# Hacettepe University
# Department of Industrial Engineering
# Undergraduate Program
# 2023-2024 Fall

## EMU 430 – Data Analytics
## Week 11
## December 15, 2023

**Instructor:** Erdi Dasdemir

edasdemir@hacettepe.edu.tr
www.erdidasdemir.com

# Outline

- Introduction to Data Wrangling
- Importing Spreadsheets
- Tidy Your Data
- Combining Tables
- Web Scrapping
- rvest Package
- JSON and jsonlite package
- API and httr2 package

# Acknowledgment

I drew inspiration primarily from Dr. Rafael Irizarry's "Introduction to Data Science" Book

and "Data Science" course by HarvardX on edX for the slides this week.

22 Aralık 2023
15:00

# Veri Bilimi ve Endüstri Müh.

Hacettepe Üniversitesi Endüstri Mühendisliği Bölümü
Davetli Konuşmalar Serisi

**Konuşmacı**
**Sami Serkan Özarık (Dr.)**
**Yöneylem Araştırması ve**
**Veri Bilimi Uzmanı**
**Amazon, Lüksemburg**

Moderatör
Erdi Daşdemir
Veri Bilimi Uzmanı ve
Dr. Öğretim Üyesi
Hacettepe Endüstri Mühendsiliği

**Etkinlik Yeri** Hacettepe Üniversitesi Endüstri Mühendisliği Bölümü, D4 Sınıfı

Not: Bu konuşma EMÜ430 Veri Analitiği ve EMÜ679 YA'da İleri Matematiksel Modelleme dersleri kapsamında düzenlenmektedir. Konuşma, EMU Hacettepe öğrencilerine ve ilgili misafir öğrencilere açıktır.

# Introduction to Data Wrangling

# Introduction to Data Wrangling

- **Data wrangling** is the process of converting raw data into a usable form.
- Data sets are not tidy in real life.
- import the data into R → tidy up the data → start analysis

# Introduction to Data Wrangling

We will learn about common data-wrangling proocess.

- importing data into R from files,
- tidying data,
- string processing,
- HTML parsing,
- working with dates and times, and
- text mining.

# Importing Spreadsheets

**Reading a file that is already on our computer.**

- In R, it is important to know your working directory. This is the directory in which R will save or look for files by default.
- Get your working directory

```
getwd()
```

- Change your working directory

```
setwd()
```

If you are using RStudio, Session –> Set Working Directory.

- **Important:** One thing file reading functions have in common is that unless a full path is provided, they search for files in the working directory.

- **Recommendation:** Create a directory for each analysis and keep the raw data files in that directory. To make it more organized, create a data directory (folder) inside your project directory.

- **Example.** dslabs package has a raw data files as example. To find their locations:

```
system.file("extdata", package = "dslabs")
```

**readr** and **readxl** are the tidyverse libraries that include functions for reading data stored in spreadsheets into R.

```
library(readr)
library(readxl)
```

# readr

| Function | Format | Typical Suffix |
|---|---|---|
| read_table | white space separated values | txt |
| read_csv | comma separated values | csv |
| read_csv2 | semicolon separated values | csv |
| read_tsv | tab delimited separated values | tsv |
| read_delim | general text file format, must define delimiter | txt |

**base R functions to import data**

- `read.table`
- `read.csv`
- `read.delim`

# readxl

| Function | Format | Typical Suffix |
|---|---|---|
| read_excel | auto detect the format | xls, xlsx |
| read_xls | original format | xls |
| read_xlsx | new format | xlsx |

We can import or download data files from web

```
url <-
"https://raw.githubusercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv"
```

To read murders.csv from web: `dat <- read_csv(url)`

To download a local copy and read it:

- `download.file(url, "murders.csv")`
- `dat <- read_csv("murders.csv"))`

# Tidy Your Data

**tidy data:** each row represents one observation and the columns represent the different variables that we have data on for those observations.

