

Big Data and Economics

Lecture 6a: Web Data in Research

Kyle Coombs (he/him/his)
Bates College | [EC/DCS 368](#)

Table of contents

1. Prologue
2. Worldwide Web of Data
3. Examples of scraping in economics research
4. Access methods
 - Click and Download
 - Server-side scraping
 - Client-side scraping
5. Ethics of web scraping

Prologue

Prologue

- We've spent the first month of this class on learning:
 - empirical organization skills ("Clean Code"),
 - basics of R
 - basics of data wrangling and tidy data
- Now we're going to tackle data acquisition via **scraping**
- Essentially, we're going to learn how to get data from the web
- As context, everything I am showing you today assumes you've:
 1. Found data on the web you want
 2. Found the relevant way to access it (APIs vs. CSS)
 3. Know the specifics needed to access the data (e.g. the name of a series, have an API key, the rough HTML structure)
- These data are usually messy in one way or another, so it'll give you something to tidy
- Extended demos for this lecture are available in [WEB APIs](#) and [web Scraping](#)

Plan for today

- What is scraping?
- Contrast Client-side and Server-side scraping
- Examples of scraping in economics research
- Ethical considerations
- Learn by doing with APIs (CSS will happen later -- potentially end of semester)

Attribution

- These slides take inspiration from the following sources:
 - [Nathan Schiff's web data lecture](#)
 - [Andrew MacDonald's slides](#)
 - [Jenny Bryan's textbook](#)
 - [Grant McDermott's notes on CSS and APIs](#)
 - [James Densmore's stance on ethics](#)

Worldwide Web of Data

Worldwide Web of Data

- Every website you visit is packed with data
- Every app on your phone is packed with data and taking data from you
- Guess what?
 - These data often measure hard to measure things
 - These data are often public (at some level of aggregation/anonymity)
 - These data are often not easily accessible and not **tidy**
 - Samples might be biased (have to navigate that)
 - This is legal (usually) and ethical (usually)
- Guess what? All this makes these data (and knowing how to access it) valuable
 - It also makes this a hard skill to pick up

Examples of scraping in economics research

What cool things can you do with web

- Can anyone think of examples of web data being used in economics research?

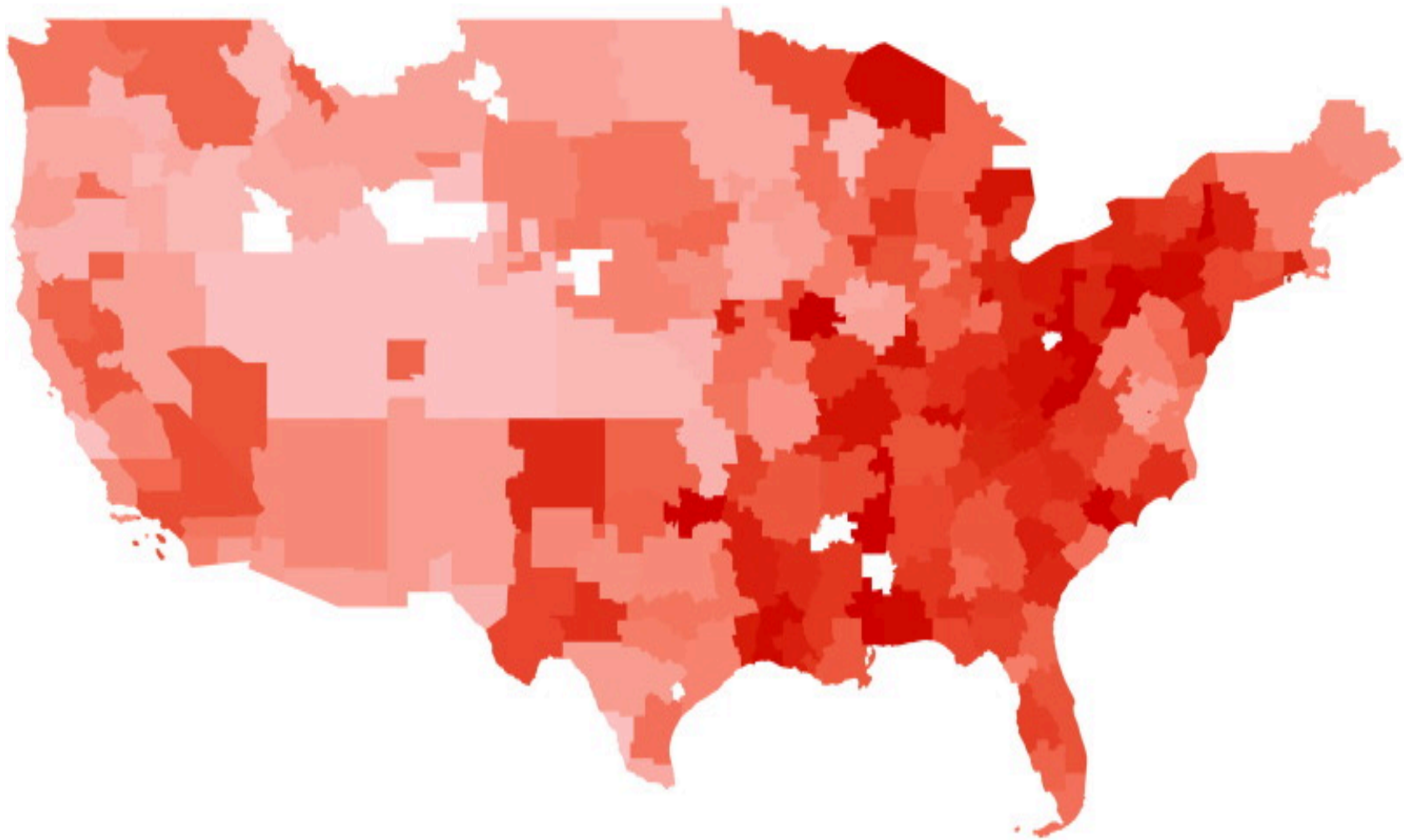
Measuring hard to measure things

- Imagine you survey a ton of people about their beliefs that a candidate is unfit to be president because of their race
- Due to social desirability bias, you get a lot of "I don't know" or "I don't think that"
- There are lots of creative survey methods to get at this, but is there some way to measure this without asking people?
- Say, why not find out the frequency that people search Google for racial epithets in connection to the candidate?
- Guess what? Stephens-Davidowitz (2014) did just that
 - Finds racial animus cost Barack Obama 4 percentage points in the 2008 election (equivalent of a home-state advantage)
 - Google search term data yield effects that are 1.5 to 3 times larger than survey estimates of racial animus

$$\text{Racially Charged Search Rate}_j = \left[\frac{\text{Google searches including the word "Word 1 (s)"} }{\text{Total Google searches}} \right]_{j, 2004-2007}$$

