Session 3: Data Wrangling

R for Data Analysis

DIME Analytics
The World Bank | WB Github

April 2025

Table of contents

Introduction

Introduction

Initial Setup

If You Created an RStudio Project in Session 2 | If You Did Not Attend Session 2

1. Go to the `dime-r-training-main` folder that you created yesterday, and open the `dime-r-training-main` R project that you created there.

Introduction

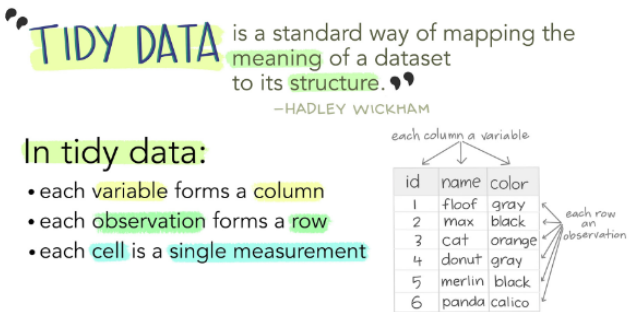
Goals of this session

- To organize data in a manner that makes it easier to analyze and communicate.

Things to keep in mind

- We'll take you through the same steps we've taken when we were preparing the datasets used in this course.
- In most cases, your datasets won't be `tidy`.

> **Tidy data**: A dataset is said to be tidy if it satisfies the following conditions:



Therefore, messy data is any other arrangement of the data.

Introduction

In this session, you'll be introduced to some basic concepts of data cleaning in R. We will cover:

🏗️ What We'll Cover:

- ✅ **Exploring a dataset**
- ✅ **Creating new variables**
- ✅ **Filtering & subsetting**
- ✅ **Merging datasets**
- ✅ **Handling factor variables**
- ✅ **Saving your cleaned data**

> 🔍 **Note:** Data cleaning involves many more tasks beyond this session's scope, but these core techniques will give you a solid foundation! 🚀

Introduction

✅ Getting Ready

Before we begin, let's make sure everything is set up properly:

1️⃣ **Start a fresh RStudio session** – Close and reopen RStudio to avoid any conflicts.

2️⃣ **Open your project** – Navigate to the **RStudio project** you created yesterday.

3️⃣ **Create a new script** – In RStudio, go to:

- `File` ➡️ `New File` ➡️ `R Script`
- Name it **`exercises-session3.R`**

Now, let’s dive into data cleaning! 💡📊

Introduction: R Packages

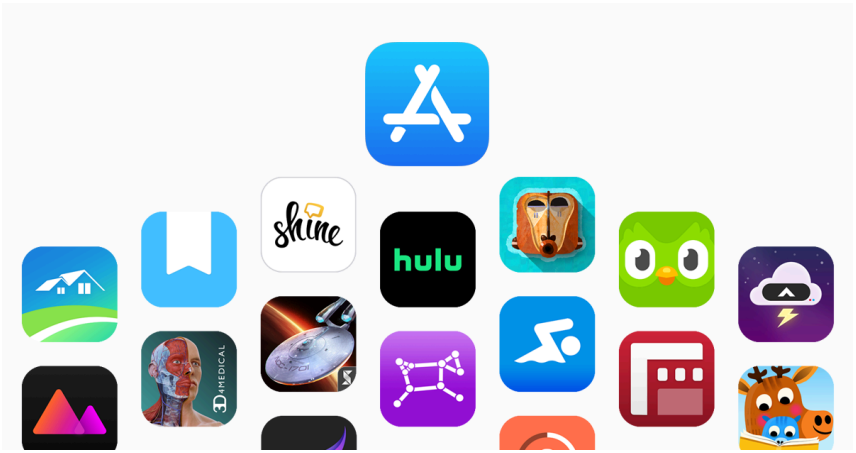
One of R's greatest strengths is its **community-driven ecosystem of packages**.

R is so powerful because thousands of people worldwide contribute by developing and sharing **useful packages**.

> 💡 **Think of R packages like apps for your phone**—they add new features and functionalities to R!

R is a new phone

R packages are apps on your phone



RECAP: Packages

To install a package you can run the following command:

```
# To install
install.packages("dplyr")
```

- Unlike Stata, R packages need to be loaded in each R session that will use them.
- That means that, for example, a function that comes from the `dplyr` package cannot be used if the package has not been installed and loaded first.

To load a package you can run the following command:

```
# To load
library(dplyr)
```

RECAP: loading packages

```
# If you haven't installed the packages uncomment the next line
# install.packages("tidyverse")
# install.packages("here")
# install.packages("janitor")
library(tidyverse)  # To wrangle data
library(here)       # A package to work with relative file paths
library(janitor)    # Additional data cleaning tools
```

```
## Warning: package 'janitor' was built under R version 4.4.3
```

> *Notes*: Remember you should always load your packages before your start coding.

RECAP: File paths

The `here` package allows you to interact with your working directory. It will look for the closest R Project and set its location as the working directory. That's why it is important to set your RStudio project correctly.

The goal of this package is to:

- Easily reference your files in project-oriented workflows.

Using `here`:

- Load the library.
- Use `here()` for relative file paths.

```
path <- here("data", "raw", "data-file.csv")
df <- read.csv(path)
```

RECAP: Loading a Dataset in R

Before we start working with our data, let's first load our dataset.

In R, we can use:

- `read.csv()` (Base R)
- `read_csv()` (from the `readr` package) – preferred for faster and more efficient reading of CSV files.

For this exercise, we will use the **World Happiness Report (WHR) datasets from 2015-2018**.

RECAP: Loading a Dataset in R

Exercise 1: Loading data using the `here` package

Load the three WHR datasets from the folder:

📂 `DataWork/DataSets/Raw/Un WHR`

Name each dataset using the format `whrYY`, where `YY` represents the last two digits of the year. For example, `WHR2015.csv` should be named `whr15`.

00:45

RECAP: Loading a Dataset in R

Solution:

```
whr15 <- read_csv(here("DataWork", "DataSets", "Raw", "Un WHR", "WHR2015.csv"))
whr16 <- read_csv(here("DataWork", "DataSets", "Raw", "Un WHR", "WHR2016.csv"))
whr17 <- read_csv(here("DataWork", "DataSets", "Raw", "Un WHR", "WHR2017.csv"))
```

RECAP: The pipe %>% (or |>) operator

- "Piping" in R can be seen as "**chaining**" functions together.
- Think of %>% as the words "...and then..."

