TEXT AS DATA: WEEK 3

MATTHIAS HABER

22 SEPTEMBER 2021

GOALS FOR TODAY

GOALS

- Introduction to data transformation with dplyr
- Learn how to collect your own (text) data from the web

PART I: DATA TRANSFORMATION WITH DPLYR

DATASET FOR TODAY

336,776 flights that departed from New York City in 2013

```
# install.packages("nycflights13")
library(nycflights13)
```

year	month	day	dep_time	sched_dep_time	dep_delay
2013	1	1	517	515	2
2013	1	1	533	529	4
2013	1	1	542	540	2
2013	1	1	544	545	-1

`DPLYR` CORE FUNCTIONS

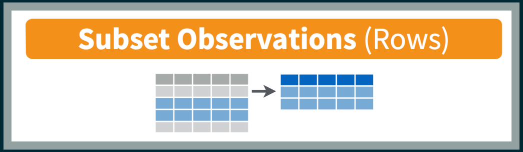
- `filter()`: select rows by their values
- `arrange()`: order rows
- `select()`: select columns by their names
- `mutate()`: create new variables
- `summarize()`: collapse many values down to a single summary
- `group_by()`: operate on it group-by-group
- `rename()`: rename columns
- `distinct()`: find distinct rows

DPLYR COMMAND STRUCTURE

- first argument is a data frame
- return value is a data frame
- nothing is modified in place

FILTER()

`filter()` allows to subset observations based on their values. The function takes logical expressions and returns the rows for which all are `TRUE`.



FILTER()

Let’s select all flights on January 1st:

```
filter(flights, month == 1, day == 1)
```

year	month	day	dep_time	sched_dep_time	dep_delay
2013	1	1	517	515	2
2013	1	1	533	529	4
2013	1	1	542	540	2
2013	1	1	544	545	-1
2013	1	1	554	600	-6
2013	1	1	554	558	-4

FILTER()

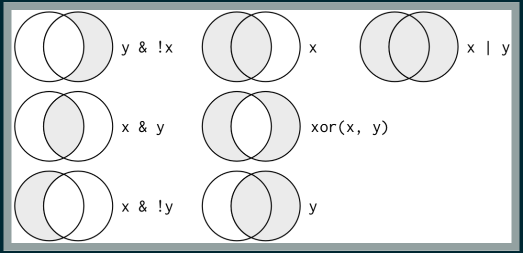
`filter()` revolves around using comparison operators: >, >=, <, <=, `!=` (not equal), and == (equal).

`dplyr` functions like `filter()` never modify inputs but instead return a new data frame that needs to be assigned to an object if you want to save the result.

```r
jan1 <- filter(flights, month == 1, day == 1)
```

BOOLEAN OPERATORS

`filter()` also supports the Boolean operators `&` (“and”), `|` (“or”), `!` (is “not”), and `xor` (exclusive “or”.



BOOLEAN OPERATORS

Why does this not work?

```
filter(flights, month == 11 | 12)
```

Generally a good idea to use `x %in% y`, which will select every row where `x` is part of the values of `y`.

```
filter(flights, month %in% c(11, 12))
```

`BETWEEN` CONDITION

Another useful dplyr filtering helper is `between()`. `between(x, left, right)` is equivalent to `x >= left & x <= right`.

To `filter()` all flights that departed between midnight and 6am (inclusive):

```
filter(flights, between(dep_time, 0, 600))
```

`FILTER()` EXERCISE

First, find all flights that had an arrival delay of two or more hours. Then find all flights that flew to Houston (`IAH` or `HOU`).