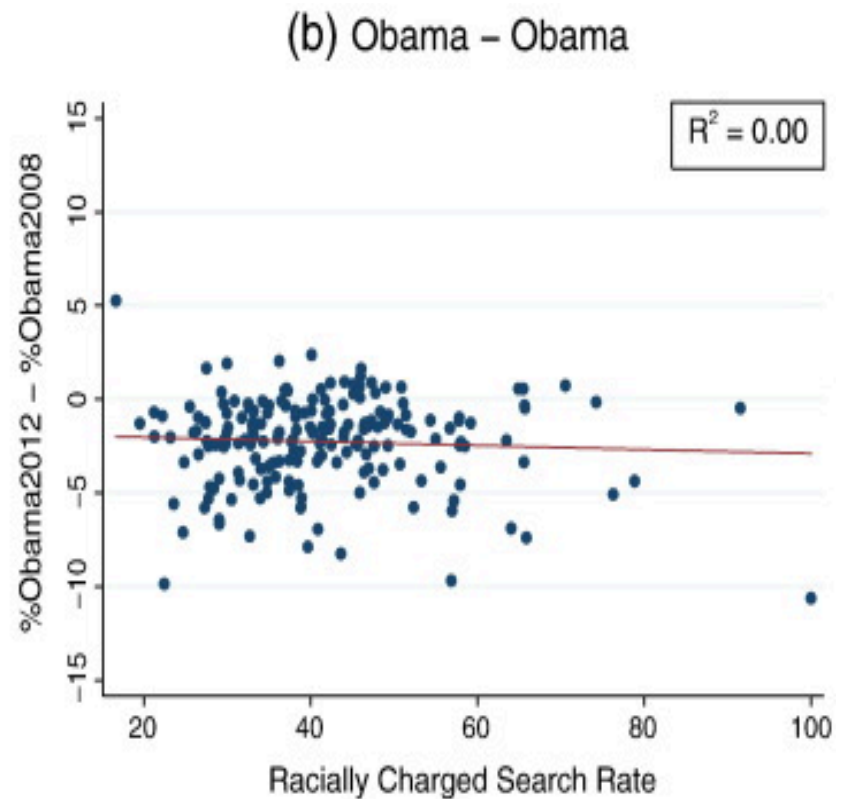
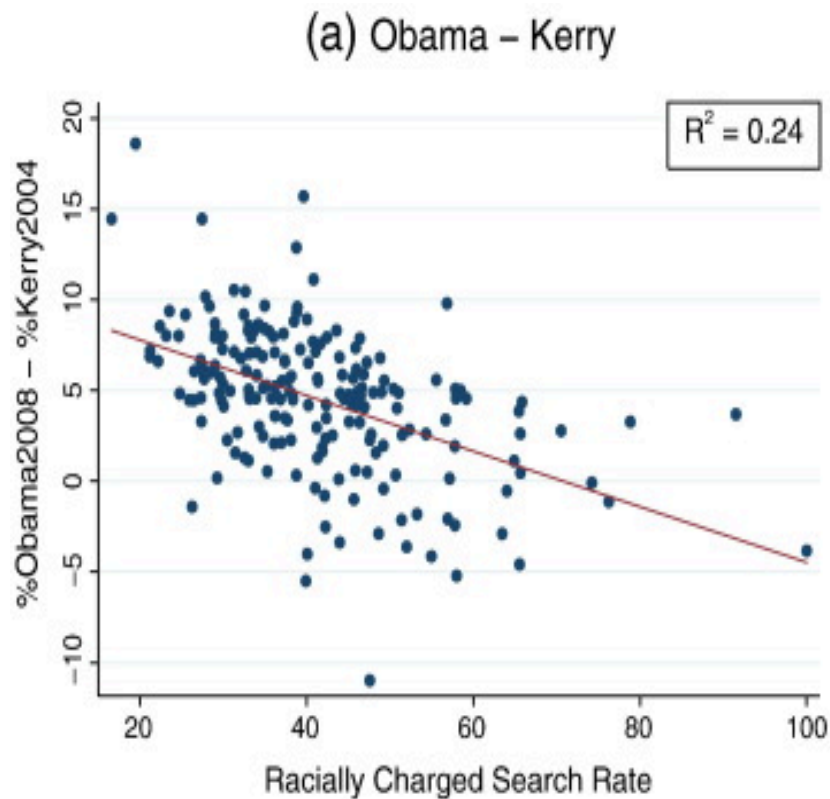
for j geographical area (state, county, etc.)

Racial Animus Map



Map of media markets by racially charged search rate from 2004 to 2007. The darker red, the more racially charged.

Election performance



Obama underperformed Kerry in areas with more racially charged search rates.

Other uses

- "Billion prices project" (Cavallo and Rigobon 2015) : collect prices from online retailers to look at macro price changes
- Davis and Dingell (2016): use Yelp to explore racial segregation in consumption
- Halket and Pginatti (2015): scrape Craigslist to look at housing markets
- Wu (2018): undergraduate hacked into online economics job market forum to look at toxic language and biases in the academic economics against women
- Glaeser (2018) uses Yelp data to quantify how neighborhood business activity changes as areas gentrify (**Student presentation**)
- Tons leverage eBay, Alibaba, etc. to look at all kinds of commercial activity
- Edelman B (2012) gives an overview of using internet data for economic research

Access methods

Access methods

There are three ways to data off the web:

