ICO Workshop R & RStudio

Part 3

Data manipulation with `dplyr`

Sven De Maeyer & Tine van Daal

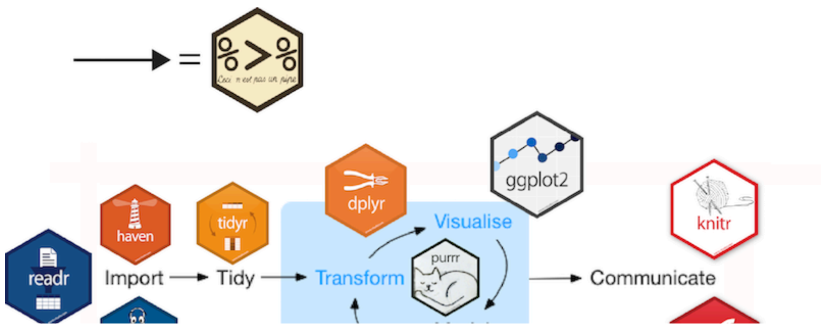
2nd - 4th July, 2024

Overview

- Tidyverse --- (Click here)
- The `dplyr` package --- (Cliick here)
- Implementation --- (Click here)

1. Tidyverse

Welcom in the `tidyverse`



Why tidyverse?

- more accessible for beginners
- consistent approach for all potential tasks
- powerful potential applications mith minimum 'effort'
- can give confidence to explore R

Tibble

Normally we work with a dataframe in R but we can have very complex data-structures as well (e.g., lists, matrices, ...)

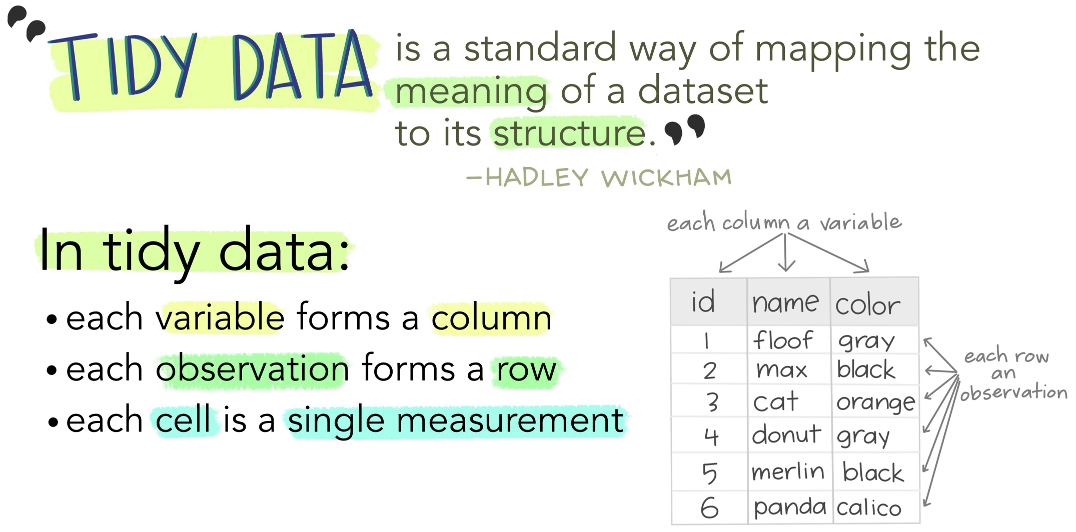
In the tidyverse ecosystem we work with a simple form of data-structure: a tibble

A tibble is a dataframe that fits the **tidy data** principle

```
Friends
```

```
## # A tibble: 108 × 4
##    student occassion condition fluency
##      <dbl>     <dbl>     <dbl>   <dbl>
##  1       1         1         1   101.
##  2       1         2         1   104.
##  3       1         3         1   117.
##  4       2         1         2    98.8
##  5       2         2         2   107.
##  6       2         3         2   111.
##  7       3         1         3   105.
##  8       3         2         3   102.
##  9       3         3         3   101.
## 10       4         1         1   102.
## # ℹ 98 more rows
```

What is **tidy data**?



"TIDY DATA is a standard way of mapping the meaning of a dataset to its structure."