**Example:** Remember South Korea and Germany example

Once the data is proper we can use our `dplyr` and `ggplot` functions easily.

```
data("gapminder")
tidy_data <- gapminder %>% filter(country %in% c("South Korea", "Germany")) %>%
select(country, year, fertility)

head(tidy_data)
tidy_data %>% ggplot(aes(year, fertility, color = country)) + geom_point()
```

Example, let's go to the original raw version of this data file.

```
path <- system.file("extdata", package="dslabs")
filename <- file.path(path, "fertility-two-countries-example.csv")
wide_data <- read_csv(filename)
wide_data %>% select(country, '1960':'1967')
```

The data is in a wide format.

**Wide Format (compared to Tidy format)**

- each row includes several observations
- one of the variables is stored in the header
- ggplot does not work with wide format → we need to wrangle it to tidy format
- `tidyr` package (included in tidyverse library)

gather(): converts wide data into tidy data

gather() is no longer under active development

```
help("gather")
```

- default version gathers all columns, therefore we need to specify the columns.
- we want to gather columns 1960 … 2015

```
new_tidy_data <- wide_data %>% gather(key = year, value =
fertility,'1960':'2015')
new_tidy_data2 <- gather(data = wide_data, key = year, value =
fertility,'1960':'2015')
head(new_tidy_data)
```

# tidy your data

## New Updates

1. `pivot_longer()` is an updated approach to `gather()`, designed to be both simpler to use and to handle more use cases. We recommend you use `pivot_longer()` for new code; `gather()` isn't going away but is no longer under active development.

2. R 4.1.0 introduced a native pipe operator, `|>`. The behaviour of the native pipe is by and large the same as that of the %>% pipe provided by the magrittr package.

## `pivot_longer()`: converts wide data into tidy data

```
help("pivot_longer")
```

- we want to gather columns 1960 …. 2015

```
new_tidy_data <- wide_data |> pivot_longer('1960':'2015', names_to =
"year", values_to = "fertility")
```

or

```
new_tidy_data <- wide_data |> pivot_longer(-country, names_to =
"year", values_to = "fertility")

head(new_tidy_data)
```

here is another issue: **pivot_longer function assumes column names are characters**

```
class(tidy_data$year)
[1] "integer"
class(new_tidy_data$year)
[1] "character“
```

we can use as.numeric() but gather as an argument for this.

```
new_tidy_data <- wide_data |> pivot_longer(-country, names_to = “year”,
values_to = “fertility”) |> mutate(year = as.integer(year))
head(new_tidy_data)
```

**`spread()` function**

spread() is no longer under active development

inverse of `gather()`

```
help(spread)

new_wide_data <- new_tidy_data %>% spread(key = year, value = fertility)

select(new_wide_data, country, '1960':'1967')
```

## `pivot_wider()` function

inverse of `gather()`

```
help(spread)
```

```
new_wide_data <- new_tidy_data |> spread(names_from = year, values_from =
fertility)
```

```
select(new_wide_data, country, '1960':'1967')
```

https://raw.githubusercontent.com/rstudio/cheatsheets/main/tidyr.pdf

# Reshape Data

- Pivot data to reorganize values into a new layout.

table4a

| country | 1999 | 2000 |
|---|---|---|
| A | 0.7K | 2K |
| B | 37K | 80K |
| C | 212K | 213K |

→

| country | year | cases |
|---|---|---|
| A | 1999 | 0.7K |
| B | 1999 | 37K |
| C | 1999 | 212K |
| A | 2000 | 2K |
| B | 2000 | 80K |
| C | 2000 | 213K |

**pivot_longer(**data, cols, names_to = "name", values_to = "value", values_drop_na = FALSE**)**

"Lengthen" data by collapsing several columns into two. Column names move to a new names_to column and values to a new values_to column.

```
pivot_longer(table4a, cols = 2:3, names_to ="year",
  values_to = "cases")
```

table2

| country | year | type | count |
|---|---|---|---|
| A | 1999 | cases | 0.7K |
| A | 1999 | pop | 19M |
| A | 2000 | cases | 2K |
| A | 2000 | pop | 20M |
| B | 1999 | cases | 37K |
| B | 1999 | pop | 172M |
| B | 2000 | cases | 80K |
| B | 2000 | pop | 174M |
| C | 1999 | cases | 212K |
| C | 1999 | pop | 1T |
| C | 2000 | cases | 213K |
| C | 2000 | pop | 1T |