ARRANGE()

`arrange()` takes a data frame and a set of column names to order the rows by. Multiple column names are evaluated subsequently.

```
arrange(flights, year, month, day)
```

year	month	day	dep_time	sched_dep_time	dep_delay
2013	1	1	517	515	2
2013	1	1	533	529	4
2013	1	1	542	540	2
2013	1	1	544	545	-1
2013	1	1	554	600	-6
2013	1	1	554	558	-4

`ARRANGE()` IN DESCENDING ORDER

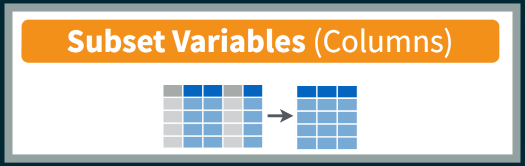
By dafault `arrange()` sorts values in ascending order. Use `desc()` to re-order by a column in descending order.

```
arrange(flights, desc(arr_delay))
```

year	month	day	dep_time	sched_dep_time	dep_delay
2013	1	9	641	900	1301
2013	6	15	1432	1935	1137
2013	1	10	1121	1635	1126
2013	9	20	1139	1845	1014
2013	7	22	845	1600	1005
2013	4	10	1100	1900	960

SELECT()

`select()` is used to select a subset of variables from a dataset.



```
select(flights, year, month, day)
```

year	month	day
2013	1	1
2013	1	1
2013	1	1
2013	1	1

`SELECT()`

`select()` has various helper functions:

- `everything()`: selects all variables.
- `starts_with("abc")`: matches names that begin with “abc”.
- `ends_with("xyz")`: matches names that end with “xyz”.
- `contains("ijk")`: matches names that contain “ijk”.
- `matches("(.)\\1")`: selects variables that match a regular expression.
- `num_range("x", 1:3)` matches `x1`, `x2` and `x3`.

SELECT()

You can use `select()` to rename variables

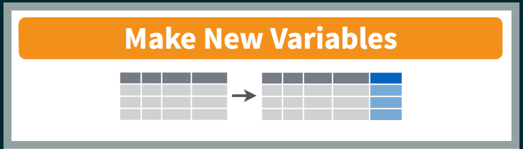
```
select(flights, tail_num = tailnum)
```

which will drop all of the variables not explicitly mentioned. Therefore it’s better to use `rename()` instead:

```
rename(flights, tail_num = tailnum)
```

MUTATE()

`mutate()` allows to add new columns to the end of your dataset that are functions of existing columns.



MUTATE( )

```
flights %>%
  select(ends_with("delay"), distance, air_time) %>%
  mutate(gain = arr_delay - dep_delay,
         speed = distance / air_time * 60
)
```

dep_delay	arr_delay	distance	air_time	gain	speed
2	11	1400	227	9	370.0441
4	20	1416	227	16	374.2731
2	33	1089	160	31	408.3750
-1	-18	1576	183	-17	516.7213
-6	-25	762	116	-19	394.1379
-4	12	719	150	16	287.6000

FUNCTIONS TO USE WITH `MUTATE()`

There are many functions for creating new variables with `mutate()`:

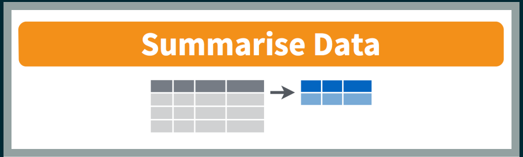
- Arithmetic operators: `+`, `-`, `*`, `/`, `^` (e.g. `air_time / 60`).
- Aggregate functions: `sum(x)` `mean(y)` (e.g. `mean(dep_delay)`).
- Logical comparisons, `<`, `<=`, `>`, `>=`, `!=`.
- ...

`MUTATE()` EXERCISES

Use `mutate()` to find the 10 most delayed flights using a ranking function (`?mutate`).

SUMMARIZE()

`summarize()` collapses a data frame to a single row.

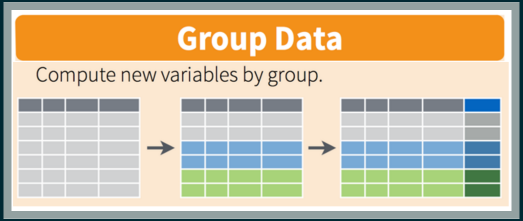


```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 × 1
##   delay
##   <dbl>
## 1  12.6
```

`SUMMARIZE()` WITH `GROUP_BY()`

`summarize()` is most effectively used with `group_by()`, which changes the unit of analysis from the complete dataset to individual groups.