1. **click-and-download** on the internet as a "flat" file, like a CSV or Excel file
 - What you're used to
 2. **Client-side** *websites contain an empty template that _request* data from a server and then fills in the template with the data
 - The request is sent to an API (application programming interface) endpoint
 - Technically you can just source right from the API endpoint (if you can find it) and skip the website altogether
 - I consider this a form of scraping
 - **Key concepts:** APIs, API endpoints
 3. **Server-side** websites that sends HTML and JavaScript to your browser, which then renders the page
 - People often call this "scraping"
 - All the data is there, but not in a tidy format
 - **Key concepts:** CSS, Xpath, HTML
- Key takeaway: if there's a structure to how the data is presented, you can exploit it to get the data

Click and Download

- You've all seen this approach before
- You go to a website, click a link, and download a file
- Sometimes you need to login first, but if not you can automate this with R's `download.file()` function
- Below will download the Occupational Employment and Wage Statistics (OEWS) data for Massachusetts in 2021 from the BLS

```
download.file("https://www.bls.gov/oes/special.requests/oesm21ma.zip", "oesm21ma.zip")
```

Client-side scraping

- The website contains an empty template of HTML and CSS.
 - E.g. It might contain a “skeleton” table without any values.
- However, when we actually visit the page URL, our browser sends a request to the host server.
- If everything is okay (e.g. our request is valid), then the server sends a response script, which our browser executes and uses to populate the HTML template with the specific information that we want.
- **Webscraping challenges:** Finding the “API endpoints” can be tricky, since these are sometimes hidden from view.
- **Key concepts:** APIs, API endpoints

APIs

- APIs are a collection of rules/methods that allow one software application to interact with another
- Examples include:
 - Web servers and web browsers
 - R libraries and R clients
 - Databases and R clients
 - Git and GitHub and so on

Key API concepts

- **Server:** A powerful computer that runs an API.
- **Client:** A program that exchanges data with a server through an API.
- **Protocol:** The “etiquette” underlying how computers talk to each other (e.g. HTTP).
- **Methods:** The “verbs” that clients use to talk with a server. The main one that we’ll be using is GET (i.e. ask a server to retrieve information), but other common methods are POST, PUT and DELETE.
- **Requests:** What the client asks of the server (see Methods above).
- **Response:** The server’s response. This includes a Status Code (e.g. “404” if not found, or “200” if successful), a Header (i.e. meta-information about the response), and a Body (i.e. the actual content that we’re interested in).
- Not covered? Explicit directions for each API we cover today
- Instead, we're covering the nuts and bolts so you can figure out how to use any API

API Endpoints

- Web APIs have a URL called an **API Endpoint** that you can use to access view the data in your web browser
- Except instead of rendering a beautifully-formatted webpage, the server sends back a ton of messy text!
 - Either a JSON (JavaScript object notation) or XML (eXtensible Markup Language) file
- It'd be pretty overwhelming to learn how to navigate these new language syntaxes
- Guess what? R has packages to help you with that
 - `jsonlite` for JSON
 - `xml2` for XML
- Today we're going to work through a few of these
- That means the hardest parts are:
 - Finding the API endpoint
 - Understanding the rules
 - Identify the words you need to use to get the data you want
- To be clear, that's all still tricky!

Easy API: New York City Open Data

- NYC OpenData provides a ton of data on New York City
- It offers it as something easy to download
- It also offers it as

Trees of NYC

[←](#) [→](#) [↺](#) [https://data.cityofnewyork.us/Environment/2015-Street-Tree-Census-Tree-Data/uvpi-gqnh](#)

[Home](#) [Data](#) [About ▾](#) [Learn ▾](#) [Alerts](#) [Contact Us](#) [Blog](#) [🔍](#) [Sign In](#)

2015 Street Tree Census - Tree Data

[Environment](#) [View Data](#) [Visualize ▾](#) [Export](#) [API](#) [...](#)

Street tree data from the TreesCount! 2015 Street Tree Census, conducted by volunteers and staff organized by NYC Parks & Recreation and partner organizations. Tree data collected includes tree species, diameter and perception of health. Accompanying blockface data is available indicating status of data collection and data release citywide.

Updated
September 13, 2018

Data Provided by
Department of Parks and Recreation (DPR)

About this Dataset

Updated
September 13, 2018

Data Last Updated	October 4, 2017
Metadata Last Updated	September 13, 2018
Date Created	June 3, 2016

Update

Update Frequency	Historical Data
Automation	No
Date Made Public	6/3/2016

Read the trees JSON

```
# library(jsonlite) ## Already loaded
nyc_trees =
  fromJSON("https://data.cityofnewyork.us/resource/nwx-4ae8.json") %>%
  as_tibble()
nyc_trees
```

```
## # A tibble: 1,000 × 45
```

```
##   tree_id block_id created_at      tree_dbh stump_diam curb_loc status health
##   <chr>   <chr>   <chr>         <chr>    <chr>        <chr>   <chr> <chr>
## 1 180683 348711 2015-08-27T00:00... 3         0         OnCurb  Alive Fair
## 2 200540 315986 2015-09-03T00:00... 21        0         OnCurb  Alive Fair
## 3 204026 218365 2015-09-05T00:00... 3         0         OnCurb  Alive Good
## 4 204337 217969 2015-09-05T00:00... 10        0         OnCurb  Alive Good
## 5 189565 223043 2015-08-30T00:00... 21        0         OnCurb  Alive Good
## 6 190422 106099 2015-08-30T00:00... 11        0         OnCurb  Alive Good
## 7 190426 106099 2015-08-30T00:00... 11        0         OnCurb  Alive Good
## 8 208649 103940 2015-09-07T00:00... 9         0         OnCurb  Alive Good
## 9 209610 407443 2015-09-08T00:00... 6         0         OnCurb  Alive Good
## 10 192755 207508 2015-08-31T00:00... 21        0         OffsetF... Alive Fair
```

```
## # i 990 more rows
```

```
## # i 37 more variables: spc_latin <chr>, spc_common <chr>, steward <chr>,
## #   guards <chr>, sidewalk <chr>, user_type <chr>, problems <chr>,
## #   root_stone <chr>, root_grate <chr>, root_other <chr>, trunk_wire <chr>,
## #   trnk_light <chr>, trnk_other <chr>, brch_light <chr>, brch_shoe <chr>,
## #   brch_other <chr>, address <chr>, zipcode <chr>, zip_city <chr>,
## #   cb_num <chr>, borocode <chr>, boroname <chr>, cnclldist <chr>, ...
```