—HADLEY WICKHAM

In tidy data:

- each variable forms a column
- each observation forms a row
- each cell is a single measurement

id	name	color
1	floof	gray
2	max	black
3	cat	orange
4	donut	gray
5	merlin	black
6	panda	calico

Wickham, H. (2014). Tidy Data. Journal of Statistical Software *59* (10). DOI: 10.18637/jss.v059.i10

What is **tidy data**?

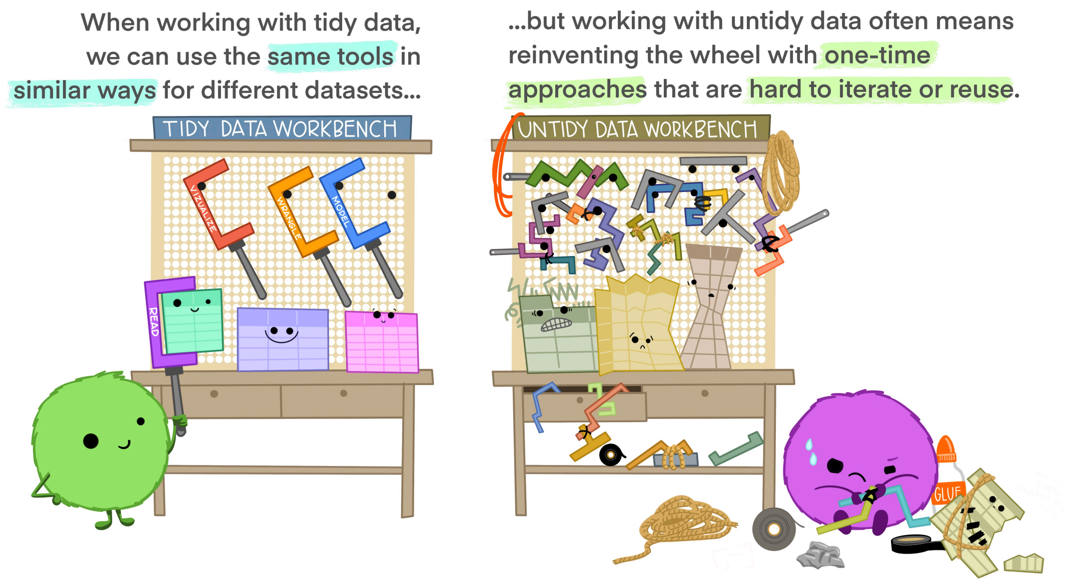
The standard structure of tidy data means that "tidy datasets are all alike..."

"...but every messy dataset is messy in its own way."

—HADLEY WICKHAM

our columns are VARIABLES & our rows are OBSERVATIONS!
my columns are values and my rows are variables
i have variables in columns AND in rows.
i have multiple variables in a single column.
i don't even KNOW what my deal is.

What is **tidy data**?


When working with tidy data, we can use the same tools in similar ways for different datasets...

...but working with untidy data often means reinventing the wheel with one-time approaches that are hard to iterate or reuse.

Artwork by @allison_horst

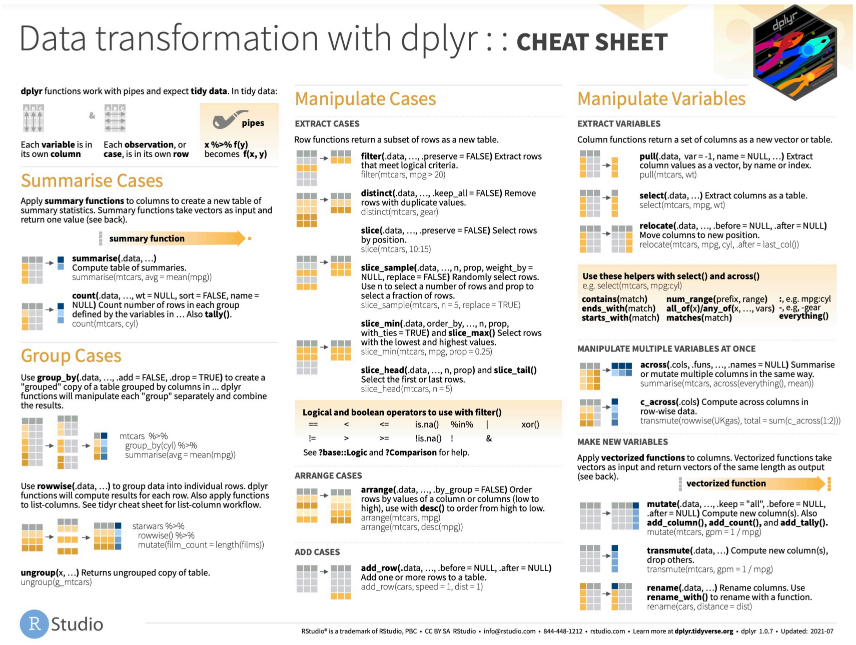
2. The `dplyr` package

`dplyr` ...

is THE package to work with tidy data !