→

| country | year | cases | pop |
|---|---|---|---|
| A | 1999 | 0.7K | 19M |
| A | 2000 | 2K | 20M |
| B | 1999 | 37K | 172M |
| B | 2000 | 80K | 174M |
| C | 1999 | 212K | 1T |
| C | 2000 | 213K | 1T |

**pivot_wider(**data, names_from = "name", values_from = "value"**)**

The inverse of pivot_longer(). "Widen" data by expanding two columns into several. One column provides the new column names, the other the values.

```
pivot_wider(table2, names_from = type,
  values_from = count)
```

```
path <- system.file("extdata", package="dslabs")
filename <- file.path(path, "life-expectancy-and-fertility-two-countries-example.csv")
raw_dat <- read_csv(filename)
select(raw_dat, 1:5)
```

We will not use column name “year” as the new column name as they also include type information.

```
dat <- raw_dat %>% pivot_longer(-country)
head(dat)
```

Encoding multiple variables in a column name is a common problem. Hence, `readr()` has a function for this: `separate()`

```
dat %>% separate_wider_delim(name, delim = "_", names = "year","name"), too_many = “merge")
```

Convert to wide format:

```
dat %>% pivot_wider() # creaates column for each variable
```

# Combining Tables

# Combining Tables

We may have multiple data files.

**Example:** We want to investigate the relationship between population and electoral votes. These are in different data sets.

```
data(murders)
head(murders)

results_us_election_2016
murders
```

The order of states is different in the two tables. We cannot simply put them together using column binding.

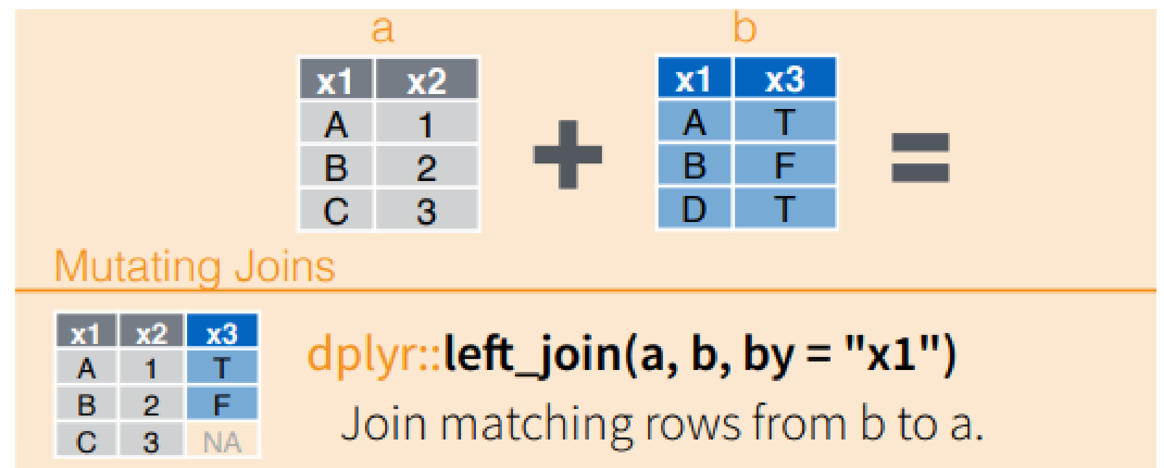
```
identical(results_us_election_2016$state, murders$state)
```

These are functions from `dplyr` package:

based on SQL joins.