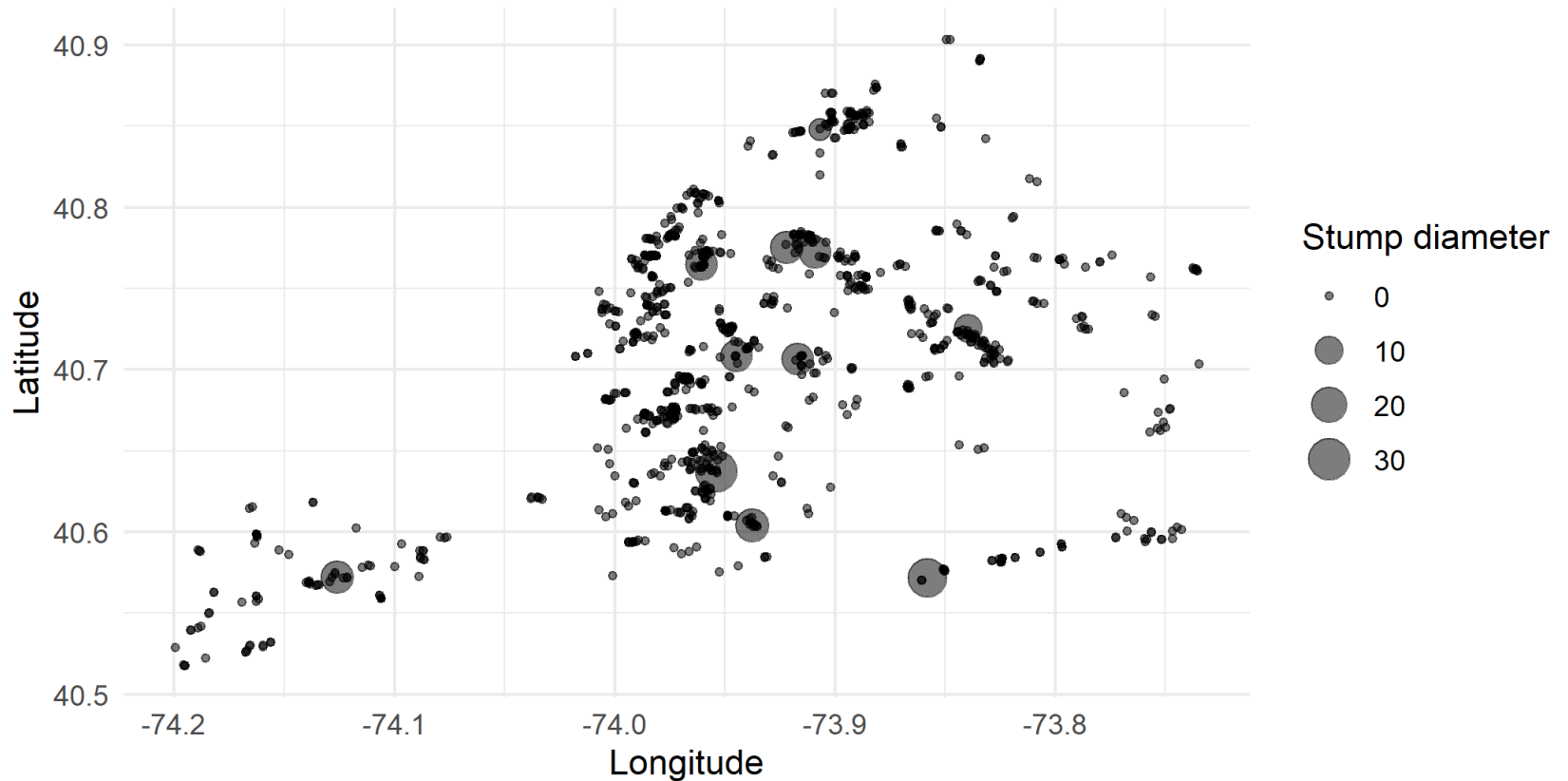

"Map" the trees

```
nyc_trees %>%  
  select(longitude, latitude, stump_diam, spc_common, spc_latin, tree_id) %>%  
  mutate_at(vars(longitude:stump_diam), as.numeric) %>%  
  ggplot(aes(x=longitude, y=latitude, size=stump_diam)) +  
  geom_point(alpha=0.5) +  
  scale_size_continuous(name = "Stump diameter") +  
  labs(  
    x = "Longitude", y = "Latitude",  
    title = "Sample of New York City trees",  
    caption = "Source: NYC Open Data"  
  )
```