✅ Using the Pipe

```
Mer %>%
  wake_up(time = "5:30") %>%
  get_out_of_bed() %>%
  do_exercise() %>%
  shower() %>%
  get_dressed() %>%
  eat(meal = "breakfast", coffee = TRUE) %>%
  brush_teeth() %>%
  work(effort = "mininum")
```

❌ Without the Pipe

```
work(
  brush_teeth(
    eat(
      get_dressed(
        shower(
          do_exercise(
            get_out_of_bed(
              wake_up(Mer, time = "5:30")
            ),
          )
        )
      ), meal = "breakfast", coffee = TRUE
    )
  ), effort = "minimum")
```

RECAP: The pipe %>% operator

From R for Data Science by Wickham & Grolemund:

> Pipes are a powerful tool for clearly expressing a sequence of multiple operations. The point of the pipe is to help you write code in a way that is easier to read and understand. [...] It focusses on verbs, not nouns. You can read this series of function compositions like it's a set of imperative actions.

(only for 🤓 nerds:)

- The `%>%` pipe is part of the `magrittr` package. `R v4.1.0` adds a *native pipe* via `|>`. you could use it like

```
whr15|> mean(variable, na.rm = T)
```

📦 The `janitor` Package: Clean Column Names Easily

- Sometimes datasets have **messy column names** (e.g., inconsistent capitalization, spaces, or special characters).
- The `clean_names()` function (from the `janitor` package) automatically **fixes** variable names into a **consistent, lowercase, and snake_case format**.

> **Tip**: You can pipe `clean_names()` right after loading a dataset!

```
whr15 <- whr15 %>%
  clean_names()
whr16 <- whr16 %>%
  clean_names()
whr17 <- whr17 %>%
  clean_names()
```

If we want to to rename our variable manually (not recommended), we could use:

```
whr15 <- whr15 %>%
  rename(
    var_newname = var_oldname
  )
```

Exploring your data

🔍 Exploring a Dataset: First Steps in Data Wrangling

📌 Why is this important?

Before analyzing or wrangling data, we need to **understand** what we're working with!
Here are some **essential functions** to explore datasets in R.

🛠 Exploring Data with Base R

These functions help you **inspect your dataset**:

```
View(data)      # Open the dataset in a viewer
class(data)     # Object type (data frame, list, etc.)
dim(data)       # Number of rows & columns
names(data)     # Variable names
str(data)       # Structure of the dataset
summary(data)   # Summary statistics
head(data)      # First few observations
tail(data)      # Last few observations
```

🏗 Exploring Data with the Tidyverse

If you're using the tidyverse, this function is super useful:

```
glimpse(data)   # Compact overview of dataset
```

✓ More readable than str() ✓ Displays variable types and first few values

Load and show a dataset

We can just show our dataset using the name of the object; in this case, `whr15`.

```
whr15
```

```
## # A tibble: 158 × 12
##    country     region                  happiness_rank happiness_score standard_error
##    <chr>       <chr>                            <dbl>           <dbl>          <dbl>
##  1 Switzerland Western Europe                       1            7.59         0.0341
##  2 Iceland     Western Europe                       2            7.56         0.0488
##  3 Denmark     Western Europe                       3            7.53         0.0333
##  4 Norway      Western Europe                       4            7.52         0.0388
##  5 Canada      North America                        5            7.43         0.0355
##  6 Finland     Western Europe                       6            7.41         0.0314
##  7 Netherlands Western Europe                       7            7.38         0.0280
##  8 Sweden      Western Europe                       8            7.36         0.0316
##  9 New Zealand Australia and New …                  9            7.29         0.0337
## 10 Australia   Australia and New …                 10            7.28         0.0408
## # ℹ 148 more rows
## # ℹ 7 more variables: economy_gdp_per_capita <dbl>, family <dbl>,
## #   health_life_expectancy <dbl>, freedom <dbl>,
## #   trust_government_corruption <dbl>, generosity <dbl>,
```

Glimpse your data

Use `glimpse()` to get information about your variables (e.g., type, row, columns,)

```
whr15 %>%
  glimpse()
```

```
## Rows: 158
## Columns: 12
## $ country                    <chr> "Switzerland", "Iceland", "Denmark", "Norw…
## $ region                     <chr> "Western Europe", "Western Europe", "Weste…
## $ happiness_rank             <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,…
## $ happiness_score            <dbl> 7.587, 7.561, 7.527, 7.522, 7.427, 7.406, …
## $ standard_error             <dbl> 0.03411, 0.04884, 0.03328, 0.03880, 0.0355…
## $ economy_gdp_per_capita     <dbl> 1.39651, 1.30232, 1.32548, 1.45900, 1.3262…
## $ family                     <dbl> 1.34951, 1.40223, 1.36058, 1.33095, 1.3226…
## $ health_life_expectancy     <dbl> 0.94143, 0.94784, 0.87464, 0.88521, 0.9056…
## $ freedom                    <dbl> 0.66557, 0.62877, 0.64938, 0.66973, 0.6329…
## $ trust_government_corruption <dbl> 0.41978, 0.14145, 0.48357, 0.36503, 0.3295…
## $ generosity                 <dbl> 0.29678, 0.43630, 0.34139, 0.34699, 0.4581…
## $ dystopia_residual          <dbl> 2.51738, 2.70201, 2.49204, 2.46531, 2.4517…
```

Dimensions of your data

Dimensions of your data

Let's see first how many columns and observations the dataset has:

- **Dimensions of your data** (Rows and Columns):

```
dim(whr15)
```

```
## [1] 158  12
```

- **The number of distinct values of a particular variable**:

```
n_distinct(DATASET$variable, na.rm = TRUE)
```

The `$` sign is a subsetting operator. In R, we have three subsetting operators (`[[`, `[`, and `$`.). It is often used to access variables in a dataframe.

The `n_distinct` function allows us to count the number of unique values of a variable length of a vector. We included `na.rm = TRUE`, so we don't count missing values.

Dimensions of your data

Exercise 2: Identify distinct values of a variable in a dataset. Using the `n_distinct` function, can you tell how many unique values these variables in the `whr15` dataset have?