VERBS are at the core:

- `filter()`
- `mutate()`
- `select()`
- `group_by() + summarise()`
- `arrange()`
- `rename()`
- `relocate()`
- `join()`



Data transformation with dplyr : : CHEAT SHEET

dplyr functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own **row**

pipes
x %>% f(y) becomes **f(x, y)**

Summarise Cases

Apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

summary function

summarise(.data, …**)**
Compute table of summaries.
summarise(mtcars, avg = mean(mpg))

count(.data, …, wt = NULL, sort = FALSE, name = NULL**)** Count number of rows in each group defined by the variables in … Also **tally()**.
count(mtcars, cyl)

Group Cases

Use **group_by(**.data, …, .add = FALSE, .drop = TRUE**)** to create a "grouped" copy of a table grouped by columns in ... dplyr functions will manipulate each "group" separately and combine the results.

```
mtcars %>%
  group_by(cyl) %>%
  summarise(avg = mean(mpg))
```

Use **rowwise(**.data, …**)** to group data into individual rows. dplyr functions will compute results for each row. Also apply functions to list-columns. See tidyr cheat sheet for list-column workflow.

```
starwars %>%
  rowwise() %>%
  mutate(film_count = length(films))
```

ungroup(x, …**)** Returns ungrouped copy of table.
ungroup(g_mtcars)

Manipulate Cases

EXTRACT CASES

Row functions return a subset of rows as a new table.

filter(.data, …, .preserve = FALSE**)** Extract rows that meet logical criteria.
filter(mtcars, mpg > 20)

distinct(.data, …, .keep_all = FALSE**)** Remove rows with duplicate values.
distinct(mtcars, gear)

slice(.data, …, .preserve = FALSE**)** Select rows by position.
slice(mtcars, 10:15)

slice_sample(.data, …, n, prop, weight_by = NULL, replace = FALSE**)** Randomly select rows. Use n to select a number of rows and prop to select a fraction of rows.
slice_sample(mtcars, n = 5, replace = TRUE)

slice_min(.data, order_by, …, n, prop, with_ties = TRUE**)** and **slice_max()** Select rows with the lowest and highest values.
slice_min(mtcars, mpg, prop = 0.25)

slice_head(.data, …, n, prop**)** and **slice_tail()** Select the first or last rows.
slice_head(mtcars, n = 5)

Logical and boolean operators to use with filter()

==	<	<=	is.na()	%in%	|	xor()
!=	>	>=	!is.na()	!	&	

See **?base::Logic** and **?Comparison** for help.

ARRANGE CASES

arrange(.data, …, .by_group = FALSE**)** Order rows by values of a column or columns (low to high), use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

ADD CASES

add_row(.data, …, .before = NULL, .after = NULL**)** Add one or more rows to a table.
add_row(cars, speed = 1, dist = 1)

Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.

pull(.data, var = -1, name = NULL, …**)** Extract column values as a vector, by name or index.
pull(mtcars, wt)

select(.data, …**)** Extract columns as a table.
select(mtcars, mpg, wt)

relocate(.data, …, .before = NULL, .after = NULL**)** Move columns to new position.
relocate(mtcars, mpg, cyl, .after = last_col())

Use these helpers with select() and across()
e.g. select(mtcars, mpg:cyl)

contains(match**)**	**num_range(**prefix, range**)**	**:**, e.g. mpg:cyl
ends_with(match**)**	**all_of(**x**)/any_of(**x, …, vars**)**	-, e.g, -gear
starts_with(match**)**	**matches(**match**)**	**everything()**

MANIPULATE MULTIPLE VARIABLES AT ONCE

across(.cols, .funs, …, .names = NULL**)** Summarise or mutate multiple columns in the same way.
summarise(mtcars, across(everything(), mean))

c_across(.cols**)** Compute across columns in row-wise data.
transmute(rowwise(UKgas), total = sum(c_across(1:2)))