"Map" the trees

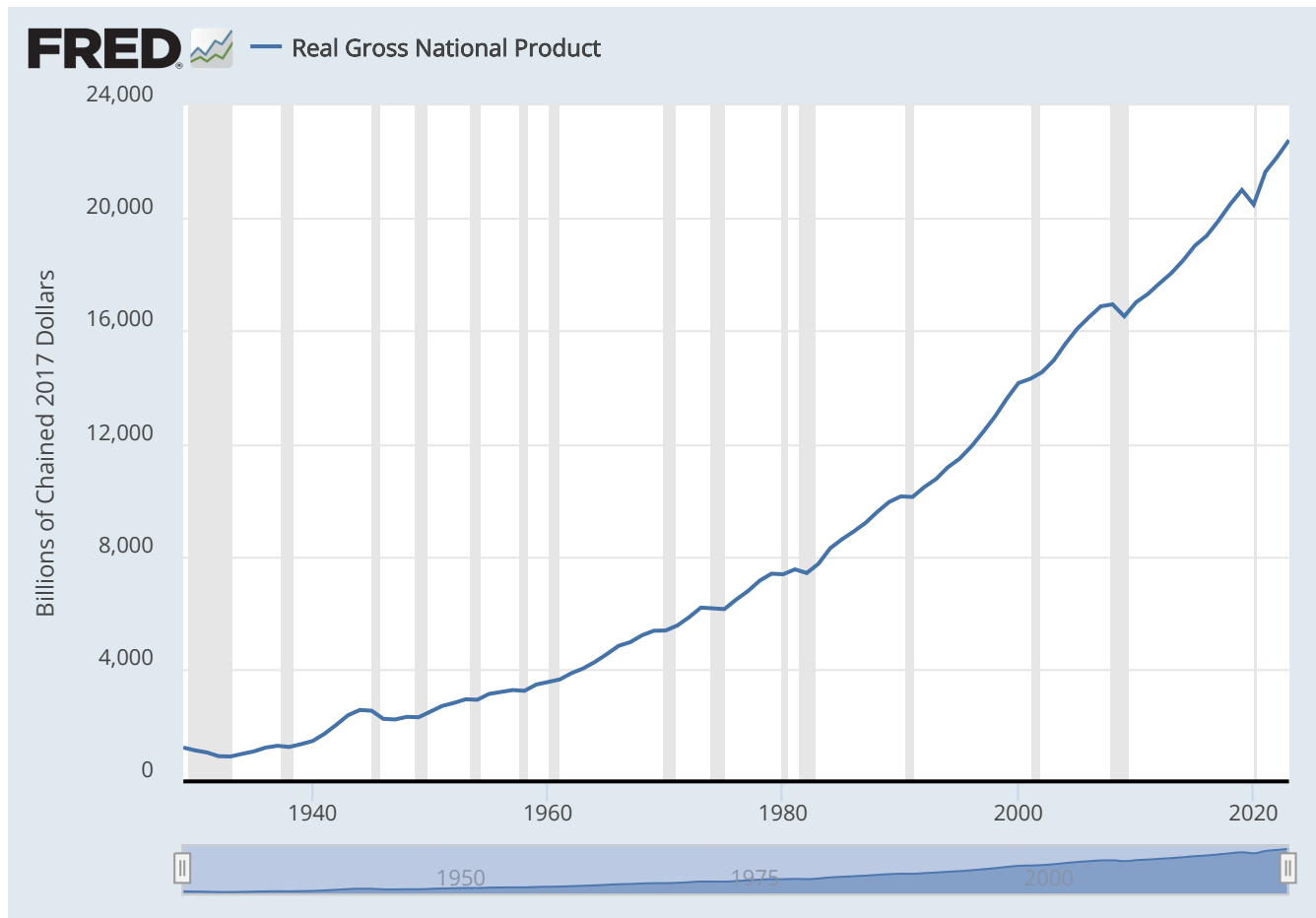
- Don't worry, better looking maps to come

Sample of New York City trees



Source: NYC Open Data

You've likely used FRED before



Source: U.S. Bureau of Economic Analysis

[Customize](#) | [Download Data](#) | [FRED - Economic Data from the St. Louis Fed](#)

Underneath is an API!

- The endpoint is https://api.stlouisfed.org/fred/series/observations?series_id=GNPCA&api_key=&file_type=json
- Just sub an your API key and you're good to go
- What's an API Key? It is a unique identifier that is used to authenticate access to the data
 - It's like a password, but it's not a password
 - It tracks who is using the API and how much they're using it
 - Fake example: `asdfjaw523a3523414at43sad`
 - FRED gives you one for free if you [register an API key](#)

```
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "observation_start": "1600-01-01", "observation_end": "2024-02-03", "series_id": "GNPCA", "file_type": "json"}
[{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1929-01-01", "value": "1202.659"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1930-01-01", "value": "1100.67"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1931-01-01", "value": "1029.038"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1932-01-01", "value": "895.802"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1933-01-01", "value": "883.847"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1934-01-01", "value": "978.188"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1935-01-01", "value": "1065.716"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1936-01-01", "value": "1201.443"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1937-01-01", "value": "1264.393"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1938-01-01", "value": "1222.966"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1939-01-01", "value": "1320.924"},
{"realtime_start": "2024-02-03", "realtime_end": "2024-02-03", "date": "1940-01-01", "value": "1435.656"}]
```

What do I need to know?

- The base URL: <https://api.stlouisfed.org/>
- The API endpoint (fred/series/observations/)
- The parameters:
 - series_id="GNPCA"
 - api_key=YOUR_API_KEY
 - file_type=json

```
endpoint = "fred/series/observations"
params = list(
    api_key= "YOUR_FRED_KEY", ## Change to your own key
    file_type="json",
    series_id="GNPCA"
)
```

Reading FRED's JSON

```
fred =  
  http::GET(  
    url = "https://api.stlouisfed.org/", ## Base URL  
    path = endpoint,      ## The API endpoint  
    query = params        ## Our parameter list  
  ) %>%  
  http::content(as="text") %>%  
  jsonlite::fromJSON()
```

- What's in there?

```
fred
```

```
## $realtime_start  
## [1] "2024-09-26"  
##  
## $realtime_end  
## [1] "2024-09-26"  
##  
## $observation_start  
## [1] "1600-01-01"  
##  
## $observation_end  
## [1] "9999-12-31"
```

Turn it into data

```
fred =  
  fred %>%  
  purrr::pluck("observations") %>% ## Extract the "$observations" list element  
  # .$observations %>% ## I could also have used this  
  # magrittr::extract("observations") %>% ## Or this  
  as_tibble() ## Just for nice formatting  
fred
```

```
## # A tibble: 95 × 4  
##   realtime_start realtime_end date      value  
##   <chr>          <chr>      <chr>    <chr>  
## 1 2024-09-26    2024-09-26 1929-01-01 1202.659  
## 2 2024-09-26    2024-09-26 1930-01-01 1100.67  
## 3 2024-09-26    2024-09-26 1931-01-01 1029.038  
## 4 2024-09-26    2024-09-26 1932-01-01 895.802  
## 5 2024-09-26    2024-09-26 1933-01-01 883.847  
## 6 2024-09-26    2024-09-26 1934-01-01 978.188  
## 7 2024-09-26    2024-09-26 1935-01-01 1065.716  
## 8 2024-09-26    2024-09-26 1936-01-01 1201.443  
## 9 2024-09-26    2024-09-26 1937-01-01 1264.393  
## 10 2024-09-26   2024-09-26 1938-01-01 1222.966  
## # i 85 more rows
```

Clean it up a bit and plot it

```
# library(lubridate) ## Already loaded above

fred =
  fred %>%
  mutate(across(realtime_start:date, ymd)) %>% # make all the dates, dates
  mutate(value = as.numeric(value)) # Make the values numeric
head(fred,3)
```