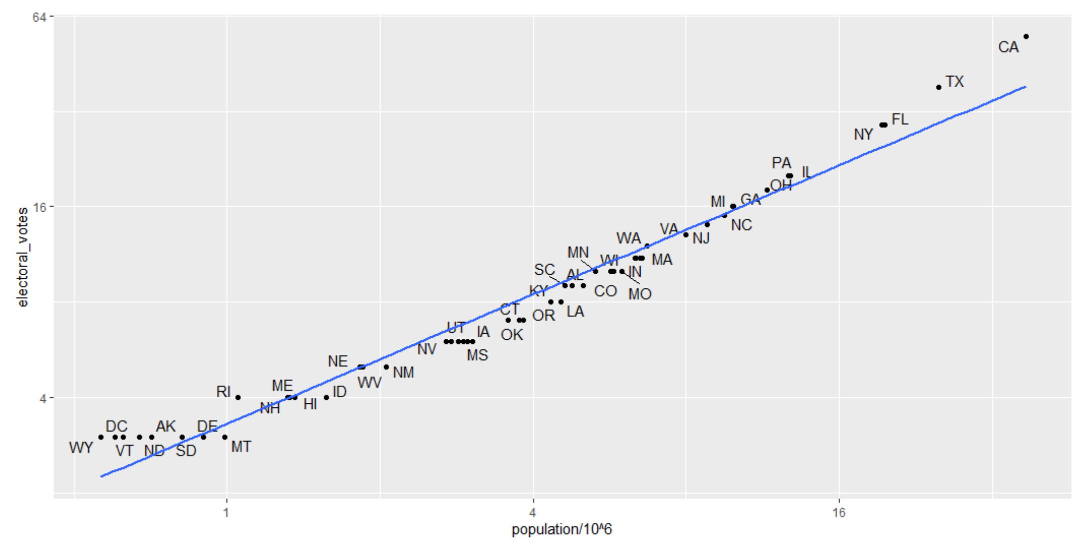
## `left_join`

```
help("left_join")
tab <- left_join(murders,
results_us_election_2016, by =
"state")
head(tab)
```



We can now make a plot to see the relationship

```
library(ggrepel)
tab %>% ggplot(aes(population/10^6,
electoral_votes, label = abb)) +
geom_point() + geom_text_repel() +
scale_x_continuous(trans = "log2") +
scale_y_continuous(trans = "log2") +
geom_smooth(method = "lm", se = FALSE)
```



- In real-life, it is not always the case that each row in one table has a matching row in the other.

Example:

```
results_us_election_2016 <- results_us_election_2016 %>%
arrange(state)

tab1 <- slice(murders, 1:6) %>%  select(state, population)
tab1

tab2 <- slice(results_us_election_2016, c(1:3, 5, 7:8)) %>%
select(state, electoral_votes)
tab2
```

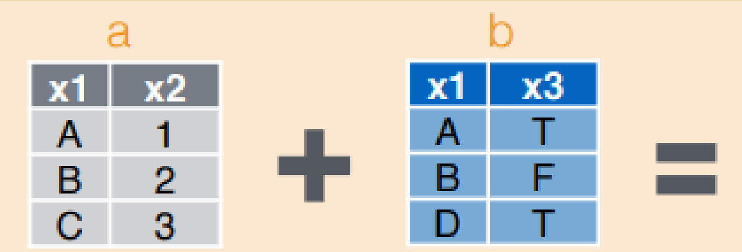
```
# left join
left_join(tab1, tab2)

# right join
right_join(tab1, tab2)

# keep only the rows that have
information in both tables
# inner join
inner_join(tab1, tab2) # intersection

# keep all rows and assign NAs
full_join(tab1, tab2)
```

## Combine Data Sets




Mutating Joins

| x1 | x2 | x3 |
| --- | --- | --- |
| A | 1 | T |
| B | 2 | F |
| C | 3 | NA |

dplyr::**left_join(a, b, by = "x1")**

Join matching rows from b to a.

| x1 | x3 | x2 |
| --- | --- | --- |
| A | T | 1 |
| B | F | 2 |
| D | T | NA |

dplyr::**right_join(a, b, by = "x1")**

Join matching rows from a to b.

| x1 | x2 | x3 |
| --- | --- | --- |
| A | 1 | T |
| B | 2 | F |

dplyr::**inner_join(a, b, by = "x1")**

Join data. Retain only rows in both sets.