Group Data

Compute new variables by group.

Grouping is most useful in conjunction with `summarise()`, but you can also do convenient operations with `mutate()` and `filter()`.

SUMMARIZE() WITH GROUP_BY()

For example, to get the average delay per date

```
flights %>%
  group_by(year, month, day) %>%
  summarise(delay = mean(dep_delay, na.rm = TRUE))
```

`SUMMARIZE()` COUNT

For aggregations it is generally a good idea to include a count `n()`. For example, let’s look at the (not cancelled) planes that have the highest average delays:

```
flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))
  group_by(tailnum) %>%
  summarise(delay = mean(arr_delay)) %>%
  arrange(delay)
```

`SUMMARIZE()` USEFUL FUNCTIONS

There are a number of useful summary functions:

- Measures of location: `mean(x)`, `sum(x)`, `median(x)`.
- Measures of spread: `sd(x)`, `IQR(x)`, `mad(x)`.
- Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`.
- Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`.
- Counts: `n()`, `sum(!is.na(x))`, `n_distinct(x)`.
- Counts and proportions of logical values: `sum(x > 10)`, `mean(y == 0)`.

`SUMMARIZE()` EXERCISES

Use `summarize()` to find the carrier with the worst delays.

LET'S TAKE A 10 MINUTE BREAK!

PART II: BASIC INTRODUCTION TO COLLECTING DATA FROM THE WEB

BROWSING VS. SCRAPING

- **Browsing**
  - you click on something
  - browser sends request to server that hosts website
  - server returns resource (often an HTML document)
  - browser interprets HTML and renders it in a nice fashion

BROWSING VS. SCRAPING

- **Scraping with R**
  - you manually specify a resource
  - R sends request to server that hosts website
  - server returns resource
  - R parses HTML (i.e., interprets the structure), but does not render it in a nice fashion
  - it’s up to you to tell R which parts of the structure to focus on and what content to extract

ONLINE TEXT DATA SOURCES

- **web pages** (e.g. http://example.com)
- **web formats** (XML, HTML, JSON, …)
- **web frameworks** (HTTP, URL, APIs, …)
- **social media** (Twitter, Facebook, LinkedIn, Snapchat, Tumbler, …)
- **data in the web** (speeches, laws, policy reports, news, …)
- **web data** (page views, page ranks, IP-addresses, …)

BEFORE SCRAPING, DO SOME GOOGLING!

- If the resource is well-known, someone else has probably built a tool which solves the problem for you.
- ropensci has a ton of R packages providing easy-to-use interfaces to open data.
- The Web Technologies and Services CRAN Task View is a great overview of various tools for working with data that lives on the web in R.

EXTRACTING DATA FROM HTML

For web scraping, we need to:

1. identify the elements of a website which contain our information of interest
2. extract the information from these elements

EXTRACTING DATA FROM HTML

For web scraping, we need to:

1. identify the elements of a website which contain our information of interest;
2. extract the information from these elements

Both steps require some basic understanding of HTML and CSS. More advanced scraping techniques require an understanding of *XPath* and *regular expressions*.

WHAT’S HTML?

HyperText Markup Language

- markup language = plain text + markups
- standard for the construction of websites
- relevance for web scraping: web architecture is important because it determines where and how information is stored

INSPECT THE SOURCE CODE IN YOUR BROWSER

Firefox 1. right click on page 2. select “View Page Source”

Chrome 1. right click on page 2. select “View Page Source”

Safari 1. click on “Safari” 2. select “Preferences” 3. go to “Advanced” 4. check “Show Develop menu in menu bar” 5. click on “Develop” 6. select “Show Page Source.”

CSS?