MAKE NEW VARIABLES

Apply **vectorized functions** to columns. Vectorized functions take vectors as input and return vectors of the same length as output (see back).

vectorized function

mutate(.data, …, .keep = "all", .before = NULL, .after = NULL**)** Compute new column(s). Also **add_column(), add_count(),** and **add_tally().**
mutate(mtcars, gpm = 1 / mpg)

transmute(.data, …**)** Compute new column(s), drop others.
transmute(mtcars, gpm = 1 / mpg)

rename(.data, …**)** Rename columns. Use **rename_with()** to rename with a function.
rename(cars, distance = dist)

RStudio

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **dplyr.tidyverse.org** • dplyr 1.0.7 • Updated: 2021-07

https://raw.githubusercontent.com/rstudio/cheatsheets/master/data-transformation.pdf

The %>% operator (a 'pipe')

To create
a chain of functions

Instead of

```
mean(c(1,2,3,4))
```

or

```
Numbers <- c(1,2,3,4)
mean(Numbers)
```

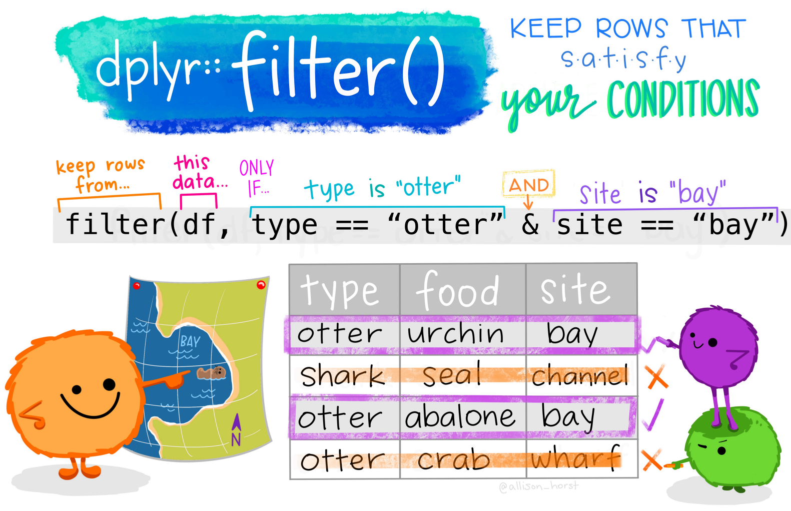
you can do

```
c(1,2,3,4) %>%
  mean( )
```

With the `%>%` you can write a sentence like:

> *I `%>%` woke up `%>%`, took a shower `%>%`, got breakfast `%>%`, took the train `%>%` and arrived at the ICO course `%>%` …*

`filter()`



```
filter(df, type == "otter" & site == "bay")
```

type	food	site
otter	urchin	bay
shark	seal	channel
otter	abalone	bay
otter	crab	wharf

Artwork by @allison_horst

Let's apply `filter()`

With the FRIENDS data:

> *We only select observations from the first measurement occassion in condition 1*

```
Friends_Occ1 <- Friends %>%
  filter(occassion == 1 & condition == 1)
```

`==` is *equals* (notice the 2 = signs!)

> *Let's clean some data, and remove observations with fluency values above 300 and that do not equal fluence value 0*

```
Friends_clean <- Friends %>%
  filter(fluency < 300 & fluency != 0)
```

`!=` means *not equal to*

`mutate()`

Artwork by @allison_horst

Let's apply `mutate()`

With the Friends data:

> *We calculate a new variable containing the fluency scores minus the average of fluency*

```
Friends <- Friends %>%
  mutate(
    fluency_centered = fluency - mean(fluency, na.rm = T)
    )
```

Let's apply `mutate()`

With the Friends data:

> *We create a factor for condition*

```
Friends <- Friends %>%
  mutate(
    condition_factor = as.factor(condition)
  )

str(Friends$condition_factor)
```

```
##  Factor w/ 3 levels "1","2","3": 1 1 1 2 2 2 3 3 3 1 ...
```

Let's apply `select()`

To select variables.