| x1 | x2 | x3 |
| --- | --- | --- |
| A | 1 | T |
| B | 2 | F |
| C | 3 | NA |
| D | NA | T |

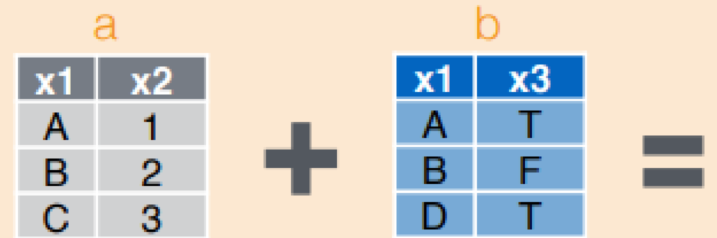
dplyr::**full_join(a, b, by = "x1")**

Join data. Retain all values, all rows.

```
# semi_join keeps the part of the
first table for which we have
information in the second.
semi_join(tab1, tab2)

# anti_join keeps the part of first
table for which we have no information
in the second.
anti_join(tab1, tab2)
```

## Combine Data Sets




Filtering Joins

| x1 | x2 |
|---|---|
| A | 1 |
| B | 2 |

dplyr::**semi_join(a, b, by = "x1")**

All rows in a that have a match in b.

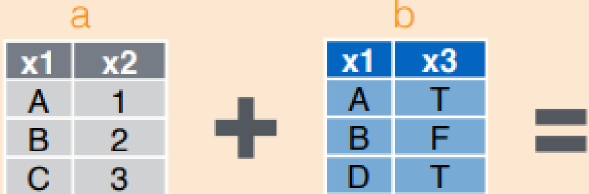
| x1 | x2 |
|---|---|
| C | 3 |

dplyr::**anti_join(a, b, by = "x1")**

All rows in a that do not have a match in b.

[Data Wrangling with dplyr and tidyr Cheat Sheet]

# Combine Data Sets



a

| x1 | x2 |
|---|---|
| A | 1 |
| B | 2 |
| C | 3 |

\+

b

| x1 | x3 |
|---|---|
| A | T |
| B | F |
| D | T |

=

## Mutating Joins

| x1 | x2 | x3 |
|---|---|---|
| A | 1 | T |
| B | 2 | F |
| C | 3 | NA |

dplyr::**left_join(a, b, by = "x1")**

Join matching rows from b to a.

| x1 | x3 | x2 |
|---|---|---|
| A | T | 1 |
| B | F | 2 |
| D | T | NA |

dplyr::**right_join(a, b, by = "x1")**

Join matching rows from a to b.

| x1 | x2 | x3 |
|---|---|---|
| A | 1 | T |
| B | 2 | F |

dplyr::**inner_join(a, b, by = "x1")**

Join data. Retain only rows in both sets.

| x1 | x2 | x3 |
|---|---|---|
| A | 1 | T |
| B | 2 | F |
| C | 3 | NA |
| D | NA | T |

dplyr::**full_join(a, b, by = "x1")**

Join data. Retain all values, all rows.

## Filtering Joins

| x1 | x2 |
|---|---|
| A | 1 |
| B | 2 |

dplyr::**semi_join(a, b, by = "x1")**

All rows in a that have a match in b.

| x1 | x2 |
|---|---|
| C | 3 |

dplyr::**anti_join(a, b, by = "x1")**

All rows in a that do not have a match in b.

dplyr has `bind_cols()`: binds two objects by putting the columns of each together in a tibble

```
bind_cols(a = 1:3, b = 4:6)

cbind(a = 1:3, b = 4:6)
```

default R column binding creates objects (data frames) rather than tibbles.