Cascading Style Sheets

- style sheet language to give browsers information of how to render HTML documents
- CSS code can be stored within an HTML document or in an external CSS file
- selectors, i.e. patterns used to specify which elements to format in a certain way, can be used to address the elements we want to extract information from
- works via tag name (e.g.,`<h2>`,`<p>`) or element attributes `id` and `class`

EXERCISE

1. Complete the first 5 levels on CSS Diner

XPATH

- XPath is a query language for selecting nodes from an XML-style document (including HTML)
- provides just another way of extracting data from static webpages
- you can also use XPath with R, it can be more powerful than CSS selectors

EXAMPLE

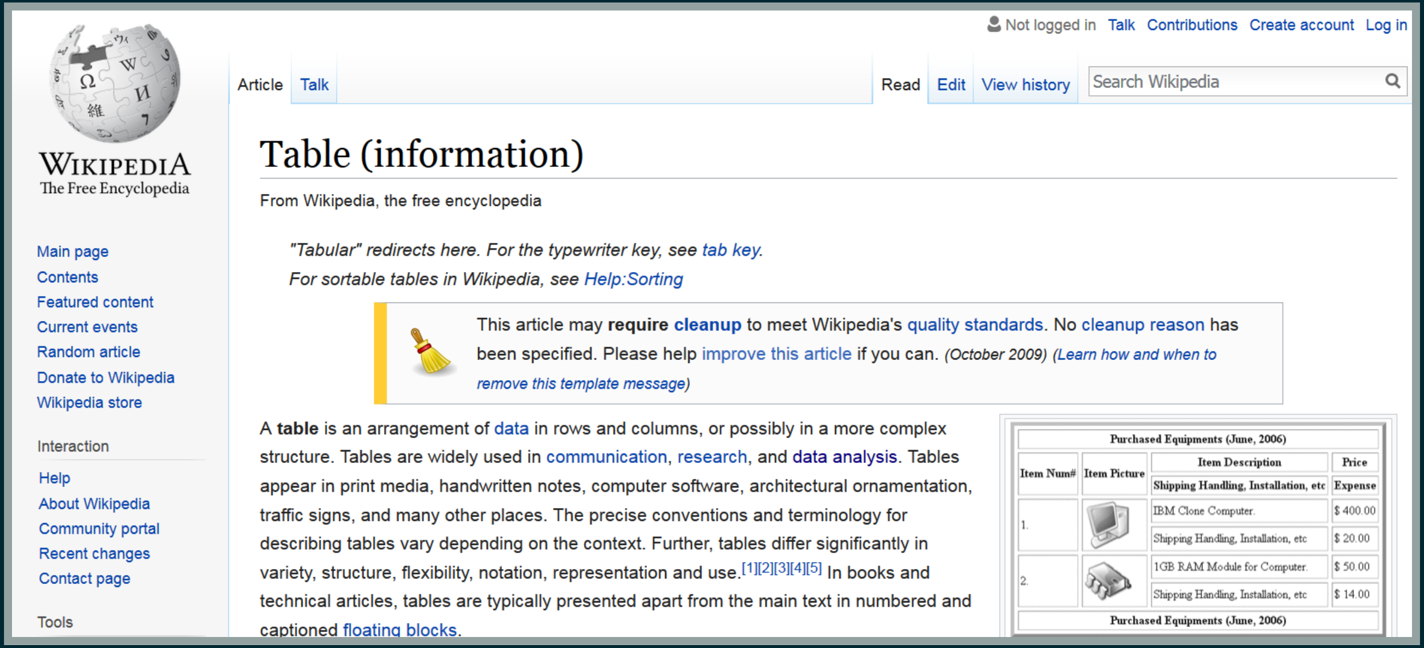


Table (information)

From Wikipedia, the free encyclopedia

"Tabular" redirects here. For the typewriter key, see tab key.
For sortable tables in Wikipedia, see Help:Sorting

This article may **require cleanup** to meet Wikipedia's quality standards. No cleanup reason has been specified. Please help improve this article if you can. *(October 2009) (Learn how and when to remove this template message)*