Some examples with the Friends data:

> *We only select `condition` and `occasion` and inspect the result with the `str()`function*

```
Friends %>%
  select(
    condition, occassion
  ) %>%
  str()
```

```
## tibble [108 × 2] (S3: tbl_df/tbl/data.frame)
##  $ condition: num [1:108] 1 1 1 2 2 2 3 3 3 1 ...
##   ..- attr(*, "value.labels")= Named chr [1:3] "3" "2" "1"
##   .. ..- attr(*, "names")= chr [1:3] "No subtitles" "Spanish" "English"
##  $ occassion: num [1:108] 1 2 3 1 2 3 1 2 3 1 ...
##  - attr(*, "variable.labels")= Named chr(0)
##   ..- attr(*, "names")= chr(0)
##  - attr(*, "codepage")= int 1252
```

Rename variables with `rename()`

Notice how the variable `occassion` is misspelled! Pretty enoying when coding... But we can easily rename variables.

Function `rename(new_name = old_name)`

> *Rename the variable `occassion` to `occasion`*

```
Friends <- Friends %>%
  rename(
    occasion = occassion
  )
```

Super combo 1: `group_by() + summarize( )`

- transform a tibble to a *grouped tibble* making use of `group_by()`
- calculate summary stats per group making use of `summarize()`

> *Calculate the average fluency and standard deviation per condition*

```
Friends %>%
  group_by(
    condition
  ) %>%
  summarize(
    mean_fluency = mean(fluency),
    sd_fluency   = sd(fluency)
  )
```

```
## # A tibble: 3 × 3
##   condition mean_fluency sd_fluency
##       <dbl>        <dbl>      <dbl>
## 1         1         109.       9.08
## 2         2         108.       6.02
## 3         3         103.       4.17
```

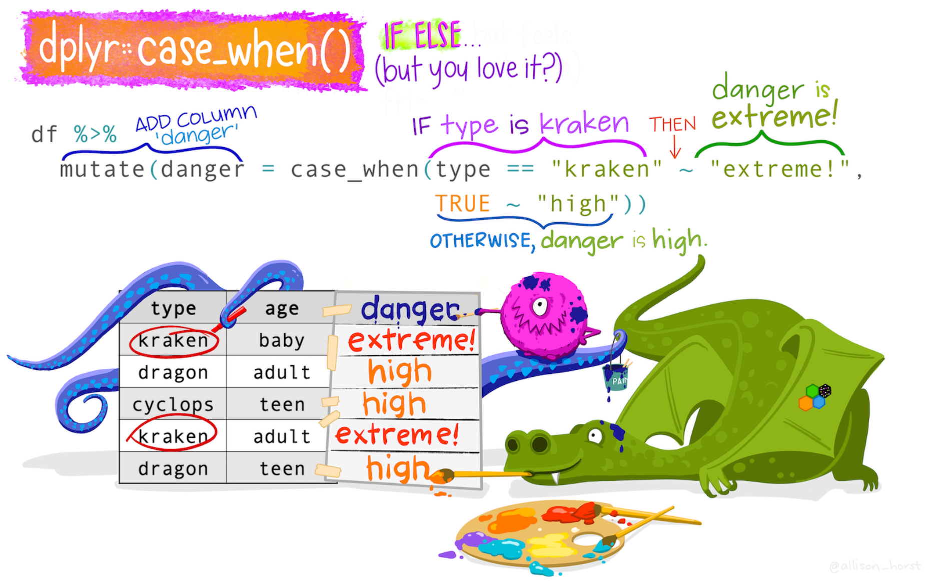
Super combo 1: `group_by() + summarize( )`

> *Calculate the number of observations for each combination of condition and occasion*

```
Friends %>%
  group_by(
    occasion, condition
  ) %>%
  summarize(
    n_observations  = n()
  )
```

```
## # A tibble: 9 × 3
## # Groups:   occasion [3]
##   occasion condition n_observations
##      <dbl>     <dbl>          <int>
## 1        1         1             12
## 2        1         2             12
## 3        1         3             12
## 4        2         1             12
## 5        2         2             12
## 6        2         3             12
## 7        3         1             12
## 8        3         2             12
## 9        3         3             12
```

Super combo 2: `mutate() + case_when( )`



```
df %>%
  mutate(danger = case_when(type == "kraken" ~ "extreme!",
                            TRUE ~ "high"))
```

Artwork by @allison_horst

Super combo 2: `mutate() + case_when( )`

To recode variables into new variables!