We can bind data frames too

```
tab1 <- tab[, 1:3]
tab2 <- tab[, 4:6]
tab3 <- tab[, 7:9]
new_tab <-
bind_cols(tab1, tab2,
tab3)
head(new_tab)
```

y

| x1 | x2 |
|---|---|
| A | 1 |
| B | 2 |
| C | 3 |

z

| x1 | x2 |
|---|---|
| B | 2 |
| C | 3 |
| D | 4 |

Binding

| x1 | x2 |
|---|---|
| A | 1 |
| B | 2 |
| C | 3 |
| B | 2 |
| C | 3 |
| D | 4 |

dplyr::**bind_rows(y, z)**

Append z to y as new rows.

| x1 | x2 | x1 | x2 |
|---|---|---|---|
| A | 1 | B | 2 |
| B | 2 | C | 3 |
| C | 3 | D | 4 |

dplyr::**bind_cols(y, z)**

Append z to y as new columns.

Caution: matches rows by position.

`bind_rows()` is similar but binds rows

```
tab1 <- tab[1:2, ]

tab2 <- tab[3:4, ]

bind_rows(tab1, tab2)

rbind(tab1, tab2)
```

y

| x1 | x2 |
|---|---|
| A | 1 |
| B | 2 |
| C | 3 |

\+

z

| x1 | x2 |
|---|---|
| B | 2 |
| C | 3 |
| D | 4 |

=

Binding

| x1 | x2 |
|---|---|
| A | 1 |
| B | 2 |
| C | 3 |
| B | 2 |
| C | 3 |
| D | 4 |

dplyr::**bind_rows(y, z)**

Append z to y as new rows.

| x1 | x2 | x1 | x2 |
|---|---|---|---|
| A | 1 | B | 2 |
| B | 2 | C | 3 |
| C | 3 | D | 4 |

dplyr::**bind_cols(y, z)**

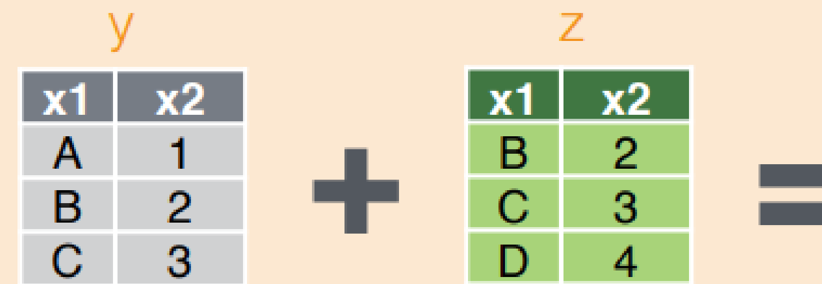
Append z to y as new columns.

Caution: matches rows by position.

```r
tab1 <- tab[1:5, ]
tab2 <- tab[3:7, ]
intersect(tab1, tab2) # intersecting rows
```

```r
tab1 <- tab[1:5, ]
tab2 <- tab[3:7, ]
union(tab1, tab2) # union rows
```

```r
tab1 <- tab[1:5, ]
tab2 <- tab[3:7, ]
setdiff(tab1, tab2) # setdiff()
```




Set Operations

| x1 | x2 |
|---|---|
| B | 2 |
| C | 3 |

**dplyr::intersect(y, z)**
Rows that appear in both y and z.

| x1 | x2 |
|---|---|
| A | 1 |
| B | 2 |
| C | 3 |
| D | 4 |

**dplyr::union(y, z)**
Rows that appear in either or both y and z.

| x1 | x2 |
|---|---|
| A | 1 |

**dplyr::setdiff(y, z)**
Rows that appear in y but not z.

```
v1 <- c(1:5)
v2 <- c(5:1)
v3 <- c(1:6)
setequal(v1, v2)
TRUE

setequal(v1, v3)
FALSE
```

# Web Scrapping

# Web Scrapping

- The data we need to answer questions are not always in a spreadsheet ready for us to read.
- For example, the US murders data set orinally came from this Wikipedia page:
- Wikipedia Page: Murder in the United States by state
- Web scraping or web harvesting are the terms used to describe the process of extracting data from a

  website.
- We can do this is because the information from web pages to our browsers are received as text from a server.

.

- A webpage is a computer code written in **HyperText Markup Language or HTML.**

- To see the code for a web page, you can actually visit the page on your browser and then view the code.