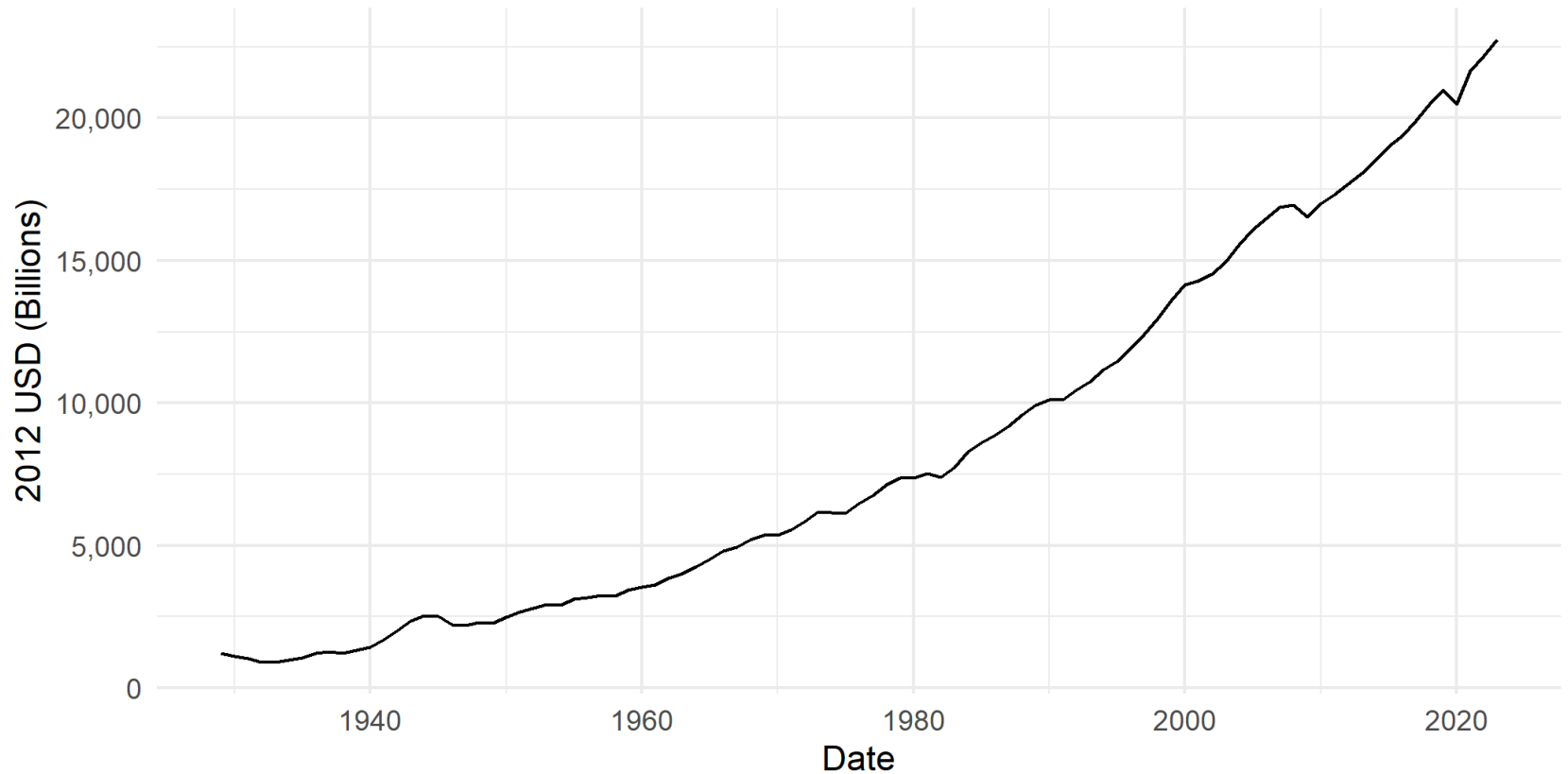
```
## # A tibble: 3 × 4
##   realtime_start realtime_end date      value
##   <date>         <date>      <date>    <dbl>
## 1 2024-09-26     2024-09-26  1929-01-01 1203.
## 2 2024-09-26     2024-09-26  1930-01-01 1101.
## 3 2024-09-26     2024-09-26  1931-01-01 1029.
```

Plot it

```
ggplot(fred, aes(x=date, y=value)) + # set your ggplot df and aesthetics
  geom_line() + # what geom?
  scale_y_continuous(labels = scales::comma) + # Make the scale prettier
  labs(
    x="Date", y="2012 USD (Billions)",
    title="US Real Gross National Product", caption="Source: FRED"
  )
```


Plot it

US Real Gross National Product



Source: FRED

Hide your API Key

- In general, you don't want to share your API key with anyone
- Instead, you can make it an environment variable either for a single session or permanently

```
Sys.setenv(FRED_API_KEY_TEST="abcdefghijklmnopqrstuvwxyz0123456789")  
FRED_API_KEY_TEST = Sys.getenv("FRED_API_KEY_TEST")  
FRED_API_KEY_TEST
```

```
## [1] "abcdefghijklmnopqrstuvwxyz0123456789"
```

- You can also permanently add it to your `.Renviron` file, by running the `edit_r_environ()` function from the **usethis** package
- Then just type in `FRED_API_KEY_TEST=abcdefghijklmnopqrstuvwxyz0123456789`, save, and re-read

```
usethis::edit_r_environ() # open R environment to edit  
readRenviron("~/Renviron") # read the .Renviron file
```

- Any time you need it, use `Sys.getenv("FRED_API_KEY_TEST")`

Popular APIs

- Many popular APIs are free to use and have a lot of documentation
- Sometimes the documentation gets a bit cumbersome though
- So kind souls have developed R packages to help you "abstract" these details (**Clean Code**)
- For example, the `tidycensus` package is a wrapper for the US Census API
 - You'll use it on your problem set
- Others include: `fredr`, `blsAPI`, `gh`, `googlesheets4`, `googledrive`, `wikipediR`, etc.
- Here's a curated list: <https://github.com/RomanTsegelskyi/r-api-wrappers>