1. Country
2. Region

Solution:

```
n_distinct(whr15$country, na.rm = TRUE)
```

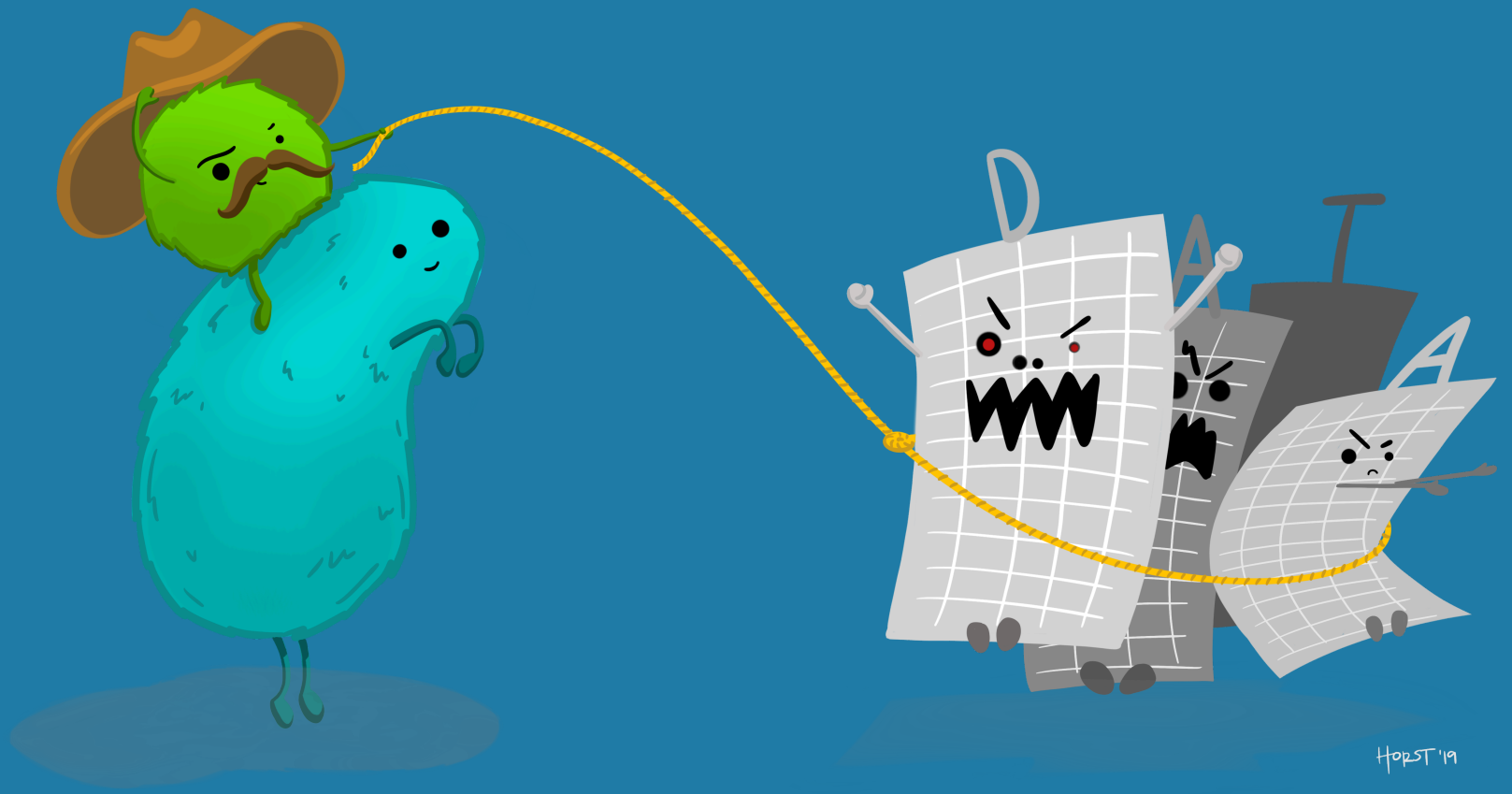
```
## [1] 158
```

```
n_distinct(whr15$region, na.rm = TRUE)
```

```
## [1] 10
```

01:00

Wrangling your data



Wrangling vs Cleaning

Cleaning:

- Detecting and addressing inconsistencies in a dataset. Removing erroneous data from your data.

Wrangling:

- Translating raw data into a more useful form. Unifying messy and complex data.

Wrangling your data: dplyr

For data wrangling you will frequently use one tidyverse package called dplyr.

- `dplyr` is part of the `tidyverse` package family.
- You are ***highly encouraged*** to read through Hadley Wickham's chapter. It's clear and concise.
- Also check out this great "cheatsheet" here.
- During this session, I will show you the most popular functions of the `dplyr` package.