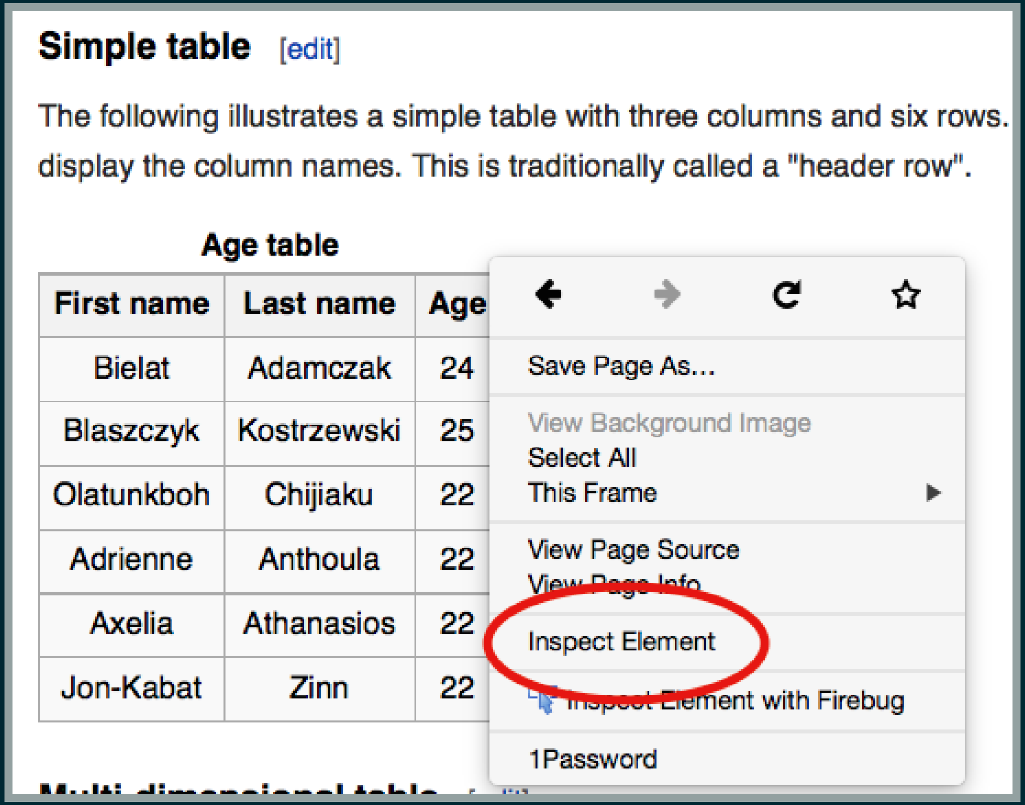
A **table** is an arrangement of data in rows and columns, or possibly in a more complex structure. Tables are widely used in communication, research, and data analysis. Tables appear in print media, handwritten notes, computer software, architectural ornamentation, traffic signs, and many other places. The precise conventions and terminology for describing tables vary depending on the context. Further, tables differ significantly in variety, structure, flexibility, notation, representation and use.[1][2][3][4][5] In books and technical articles, tables are typically presented apart from the main text in numbered and captioned floating blocks.

Purchased Equipments (June, 2006)

Item Num#	Item Picture	Item Description	Price
		Shipping Handling, Installation, etc	Expense
1.		IBM Clone Computer.	$ 400.00
		Shipping Handling, Installation, etc	$ 20.00
2.		1GB RAM Module for Computer.	$ 50.00
		Shipping Handling, Installation, etc	$ 14.00

Purchased Equipments (June, 2006)

INSPECTING ELEMENTS



Simple table [edit]