Without tidycensus

- Sign up for a [Census API key](#)
- Get [API endpoint you want](#)
- Define other parameters
 - Series you want, your "get,:" i.e. B19013_001E is median household income, NAME is the name of the geography, GEOID is a census identifier
 - Figure out the types of parameters
 - Name the groups you want, in Census that is the "for" -- e.g. state, county, etc.
 - Name the groups you want, in Census that is your "in" -- e.g. Maine, Cumberland County, etc.

```
params_census <- list("key"=Sys.getenv('CENSUS_API_KEY'), ## Our parameter list
  "get" = "NAME,B19013_001E",
  "for" = "county:*",
  "in" = "state:23")
```

```
census =
  httr::GET(
    url = "https://api.census.gov/", ## Base URL
    path = "data/2017/acs/acs5", ## The API endpoint
    query = params_census,
    ) %>%
  httr::content(as="text") %>%
  jsonlite::fromJSON()
```

Census API differs from FRED

- Hey, wait that output a different structure than FRED did through this point
- So you need a different process to turn into a data table!

```
print(census)
```

```
##      [,1]      [,2]      [,3]      [,4]
## [1,] "NAME"    "B19013_001E" "state" "county"
## [2,] "Oxford County, Maine" "44582" "23"    "017"
## [3,] "Waldo County, Maine"  "50162" "23"    "027"
## [4,] "Penobscot County, Maine" "47886" "23"    "019"
## [5,] "Piscataquis County, Maine" "38797" "23"    "021"
## [6,] "Androscoggin County, Maine" "49538" "23"    "001"
## [7,] "Aroostook County, Maine" "39021" "23"    "003"
## [8,] "Washington County, Maine" "40328" "23"    "029"
## [9,] "Cumberland County, Maine" "65702" "23"    "005"
## [10,] "Knox County, Maine" "53117" "23"    "013"
## [11,] "Sagadahoc County, Maine" "60457" "23"    "023"
## [12,] "York County, Maine" "62618" "23"    "031"
## [13,] "Kennebec County, Maine" "50116" "23"    "011"
## [14,] "Franklin County, Maine" "45541" "23"    "007"
## [15,] "Somerset County, Maine" "41549" "23"    "025"
## [16,] "Hancock County, Maine" "51438" "23"    "009"
## [17,] "Lincoln County, Maine" "54041" "23"    "015"
```

For completion janitor package

- Oh shoot, I don't have a GEOID (FIPS code) for the counties!

```
# library(tidyverse)
# library(janitor)
census %>%
  as_tibble() %>%
  row_to_names(row_number=1)
```

```
## # A tibble: 16 × 4
##   NAME                                B19013_001E state county
##   <chr>                                <chr>      <chr> <chr>
## 1 Oxford County, Maine                44582      23    017
## 2 Waldo County, Maine                 50162      23    027
## 3 Penobscot County, Maine             47886      23    019
## 4 Piscataquis County, Maine           38797      23    021
## 5 Androscoggin County, Maine           49538      23    001
## 6 Aroostook County, Maine              39021      23    003
## 7 Washington County, Maine            40328      23    029
## 8 Cumberland County, Maine             65702      23    005
## 9 Knox County, Maine                  53117      23    013
## 10 Sagadahoc County, Maine             60457      23    023
## 11 York County, Maine                  62618      23    031
## 12 Kennebec County, Maine              50116      23    011
## 13 Franklin County, Maine              45541      23    007
## 14 Somerset County, Maine              41549      23    025
## 15 Hancock County, Maine               51438      23    009
## 16 Lincoln County, Maine               54041      23    015
```

Tidycensus

- Tidycensus embraces the **abstraction** principle of clean code

```
#library(tidycensus) # Already loaded
census_api_key("YOUR API KEY GOES HERE") # type this once and do not share your key
```

```
get_acs(geography = "county",
        state="ME",
        variables = "B19013_001E", # Median household income
        year = 2017,
        show_call = TRUE,
        survey='acs5')
```

```
## # A tibble: 16 × 5
##   GEOID NAME                variable estimate moe
##   <chr> <chr>                <chr>      <dbl> <dbl>
## 1 23001 Androscoggin County, Maine B19013_001 49538 1293
## 2 23003 Aroostook County, Maine   B19013_001 39021 1177
## 3 23005 Cumberland County, Maine  B19013_001 65702 1115
## 4 23007 Franklin County, Maine   B19013_001 45541 2739
## 5 23009 Hancock County, Maine    B19013_001 51438 1931
## 6 23011 Kennebec County, Maine    B19013_001 50116 1664
## 7 23013 Knox County, Maine       B19013_001 53117 2506
## 8 23015 Lincoln County, Maine     B19013_001 54041 2895
## 9 23017 Oxford County, Maine     B19013_001 44582 1758
## 10 23019 Penobscot County, Maine  B19013_001 47886 1189
## 11 23021 Piscataquis County, Maine B19013_001 38797 2314
## 12 23023 Sagadahoc County, Maine  B19013_001 60457 2953
## 13 23025 Somerset County, Maine  B19013_001 41540 1533
```

Hidden APIs

- Sometimes the API endpoint is hidden from view
- But you can find it by using the "Inspect" tool in your browser
- It will require some detective work!
- But if you pull it off, you can get data that no one else has

Server-side scraping

- The scripts that “build” the website are not run on our computer, but rather on a host server that sends down all of the HTML code.
 - E.g. Wikipedia tables are already populated with all of the information — numbers, dates, etc. — that we see in our browser.
- In other words, the information that we see in our browser has already been processed by the host server.
- You can think of this information being embedded directly in the webpage’s HTML.
 - So if we can get our hands on the HTML, we can get our hands on the data.
 - We just have to figure out how to strip off the HTML and get the data into a tidy format.
- **Webscrapping challenges:** Finding the correct CSS (or Xpath) “selectors”. Iterating through dynamic webpages (e.g. “Next page” and “Show More” tabs).
- **Key concepts:** CSS, Xpath, HTML
- **R package:** `rvest` has a suite of functions to help convert HTML to a tidy format

Underneath Wikipedia

W List of Olympic records in athlet

en.wikipedia.org/wiki/List_of_Olympic_records_in_athletics

Google Voice - Inbo... us.megabus.com/ab... Gmail YouTube Maps Random Econ Ideas DND Columbia Stuff Life Jazz All Bookmarks

Contents hide

(Top)

Men's records

Women's records

Mixed records

See also

References

External links


Beamon's compatriot, [Mike Powell](#), jumped farther in the [1991 World Championships in Athletics](#) in [Tokyo](#).^[1]

Note, only those events currently competed for and recognised by the IOC as Summer Olympic events are listed.^[8]


Men's records [edit]

♦ denotes a performance that is also a current [world record](#). Statistics are correct as of August 3, 2021.