Wrangling your data: dplyr

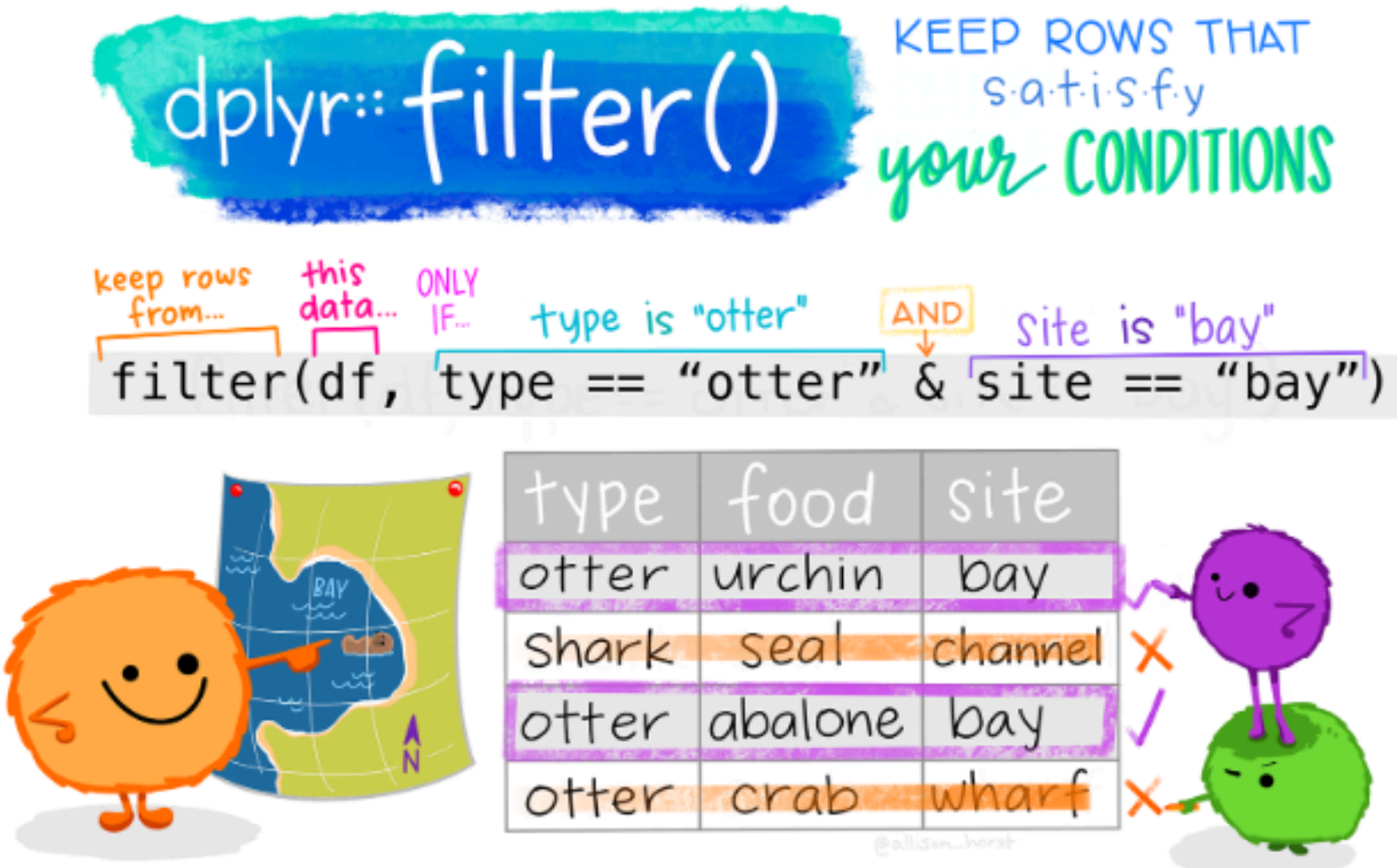
- The package `dplyr` (and the rest of the Tidyverse) is organized around a set of **verbs**, i.e. *actions* to be taken.
- We operate on `data.frames`.
- All *verbs* work as follows:

$$\text{verb}(\underbrace{\text{data.frame}}_{\text{1st argument}}, \underbrace{\text{what to do}}_{\text{2nd argument}})$$

- Alternatively you can (should) use the `pipe` operator `%>%`:

$$\underbrace{\text{data.frame}}_{\text{1st argument}} \ \underbrace{\%>\%}_{\text{"pipe" operator}} \ \text{verb}(\underbrace{\text{what to do}}_{\text{2nd argument}})$$

Filtering and sorting (dplyr::filter)



Filtering and sorting dplyr::filter

- The `filter` function is used to subset rows in a dataset.

```
whr15 %>% filter(region == "Western Europe")
```

```
## # A tibble: 21 × 12
##    country     region         happiness_rank happiness_score standard_error
##    <chr>       <chr>                   <dbl>           <dbl>          <dbl>
##  1 Switzerland Western Europe              1            7.59         0.0341
##  2 Iceland     Western Europe              2            7.56         0.0488
##  3 Denmark     Western Europe              3            7.53         0.0333
##  4 Norway      Western Europe              4            7.52         0.0388
##  5 Finland     Western Europe              6            7.41         0.0314
##  6 Netherlands Western Europe              7            7.38         0.0280
##  7 Sweden      Western Europe              8            7.36         0.0316
##  8 Austria     Western Europe             13            7.2          0.0375
##  9 Luxembourg  Western Europe             17            6.95         0.0350
## 10 Ireland     Western Europe             18            6.94         0.0368
## # ℹ 11 more rows
## # ℹ 7 more variables: economy_gdp_per_capita <dbl>, family <dbl>,
## #   health_life_expectancy <dbl>, freedom <dbl>,
## #   trust_government_corruption <dbl>, generosity <dbl>,
```

Filtering and sorting dplyr::filter

Exercise 3: Use `filter()` to extract rows in these regions: (1) Eastern Asia and (2) North America. Hint: use the **or** operator (`|`):

Solution:

```
whr15 %>%
  filter(region == "Eastern Asia" | region == "North America")
```

A more elegant approach would be to use the `%in%` operator (equivalent to `inlist()` in Stata):

```
whr15 %>%
  filter(region %in% c("Eastern Asia", "North America"))
```

01:00

Filtering and sorting missing cases

If you want to remove (or identify) missing cases for a specific variable, you can use `is.na()`.

- This function returns a value of true and false for each value in a data set.
- If the value is *NA* the `is.na()` function returns TRUE, otherwise, it returns FALSE.
- In this way, we can check NA values that can be used for other functions.
- We can also negate the function using `!is.na()` which indicates that we want to return those observations with no missing values in a variable.

The function syntax in a pipeline is as follows:

```
DATA %>%
  filter(
    is.na(VAR)
  )
```

What are we returning here?

The observations with missing values for the variable VAR.

Filtering and sorting missing cases

Let's try filtering the `whr15` data. Let's keep those observations that have information per region, i.e., no missing values.

```
whr15 %>%
  filter(!is.na(region)) %>%
  head(5)
```

```
## # A tibble: 5 × 12
##   country     region         happiness_rank happiness_score standard_error
##   <chr>       <chr>                   <dbl>           <dbl>          <dbl>
## 1 Switzerland Western Europe              1            7.59         0.0341
## 2 Iceland     Western Europe              2            7.56         0.0488
## 3 Denmark     Western Europe              3            7.53         0.0333
## 4 Norway      Western Europe              4            7.52         0.0388
## 5 Canada      North America               5            7.43         0.0355
## # ℹ 7 more variables: economy_gdp_per_capita <dbl>, family <dbl>,
## #   health_life_expectancy <dbl>, freedom <dbl>,
## #   trust_government_corruption <dbl>, generosity <dbl>,
## #   dystopia_residual <dbl>
```

> Notice that we are negating the function, i.e., `!is.na()`
> In case we want to keep the observations that contains missing information we will only use `is.na()`.