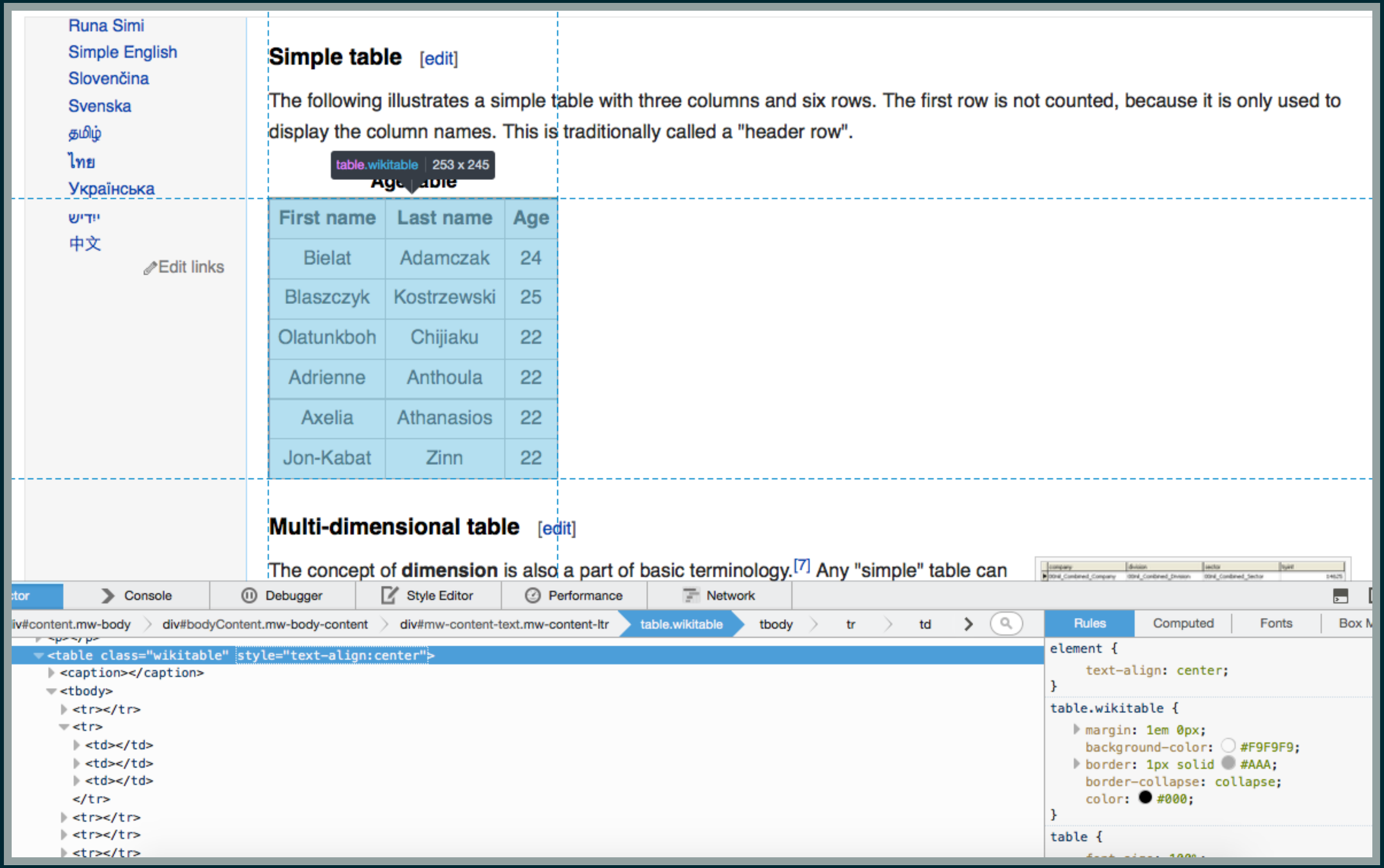
The following illustrates a simple table with three columns and six rows.
display the column names. This is traditionally called a "header row".

Age table

First name	Last name	Age
Bielat	Adamczak	24
Blaszczyk	Kostrzewski	25
Olatunkboh	Chijiaku	22
Adrienne	Anthoula	22
Axelia	Athanasios	22
Jon-Kabat	Zinn	22

Multi-dimensional table

HOVER TO FIND DESIRED ELEMENTS



Simple table [edit]

The following illustrates a simple table with three columns and six rows. The first row is not counted, because it is only used to display the column names. This is traditionally called a "header row".

