url:

```
"/search?
search_terms=restaurant&geo_location_terms=New%20York%2C%20NY&page=2
```

This means that we can go directly to a given page of search results without ever manually searching!

Often once we start interacting with a website we'll find shortcuts like these that will greatly speed up an automated scraping routine.

Scraping Dynamic Sites

Note that **rvest** has an experimental set of features built around `read_html_live()` and the `LiveHTML` object that does the same thing

- Requires Google Chrome is installed on your machine
- Uses the `chromote` package to run a live browser in the background

Table of Contents