Filtering

We can use the `%in%` operator to test if values belong to a vector. To **filter out** values, we use the **negation** `!%in%`.

For instance, if we want to exclude the regions "Eastern Asia" and "North America":

```
whr15 %>%
  filter(!region %in% c("Eastern Asia", "North America")) %>%
  head(5)
```

```
## # A tibble: 5 × 12
##   country     region         happiness_rank happiness_score standard_error
##   <chr>       <chr>                   <dbl>           <dbl>          <dbl>
## 1 Switzerland Western Europe              1            7.59         0.0341
## 2 Iceland     Western Europe              2            7.56         0.0488
## 3 Denmark     Western Europe              3            7.53         0.0333
## 4 Norway      Western Europe              4            7.52         0.0388
## 5 Finland     Western Europe              6            7.41         0.0314
## # ℹ 7 more variables: economy_gdp_per_capita <dbl>, family <dbl>,
## #   health_life_expectancy <dbl>, freedom <dbl>,
## #   trust_government_corruption <dbl>, generosity <dbl>,
## #   dystopia_residual <dbl>
```

This filters the data to show only rows where `region` is **not** in the specified list.

Creating new variables (dplyr::mutate)

Creating new variables (dplyr::mutate)

- `mutate` will take a statement like this:

```
mutate(variable_name = some_calculation)
```

- And attach variable_name at the end of the dataset.

For example:

```
whr15 %>%
  mutate(
    hap_hle = happiness_score * health_life_expectancy
  )
```

This will add a new variable called `hap_hle` which is the interaction of happiness score and health life expectancy.

Creating new variables: Dummy variables

```
whr15 %>%
  mutate(happiness_score_6 = (happiness_score > 6))
```

What do you think is happening to this variable?

This new variables contains either `TRUE` or `FALSE`. To have it as a numeric variable (1 or 0, respectively), we include the `as.numeric()` function.

```
whr15 %>%
  mutate(happiness_score_6 = as.numeric((happiness_score > 6)))
```

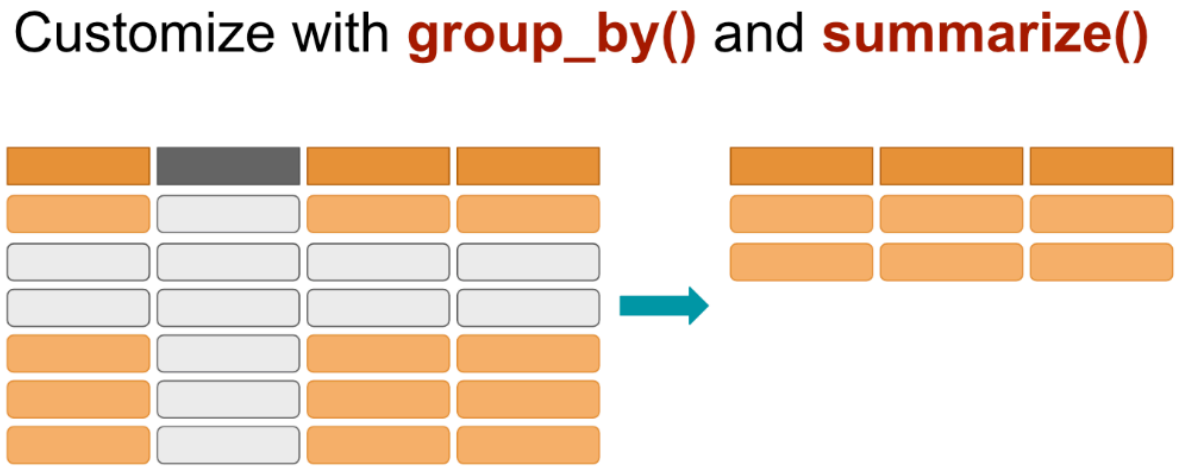
Finally, instead of using a random number, such as 6, we can do the following:

```
whr15 %>%
  mutate(
    happiness_high_mean = as.numeric((happiness_score > mean(happiness_score, na.rm = TRUE)))
  )
```

Creating variables by groups group_by() and

We use this when we want to aggregate your data (by groups).

This is one of the most commons operations. Sometimes we want to calculate statistics by groups



Creating variables by groups group_by() and

In R, we can use `dplyr::group_by()` before we mutate to group an estimation. For example, we are going to pipe the following functions:

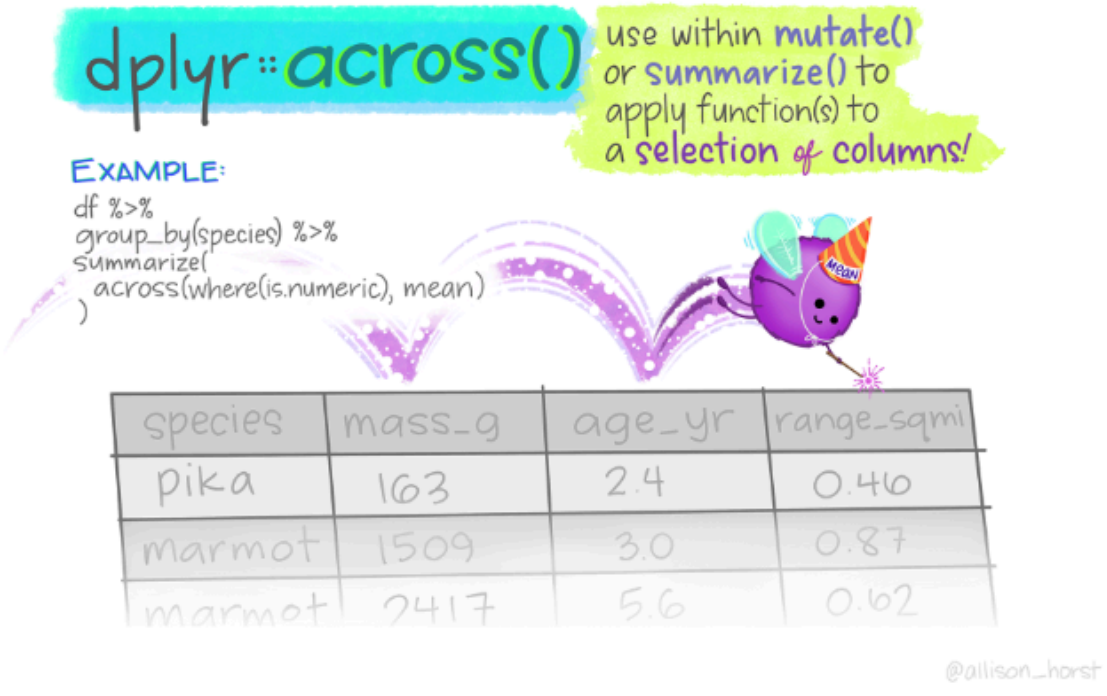
1. Group our data by the `region` variable.
2. Create a variable that would be the mean of `happiness_score` by each region.
3. Select the variables `country, region, happiness_score, mean_hap`.