Age table

First name	Last name	Age
Bielat	Adamczak	24
Blaszczyk	Kostrzewski	25
Olatunkboh	Chijiaku	22
Adrienne	Anthoula	22
Axelia	Athanasios	22
Jon-Kabat	Zinn	22

Multi-dimensional table [edit]

The concept of **dimension** is also a part of basic terminology.[7] Any "simple" table can

```
<table class="wikitable" style="text-align:center">
  <caption></caption>
  <tbody>
    <tr></tr>
    <tr>
      <td></td>
      <td></td>
      <td></td>
    </tr>
    <tr></tr>
    <tr></tr>
    <tr></tr>
```

```
element {
    text-align: center;
}
table.wikitable {
    margin: 1em 0px;
    background-color: #F9F9F9;
    border: 1px solid #AAA;
    border-collapse: collapse;
    color: #000;
}
table {
```

RVEST

rvest is a nice R package for web-scraping by (you guessed it) Hadley Wickham.

- see also: https://github.com/hadley/rvest
- convenient package to scrape information from web pages
- builds on other packages, such as xml2 and httr
- provides very intuitive functions to import and process webpages

BASIC WORKFLOW OF SCRAPING WITH RVEST

```r
# 1. specify URL
"http://en.wikipedia.org/wiki/Table_(information)" %>%

# 2. download static HTML behind the URL and parse it into an XML file
read_html() %>%

# 3. extract specific nodes with CSS (or XPath)
html_node(".wikitable") %>%

# 4. extract content from nodes
html_table(fill = TRUE)
```

```
## # A tibble: 9 × 3
##   `First name` `Last name`   Age
##   <chr>        <chr>       <int>
## 1 Tinu         Elejogun       14
## 2 Javier       Zapata         28
## 3 Lily         McGarrett      18
## 4 Olatunkbo    Chijiaku       22
## 5 Adrienne     Anthoula       22
## 6 Axelia       Athanasios     22
## 7 Jon-Kabat    Zinn           22
## 8 Thabang      Mosoa          15
## 9 Kgaogelo     Mosoa          11
```

SELECTORGADGET

- Selectorgadget is a Chrome browser extension for quickly extracting desired parts of an HTML page.
- to learn about it, use vignette(“selectorgadget”)
- to install it, visit http://selectorgadget.com/

SELECTORGADGET

```r
url <- "http://spiegel.de/schlagzeilen"
css <- ".mr-6"
url_parsed <- read_html(url)
html_nodes(url_parsed, css = css) %>% html_text
```

EXERCISE

We want to collect some questions that users asks SPD candidate Olaf Scholz from https://www.abgeordnetenwatch.de/profile/olaf-scholz/fragen-antworten.

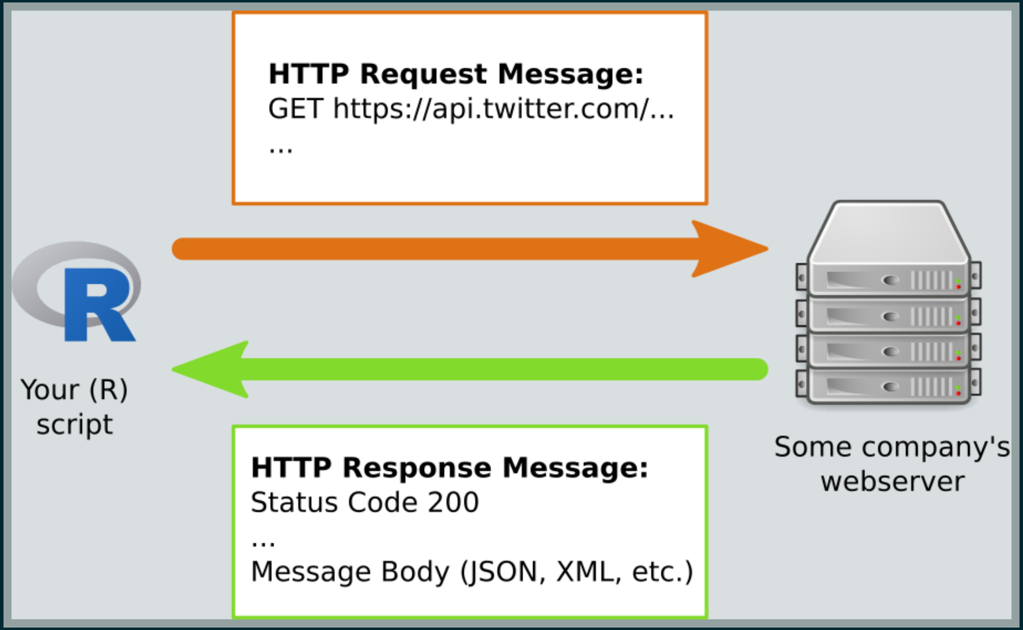
1. What’s the first thing you would do?
2. Oh no, SelectorGadget does not work. What else can we do?
3. Write a simple scraper to collect the first 12 questions
4. How can we get all the questions for Olaf Scholz? Every candidate?