- Different browsers have different ways of doing this. In Chrome you can click on View Source to see it.

- Because the code is accessible, we can download the HTML files, import it into R, and then write programs to extract the information we need from the page.

- Once we look at HTML code, this might seem like a difficult task.
- Fortunately, there are convenient tools to facilitate the process.
- Lets look at the source code and search Alabama.
- We can see the data and the pattern that the data is defined with.
- If you know HTML, you know what these patterns are, and you can leverage this knowledge to extract what we need.
- We can also take advantage of a language widely used to make web pages look pretty called **Cascading Style Sheets, or CSS.**

- Although we will learn about the tools that make it possible to scrape data without knowing HTML, for data scientists, **it is quite useful to learn some HTML and some CSS.**

- Useful courses for web design and development

# rvest Package

- We will use `rvest` package for web scraping.
- It is part of the tidyverse.
- The first step using this package is to import the web page into R:

```
library(rvest)
url <-
"https://en.wikipedia.org/w/index.php?title=Gun_violence_in_the_United_Stat
es_by_state&direction=prev&oldid=810166167"
data_html <- read_html(url)
class(data_html)
```

## XML: General Markup Language

- The `rvest` package is actually more general. It handles **XML** documents, not just **HTML** documents.
- XML can be used to represent any kind of data.
- HTML is a specific type of XML, specifically developed for representing web pages.

## Extracting Information

- We know that the information is store in an HTML table (refer to source code).
- In HTML, information is stored inside nodes < >

For example,

- `<td> 348 </td>`
- `<p><strong>Hi, I’m Aykut. I’m third year Industrial Engineering Student at Hacettepe University.</strong></p>`
- `rvest` package has functions to extract nodes from HTML documents.

- We know that the information is store in an HTML table (refer to source code).
- In HTML, information is stored inside nodes < >

For example,

- `<td> 348 </td>`
- `<p><strong>Hi, I'm Aykut. I'm third year Industrial Engineering Student at Hacettepe University.</strong></p>`
- `rvest` package has functions to extract nodes from HTML documents.

- The default look of a webpage made with the most basic HTML is quite unattractive.
- The aesthetically pleasing pages we see today are made using CSS
- The general way these CSS files work is by defining how each of the elements of a webpage will look.
- CSS does this by leveraging patterns used to define these elements, referred to as *selectors*. An example of such a pattern, which we used above, is `table`, but there are many, many more.
- If we want to grab data from a webpage and we happen to know a selector that is unique to the part of the page containing this data, we can use the `html_nodes` function.
- However, knowing which selector can be quite complicated.

- **SelectorGadget** is piece of software that allows you to interactively determine what CSS selector you need to extract specific components from the webpage.

- A Chrome extension is available which permits you to turn on the gadget and then, as you click through the page, it highlights parts and shows you the selector you need to extract these parts.

Demos:

- https://rvest.tidyverse.org/articles/selectorgadget.html

- https://www.analyticsvidhya.com/blog/2017/03/beginners-guide-on-web-scraping-in-r-using-rvest-with-hands-on-knowledge/

```
library(rvest)
url <-
"https://en.wikipedia.org/w/index.php?title=Gun_violence_in_the_United_Stat
es_by_state&direction=prev&oldid=810166167"
data_html <- read_html(url)
class(data_html)

tab <- data_html |> html_nodes("table")
tab <- tab[[1]] |> html_table()
tab <- tab |> setNames(c("state", "population", "total", "murder_rate"))
head(tab)
```

# JSON
# and
# jsonlite package

- Sharing data on the internet has become more and more common.
- There are some standards that are also becoming more common.
- Currently, a format that is widely being adopted is the **JavaScript Object Notation or JSON.**

```
#> [
#>   {
#>     "name": "Miguel",
#>     "student_id": 1,
#>     "exam_1": 85,
#>     "exam_2": 86
#>   },
#>   {
#>     "name": "Sofia",
#>     "student_id": 2,
#>     "exam_1": 94,
#>     "exam_2": 93
#>   },
#>   {
#>     "name": "Aya",
#>     "student_id": 3,
#>     "exam_1": 87,
#>     "exam_2": 88
#>   },
#>   {
#>     "name": "Cheng",
#>     "student_id": 4,
#>     "exam_1": 90,
#>     "exam_2": 91
#>   }
#> ]
```

## `jsonlite` package

We can use the function `fromJSON` from the **jsonlite** package to read JSON files.