Event	Record	Athlete(s)	Nation	Games	Date	Ref(s)
100 metres	9.63	Usain Bolt	 Jamaica (JAM)	2012 London	August 5, 2012	^[9]
200 metres	19.30	Usain Bolt	 Jamaica (JAM)	2008 Beijing	August 20, 2008	^[10]
400 metres	♦43.03	Wayde van Niekerk	 South Africa (RSA)	2016 Rio de Janeiro	August 14, 2016	^[11]
800 metres	♦1:40.91	David Rudisha	 Kenya (KEN)	2012 London	August 9, 2012	^[12]
1,500 metres	3:28.32	Jakob Ingebrigtsen	 Norway (NOR)	2020 Tokyo	August 7, 2021	^[13]
5,000 metres	12:57.82	Kenenisa Bekele	 Ethiopia (ETH)	2008 Beijing	August 23, 2008	^[14]
10,000 metres	27:01.17	Kenenisa Bekele	 Ethiopia (ETH)	2008 Beijing	August 17, 2008	^[15]
Marathon	2:06:32	Samuel Wanjiru	 Kenya (KEN)	2008 Beijing	August 24, 2008	^[16]
110 metres	12.91	Liu Xiang	 China (CHN)	2004	August 27, 2004	^[17]



[Usain Bolt](#) currently holds three Olympics records, two individually and one with the Jamaican 4 × 100m relay team.



The HTML source

- If we can just cut out all the HTML and get the data into a tidy format, we're golden
- Better yet, we can use some of the HTML to help us find **harvest** the data we want

```
<caption>List of men's Olympic records in athletics
</caption>
<tbody><tr>
<th scope="col" width="12%">Event
</th>
<th class="unsortable" width="5%">Record
</th>
<th scope="col" width="10%">Athlete(s)
</th>
<th scope="col" width="15%">Nation
</th>
<th scope="col" width="10%">Games
</th>
<th scope="col" width="5%">Date
</th>
<th scope="col" class="unsortable" width="3%">Ref(s)
</th></tr>
<tr>
<th scope="row"><span data-sort-value="00100&#160;!"><a href="/wiki/100_metres" title="100 metres">100
</th>
<td align="right">9.63&#160;
</td>
<td><span data-sort-value="Bolt, Usain"><span class="vcard"><span class="fn"><a href="/wiki/Usain_Bolt"
</td>
<td><span class="mw-image-border" typeof="mw:File"><span><img alt="" src="//upload.wikimedia.org/wikipedia
```

Stability and CSS scraping

- Websites change over time
- That can break your scraping code
- This makes scraping as much of an "art" as it is a science

Ethics of web scraping

Legality of web scraping

- All of today is about how to get data off the web
- If you can see it in a browser window and work out its structure, you can scrape it
- And the legal restrictions are pretty obscure, fuzzy, and ripe for reform
 - hiQ Labs vs LinkedIn court ruling defended hiQ's right to scrape, then the Supreme Court vacated the ruling, and the final decision was against HiQ Labs
 - The Computer Fraud and Abuse Act (CFAA) protects the scraping of publicly available data
 - Legality gets messy around personal data and intellectual property (for good reason, but again reform is needed)

Ethics of web scraping

- Technically, web scraping just automates what you (or a team of **well**-compensated RAs) could do manually
 - It's just a lot faster and more efficient (no offense)
- Webscraping is an integral tool to modern investigative journalism
 - Sometimes companies hide things in their HTML that they don't want the public to see
 - Pro Publica has developed a tool called [Upton](#) to make it more accessible
- So I stand firmly on the pro-scraping side with a few ethical caveats
 - Just because you can scrape it, doesn't mean you should
 - It's pretty easy to write up a function or program that can overwhelm a host server or application through the sheer weight of requests
 - Or, just as likely, the host server has built-in safeguards that will block you in case of a suspected malicious Denial-of-serve (DoS) attack

Be nice

- Once you get over the initial hurdles, scraping is fairly easy to do (cleaning can be trickier)
- There's plenty of digital ink spilled on the [ethics of web scraping](#)
- The key takeaway is to be nice
 - If a public API exists, use it instead of scraping
 - Only take the data that is necessary
 - Have good reason to take data that is not intentionally public
 - Do not repeatedly swarm a server with requests (use `sys.sleep()` to space out requests)
 - Scrape to add value to the data, not to take value from the host server
 - Properly cite any scraped content and respect the terms of service of the website
 - Document the steps taken to scrape the data

polite package and robots.txt

- Sites often have a "robot.txt," which is a file that tells you what you can and cannot scrape
- A "web crawler" should be written to start with the robots.txt and then follow the rules
- The `polite` package is a tool to help you be nice
- It explicitly checks for permissions and goes to the robots.txt of any site you visit
- As you get better at scraping and start trying to scrape at scale, you should use this

Conclusion

- Web content can be rendered either 1) server-side or 2) client-side.
- Client-side content is often rendered using an API endpoint, which is a URL that you can use to access the data directly.
 - APIs are a set of rules/methods that allow one software application to interact with another they often require an access token
 - You can use R packages (**httr**, **xml2** **jsonlite**) to access these endpoints and tidy the data.
 - Popular APIs have packages in R or other software that streamline access
- Server-side content is often rendered using HTML and CSS.
 - Use the **rvest** package to read the HTML document into R and then parse the relevant nodes.
 - A typical workflow is: `read_html(URL) %>% html_elements(CSS_SELECTORS) %>% html_table()`.
 - You might need other functions depending on the content type (e.g. `html_text`).
- Just because you can scrape something doesn't mean you should (i.e. ethical and possibly legal considerations).
- Webscraping involves as much art as it does science. Be prepared to do a lot of experimenting and data cleaning.

Next: Onto scraping and API activities!