APIS

- API stands for *Application Programming Interface*
- defined interface for communication between software components
- *Web* API: provides an interface to structured data from a web service
- APIs should be used whenever you need to automatically collect mass data from the web
- it should definitely be preferred over web scraping

FUNCTIONALITY OF APIS

Web APIs usually employ a **client-server model**. The client – that is you. The server provides the *API endpoints* as URLs.


HTTP Request Message:
GET https://api.twitter.com/...
...
Your (R) script
Some company's webserver
HTTP Response Message:
Status Code 200
...
Message Body (JSON, XML, etc.)

FUNCTIONALITY OF APIS

Communication is done with request and response messages over *Hypertext transfer protocol (HTTP)*.

Each HTTP message contains a *header* (message meta data) and a *message body* (the actual content). The three-digit HTTP status code plays an important role:

- 2xx: Success
- 4xx: Client error (incl. the popular *404: Not found* or *403: Forbidden*)
- 5xx: Server error
- The message body contains the requested data in a specific format, often JSON or XML.

EXAMPLES OF POPULAR APIS

Social media:

- Twitter
- Facebook Graph API (restricted to own account and public pages)
- YouTube (Google)
- LinkedIn
- For more, see programmableweb.com.

API WRAPPER PACKAGES

- Working with a web API involves:
    - constructing request messages
    - parsing result messages
    - handling errors
- For popular web services there are already “API wrapper packages” in R:
    - implement communication with the server
    - provide direct access to the data via R functions
    - examples: *rtweet*, *ggmap* (geocoding via Google Maps), *wikipediR*, etc.

TWITTER

Twitter has two types of APIs

- REST APIs –> reading/writing/following/etc.
- Streaming APIs –> low latency access to 1% of global stream - public, user and site streams
- authentication via OAuth
- documentation at https://dev.twitter.com/overview/documentation

ACCESSING THE TWITTER APIS

- To access the REST and streaming APIs, all you need is a Twitter account and you can be up in running in minutes!
- Simply send a request to Twitter’s API (with a function like `search_tweets()`) during an interactive session of R, authorize the embedded `rstats2twitter` app (approve the browser popup), and your token will be created and saved/stored (for future sessions) for you!
- You can obtain a developer account to get more stability and permissions.

USE TWITTER IN R

```r
library(rtweet)

## search for 1000 tweets using the #baerbock hashtag
tweets <- search_tweets(
  "#baerbock", n = 1000, include_rts = FALSE
)
```

Find out what else you can do with the `rtweet` package: https://github.com/mkearney/rtweet

HOMEWORK EXERCISE

1. We are still interested in getting data from abgeordnetenwatch.de. The site has an API, so technically there is no need to scrape anything. Load the package `jsonlite` into library.

2. Go to https://www.abgeordnetenwatch.de/api and figure our the syntax of their APIs.

3. Collect some data from their API from a politician running in your constituency (or a constituency of your choice). Tip: Use the `fromJSON` function in `jsonlite` to load a JSON file via the API into R. Convert the output into a nicely formatted dataframe.

WRAPPING UP

QUESTIONS?

OUTLOOK FOR OUR NEXT SESSION

- Next week we will learn how to clean and transform text
- We will meet online on MS Teams again

THAT'S IT FOR TODAY

Thanks for your attention!