Example | With variables | Output

```
DATASET %>%
  group_by(GROUPING VARIABLE) %>%
  mutate(
    NAME OF NEW VAR = mean(VARIABLE, na.rm = TRUE)
  ) %>%
  select(VAR1, VAR2, VAR3, VAR4)
```

Creating multiple variables at the same time

We can create multiple variables in an easy way. Let's imagine that we want to estimate the mean value for the variables: `happiness_score`, `health_life_expectancy`, and `trust_government_corruption` .



Creating multiple variables at the same time

How we can do it?

- We can use the function `across()`.
  Syntax: `across(VARS that you want to transform, FUNCTION to execute)`.
- `across()` should be always use inside `summarise()` or `mutate()`.

Across | Output

```
vars <- c("happiness_score", "health_life_expectancy", "trust_government_corruption")

whr15 %>%
  group_by(region) %>%
  summarize(
    across(
      all_of(vars), mean
    )
  ) %>%
  head(3)
```

Creating variables

01:00

Exercise 4: Create a variable called `year` that equals to the year of each dataframe using the `mutate()`. Remember to assign it to the same dataframe.

Solution:

```
whr15 <- whr15 %>%
  mutate(year = 2015)

whr16 <- whr16 %>%
  mutate(year = 2016)

whr17 <- whr17 %>%
  mutate(year = 2017)
```

Appending and merging data sets

Appending dataframes

Now that we can identify the observations, we can combine the data set. Here are two functions to append objects by row

```
rbind(df1, df2, df3) # The base R function

bind_rows(df1, df2, df3) # The dplyr function, making some improvements to base R
```

Exercise 5: Append data sets. Use the function `bind_rows` to append the three WHR datasets:

Solution:

```
bind_rows(whr15, whr16, whr17)
```

Notes

- One of the problems with binding rows like this is that, sometimes, some columns are not *fully* compati

00:45

Appending and merging data sets

Exercise 6: Fixing our variables and appending the data frames correctly.

Exercise 6a

- Load the R data set `regions.RDS` from `DataWork/DataSets/Raw/Un WHR` using the `read_rds` function.

Solution:

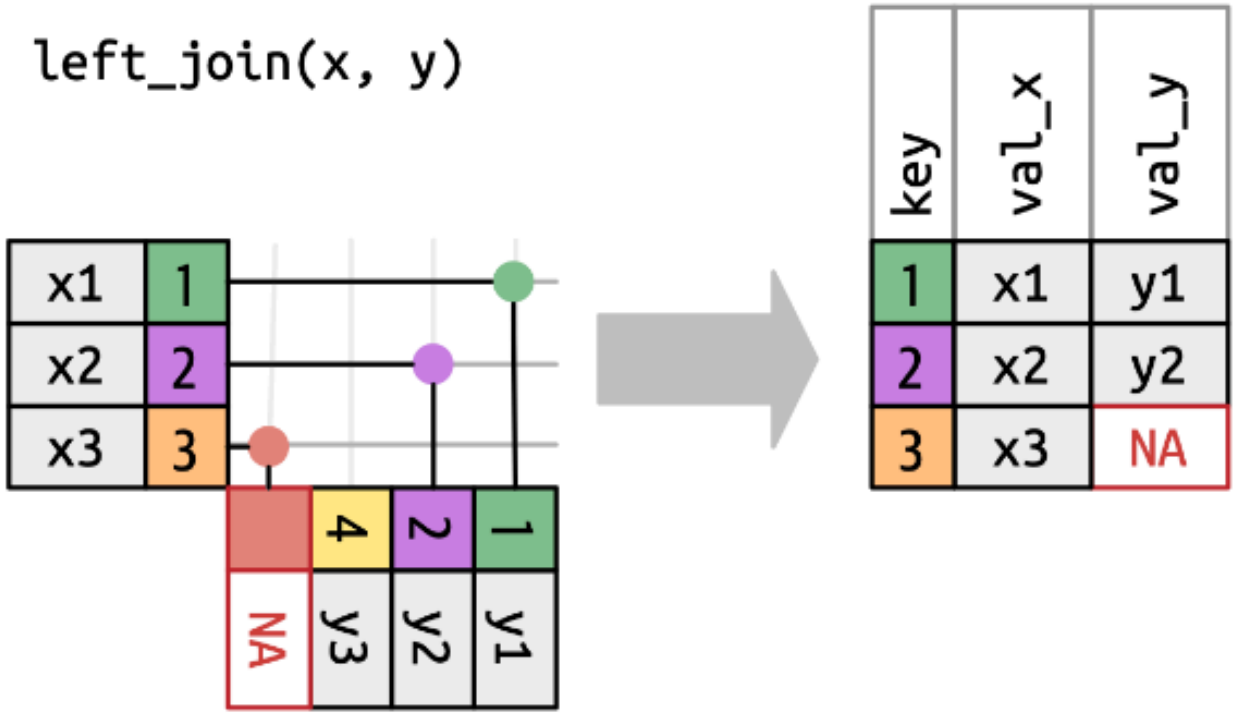
```
regions <- read_rds(here("DataWork", "DataSets", "Raw", "Un WHR", "regions.RDS"))
```

00:45

Appending and merging data sets

We can use the `dplyr::left_join()` function to merge two dataframes. The function syntax is: `left_join(a_df, another_df, by = c("id_col1"))`.

> A left join takes all the values from the first table, and looks for matches in the second table. If it finds a match, it adds the data from the second table; if not, it adds missing values. It is the equivalent of `merge, keep(master matched)` in Stata.



Appending and merging data sets

Exercise 6b: Join the dataframes: `regions` and `whr17`.