> *We create a new categorical variant of fluency with 3 groups, then we select this new variable and have a look to the top 5 observations...*

```
Friends %>%
  mutate(
    fluency_grouped = case_when(
      fluency < 106.625 - 7.1 ~ 'low',
      fluency >= 106.625 - 7.1 & fluency < 106.625 + 7.1 ~ 'aver
      fluency >= 106.625 + 7.1 ~ 'high'
    )
  ) %>%
  select(
    fluency,
    fluency_grouped
    ) %>%
  head(5)
```

```
## # A tibble: 5 × 2
##   fluency fluency_grouped
##     <dbl> <chr>
## 1   101.  average
## 2   104.  average
## 3   117.  high
## 4    98.8 low
## 5   107.  average
```

How to define conditions

- `x == y` $\rightarrow$ 'x is equal to y'
- `x != y` $\rightarrow$ 'x is NOT equal to y'
- `x < y` $\rightarrow$ 'x is smaller than y'
- `x <= y` $\rightarrow$ 'x is smaller or equal to y'
- `x > y` $\rightarrow$ 'x is higher than y'
- `x >= y` $\rightarrow$ 'x is higher or equal to y'

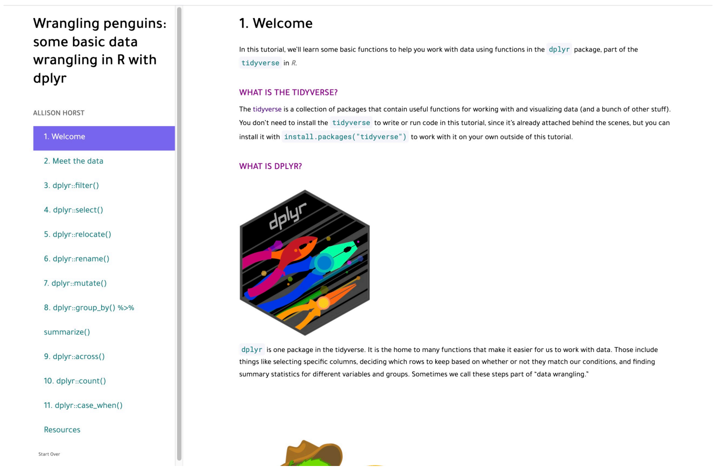
Bolean operators

We can combine conditions!

- `&` $\rightarrow$ 'and' $\rightarrow$ example: `gender == 1 & age <=18`
- `|` $\rightarrow$ 'or' $\rightarrow$ example: `gender == 1 | gender == 2`
- `!` $\rightarrow$ 'not' $\rightarrow$ example: `gender == 1 & !age <=18`

Interactive tutorial about `dplyr()`

If you want some more material and a place to exercise your skills? This online and freetutorial (made with the package `learnr`) is strongly advised!



Wrangling penguins: some basic data wrangling in R with dplyr

ALLISON HORST

1. Welcome
2. Meet the data
3. dplyr::filter()
4. dplyr::select()
5. dplyr::relocate()
6. dplyr::rename()
7. dplyr::mutate()
8. dplyr::group_by() %>% summarize()
9. dplyr::across()
10. dplyr::count()
11. dplyr::case_when()

Resources

Start Over

1. Welcome

In this tutorial, we'll learn some basic functions to help you work with data using functions in the `dplyr` package, part of the `tidyverse` in *R*.

WHAT IS THE TIDYVERSE?

The tidyverse is a collection of packages that contain useful functions for working with and visualizing data (and a bunch of other stuff). You don't need to install the `tidyverse` to write or run code in this tutorial, since it's already attached behind the scenes, but you can install it with `install.packages("tidyverse")` to work with it on your own outside of this tutorial.

WHAT IS DPLYR?

`dplyr` is one package in the tidyverse. It is the home to many functions that make it easier for us to work with data. Those include things like selecting specific columns, deciding which rows to keep based on whether or not they match our conditions, and finding summary statistics for different variables and groups. Sometimes we call these steps part of "data wrangling."

https://allisonhorst.shinyapps.io/dplyr-learnr/#section-welcome

Exercise `dplyr`

- You can find the qmd-file `Exercises_dplyr.qmd` in the Exercises folder (you created the project yesterday!) (Exercises > Exercise2_dplyr)
- Open this document
- You get a set of tasks with empty code blocks to start coding
- Write and test the necessary code
- Stuck? No Worries!
    - We are there
    - Help each other
    - There is a solution key (`Exercises_dplyr_solutions.qmd`)