Note that JSON files are often made available via the internet.

Several organizations provide a JSON API or a web service that you can connect directly to and obtain data.

## jsonlite package

Here is an example providing information Nobel prize winners:

```
library(jsonlite)
library(dplyr)
nobel <- fromJSON("http://api.nobelprize.org/v1/prize.json")
nobel$prizes %>% .$category
nobel$prizes %>% .$year
nobel$prizes %>% filter(category == "literature" & year == "1971") %>%
pull(laureates) %>% first() %>% select(id, firstname, surname)
   id firstname surname
```

You can learn much more by examining tutorials and help files for `jsonlite` and `rjason` packages.

# API and httr2 package

- An Application Programming Interface (API) is a set of rules and protocols that allows different software entities to communicate with each other.

- It defines methods and data formats that software components should use when requesting and exchanging information.

- APIs play a crucial role in enabling the integration that make today's software so interconnected and versatile.

- There are several types of APIs. The main ones related to retrieving data are:

- **Web Services** - Often built using protocols like HTTP/HTTPS. Commonly used to enable applications to communicate with each other over the web. For instance, a weather application for a smartphone may use a web API to request weather data from a remote server.

- **Database APIs** - Enable communication between an application and a database, SQL-based calls for example.

Key concepts associated with APIs:

- **Endpoints**: Specific functions available through the API. For web APIs, an endpoint is usually a specific URL where the API can be accessed.
- **Methods**: Actions that can be performed. In web APIs, these often correspond to HTTP methods like GET, POST, PUT, or DELETE.
- **Requests and Responses**: The act of asking the API to perform its function is a *request*. The data it returns is the *response*.
- **Rate Limits**: Restrictions on how often you can call the API, often used to prevent abuse or overloading of the service.
- **Authentication and Authorization**: Mechanisms to ensure that only approved users or applications can use the API. Common methods include *API keys*, *OAuth*, or *Jason Web Tokens* (JWT).
- **Data Formats**: Many web APIs exchange data in a specific format, often JSON or CSV.

- **HTTP: Hyper-Text Transfer Protocol**

- HTTP is the most widely used protocol for data sharing through the internet.
- The **httr2** package provides functions to work with HTTP requests.
- One of the core functions in this package is `request`, which is used to form request to send to web services.
- The `req_perform` function sends the request.
- This `request` function forms an HTTP GET request to the specified URL.

- **HTTP: Hyper-Text Transfer Protocol**

- Typically, HTTP GET requests are used to retrieve information from a server based on the provided URL.

- The function returns an object of class `response`.

- This object contains all the details of the server's response, including status code, headers, and content.

- You can then use other **httr2** functions to extract or interpret information from this response.

**Example:**

Let's say you want to retrieve COVID-19 deaths by state from the CDC. By visiting their data catalog you can search for datasets and find that the data is provided through this API:

https://data.cdc.gov/

```
# install.packages("httr2")
library(httr2)
library(readr)
library(jsonlite)
url <- "https://data.cdc.gov/resource/r8kw-7aab.json"
response <- request(url) |> req_perform()
tab <- response |> resp_body_string() |> fromJSON(flatten=TRUE)

# increase return limit
response <- request(url) |> req_url_path_append("?$limit=100000") |> req_perform()

tab <- response |> resp_body_string() |> fromJSON(flatten = TRUE)
```

When working with APIs, it's essential to check the API's documentation for rate limits, required headers, or authentication methods.

The `httr2`package provides tools to handle these requirements, such as setting headers or authentication parameters.

- When working with APIs, it's essential to check the API's documentation for rate limits, required headers, or authentication methods.
- The `httr2` package provides tools to handle these requirements, such as setting headers or authentication parameters.

Any
Questions