Solution:

```
whr17 <- whr17 %>%
  left_join(regions, by = "country") %>%
  select(country, region, everything())
```

Notes:

Look at the `everything()` function. It takes all the variables from the dataframe and put them after country and region. In this way, select can be use to **order** columns!

01:00

Appending and merging data sets

01:00

Exercise 6c: Check if there is any other countries in `whr17` without region info:

- Only use pipes %>%
- And `filter()`
- Do not assign it to an object.

Solution:

```
whr17 %>%
  filter(is.na(region))
```

```
## # A tibble: 2 × 14
##   country         region happiness_rank happiness_score whisker_high whisker_low
##   <chr>           <chr>           <dbl>           <dbl>        <dbl>       <dbl>
## 1 Taiwan Provinc… <NA>               33            6.42         6.49        6.35
## 2 Hong Kong S.A.… <NA>               71            5.47         5.55        5.39
## # ℹ 8 more variables: economy_gdp_per_capita <dbl>, family <dbl>,
## #   health_life_expectancy <dbl>, freedom <dbl>, generosity <dbl>,
## #   trust_government_corruption <dbl>, dystopia_residual <dbl>, year <dbl>
```

So we ended up with two countries with NAs

This is due to the name of the countries. The regions dataset doesn't have "Taiwan Province of China" nor "Hong Kong S.A.R., China" but "Taiwan" and "Hong Kong."

How do you think we should solve this?

- My approach would be to:

1. fix the names of these countries in the `whr17` dataset (a data cleaning task) and;
2. merge (left_join) it with the regions dataset.

Appendix: case_when and mutate for more information.

Appending and merging data sets

01:00

Finally, let's keep those relevant variables first and bind those rows.

Exercise 7: Bind all rows and create a panel called: `whr_panel`.

- Select the variables: `country`, `region`, `year`, `happiness_rank`, `happiness_score`, `economy_gdp_per_capita`, `health_life_expectancy`, `freedom` for each df, i.e., `whr15`, `whr16`, `whr17`.
- Use `rbind()`

Solution:

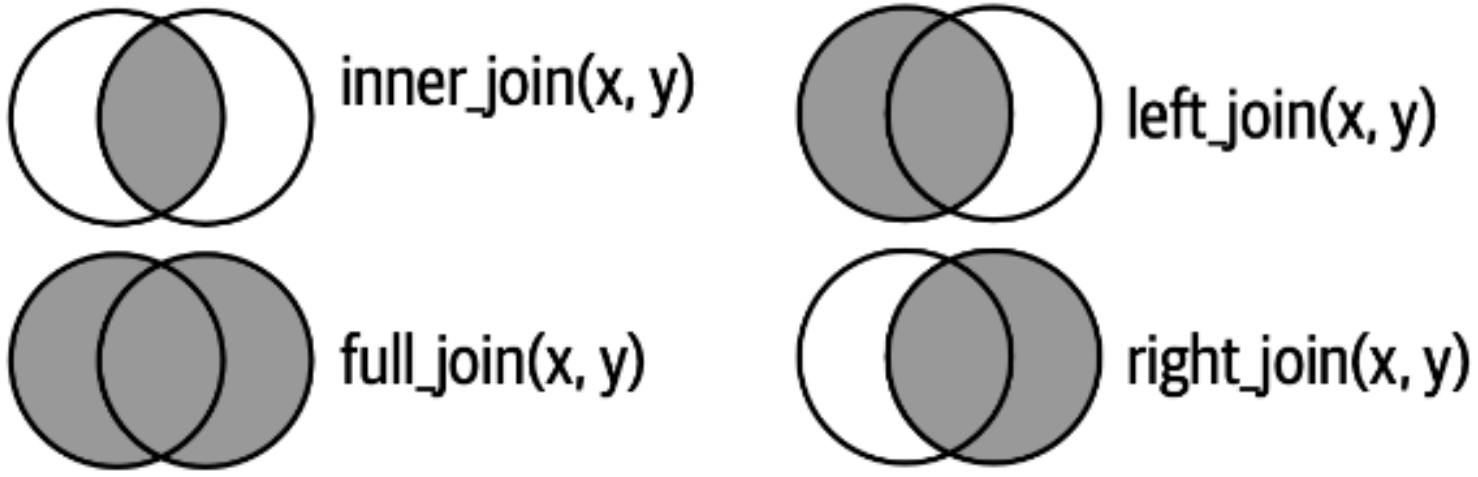
```
vars_to_keep <- c("country", "region", "year", "happiness_rank",
                  "happiness_score", "economy_gdp_per_capita",
                  "health_life_expectancy", "freedom")

whr15 <- select(whr15, all_of(vars_to_keep))
whr16 <- select(whr16, all_of(vars_to_keep))
whr17 <- select(whr17, all_of(vars_to_keep))

whr_panel <- rbind(whr15, whr16, whr17)    # or bind_rows
```

Appending and merging data sets

There are other types of joins in the `dplyr` package. We won't get into detail, but here are some examples.



You can also check this chapter, which is very clear.

More wrangling operations

These were two examples we chose to show different possible data wrangling operations. A summary of these and other common operations are:

Operation	Function in `dplyr`
Subset columns	`select()`
Subset rows (based on condition)	`filter()`
Create new columns	`mutate()`
Create new columns based on condition	`mutate()` and `case_when()`
Create new rows	`add_row()`
Merge dataframes	`inner_join()`, `left_join()`, `right_join()`, `full_join()`
Append dataframes	`bind_rows()`
Deduplicate	`distinct()`
Collapse and create summary indicators	`group_by()`, `summarize()`
Pass a result as the first argument for the next function	`%>%` (operator, not function (**tomorrow**))

Saving a dataset

Saving a dataset

- The dataset you have is the same data set we've been using for earlier sessions, so we can save it now.
- To save a dataset we can use the `write_csv` function from the `readr` package, or `write.csv` from base R.

The function takes the following syntax:

`write_csv(x, file, append = FALSE, row.names = FALSE, na = "")`:

- `x:` the object (usually a data frame) you want to export to CSV
- `file:` the file path to where you want to save it, including the file name and the format (".csv")

Saving a dataset

01:00

Exercise 8: Save the dataset as csv format in the "Final" folder with the name `whr_panel_**YOUR INITIALS**.csv`

- Use `write_csv()`
- Use `here()`

Solution:

```
write_csv(
  whr_panel, here("DataWork", "DataSets", "Final", "whr_panel_MA.csv")
)
```

- The problem with CSVs is that they cannot differentiate between `strings` and `factors`
- They also don't save factor orders
- Data attributes (which are beyond the scope of this training, but also useful to document data sets) are also lost.

Saving a dataset

The R equivalent of a `.dta` file is a `.rds` file. It can be saved and loaded using the following commands:

- `write_rds(object, file = "")`: Writes a single R object to a file.
- `read_rds(file)`: Load a single R object from a file.

```
# Save the data set

write_rds(
  whr_panel,
  here("DataWork", "DataSets", "Final", "whr_panel_MA.Rds")
)
```

And that's it for this session. Join us next week!!

Appendix

Missing values in R

Quick Note:

- Missings in R are treated differently than in Stata. They are represented by the NA symbol.
- Impossible values are represented by the symbol NaN which means 'not a number.'
- R uses the same symbol for character and numeric data.
- NA is not a string or a numeric value, but an indicator of missingness.
- NAs are contagious. This means that if you compare a number with NAs you will get NAs.
- Therefore, always remember the `na.rm = TRUE` argument if needed.

Other relevant functions: slice, subset, select

Arrange | Slice | Select | Combining functions

`Arrange`: allows you to order by a specific column.

```
whr15 %>%
  arrange(region, country) %>%
  head(5)
```

```
## # A tibble: 5 × 8
##   country     region  year happiness_rank happiness_score economy_gdp_per_capita
##   <chr>       <chr>  <dbl>          <dbl>           <dbl>                  <dbl>
## 1 Australia   Austr…  2015             10            7.28                  1.33
## 2 New Zealand Austr…  2015              9            7.29                  1.25
## 3 Albania     Centr…  2015             95            4.96                  0.879
## 4 Armenia     Centr…  2015            127            4.35                  0.768
## 5 Azerbaijan  Centr…  2015             80            5.21                  1.02
## # ℹ 2 more variables: health_life_expectancy <dbl>, freedom <dbl>
```

Using ifelse when creating a variable

We can also create a dummy variable with the `ifelse()` function. The way we use this function is as: `ifelse(test, yes, no)`. We can also use another function called `case_when()`.

```
whr15 %>%
  mutate(
    latin_america_car = ifelse(region == "Latin America and Caribbean", 1, 0)
  ) %>%
  arrange(-latin_america_car) %>%
  head(5)
```

```
## # A tibble: 5 × 9
##   country    region   year happiness_rank happiness_score economy_gdp_per_capita
##   <chr>      <chr>   <dbl>          <dbl>           <dbl>                  <dbl>
## 1 Costa Rica Latin …  2015             12            7.23                  0.956
## 2 Mexico     Latin …  2015             14            7.19                  1.02
## 3 Brazil     Latin …  2015             16            6.98                  0.981
## 4 Venezuela  Latin …  2015             23            6.81                  1.04
## 5 Panama     Latin …  2015             25            6.79                  1.06
## # ℹ 3 more variables: health_life_expectancy <dbl>, freedom <dbl>,
## #   latin_america_car <dbl>
```

Using case_when() to update a variable

Recall the problem we have with regions in the `whr17` data. We can fix it as follows:

```
whr17 <- whr17 %>%
  mutate(
    country = case_when(
      country == "Hong Kong S.A.R., China" ~ "Hong Kong",
      country == "Taiwan Province of China" ~ "Taiwan",
      TRUE ~ country
    )
  )

whr17 %>%
  left_join(regions, by = "country") %>%
  rename(region = region.y) %>%
  select(-region.x) %>%
  select(country, region, everything()) %>%
  filter(is.na(region))
```

Factor variables

Factor variables

- When we imported this data set, we told R explicitly to not read strings as factor.
- We did that because we knew that we'd have to fix the country names.
- The region variable, however, should be a factor.

```
str(whr_panel$region)
```

```
##  chr [1:470] "Western Europe" "Western Europe" "Western Europe" ...
```

Factor variables

To create a factor variable, we use the `factor()` function (or `as_factor()` from the `forcats` package).

- `factor(x, levels, labels)` : turns numeric or string vector `x` into a factor vector.
- `levels`: a vector containing the possible values of `x`.
- `labels`: a vector of strings containing the labels you want to apply to your factor variable
- `ordered`: logical flag to determine if the levels should be regarded as ordered (in the order given).

If your categorical variable does not need to be ordered, and your string variable already has the label you want, making the conversion is quite easy.

Factor variables

Extra exercise: Turn a string variable into a factor.

- Use the mutate function to create a variable called region_cat containing a categorical version of the region variable.
- TIP: to do this, you only need the first argument of the factor function.

Solution:

```
whr_panel <- mutate(whr_panel, region_cat = factor(region))
```

And now we can check the class of our variable.

```
class(whr_panel$region_cat)
```

```
## [1] "factor"
```

Reshaping a dataset

Reshaping a dataset

Finally, let's try to reshape our dataset using the tidyverse functions. No more `reshape` from Stata. We can use `pivot_wider` or `pivot_longer`. Let's assign our wide format panel to an object called whr_panel_wide.

Long to Wide | Wide to Long

```
whr_panel %>%
  select(country, region, year, happiness_score) %>%
  pivot_wider(
    names_from = year,
    values_from = happiness_score
  ) %>%
  head(3)
```

```
## # A tibble: 3 × 5
##   country     region         `2015` `2016` `2017`
##   <chr>       <chr>           <dbl>  <dbl>  <dbl>
## 1 Switzerland Western Europe   7.59   7.51   7.49
## 2 Iceland     Western Europe   7.56   7.50   7.50
## 3 Denmark     Western Europe   7.53   7.53   7.52
```

Useful links

- **R for Data Science** by Hadley Wickham and Garrett Grolemund
  - Comprehensive introduction to data wrangling and visualization.
  - Read it online for free.
- **Tidyverse Cookbook**
  - Practical solutions for common data wrangling tasks.
  - Tidyverse Cookbook GitHub.
- **Tidyverse Cheat Sheets**
  - Official cheat sheets for `dplyr`, `tidyr`, and other Tidyverse packages.
  - Tidyverse Resources.

Thanks! // ¡Gracias!