

# Data preparation

An introduction to R, day 2

**Prof. Dr. Claudio Gräbner-Radkowitsch**  
**Europa-University Flensburg, Department of Pluralist Economics**

[www.claudius-graebner.com](http://www.claudius-graebner.com) | [@ClaudiusGraebner](https://twitter.com/ClaudiusGraebner) | [claudius@claudius-graebner.com](mailto:claudius@claudius-graebner.com)

# Learning goals

- I. Clarify questions on day 1
- II. Learn how to import csv files using the function `data.table::fread()`
- III. Clarify the process of data preparation and the concept of tidy data
- IV. Get an overview over the most common transformation routines and the corresponding functions from the `tidyverse` and `dplyr` packages

# Recap day 1

- R is a **high-level programming language** with dialects, R Studio an IDE
- R **packages** as a way to expand the capabilities of base R
- For your projects use R **projects**, a clear **folder structure** and the **here package**
- “Everything that exists is an object, everything that happens is a function call.”
- **Four basic object types**, advanced ones as modifications of the basic types

Questions?

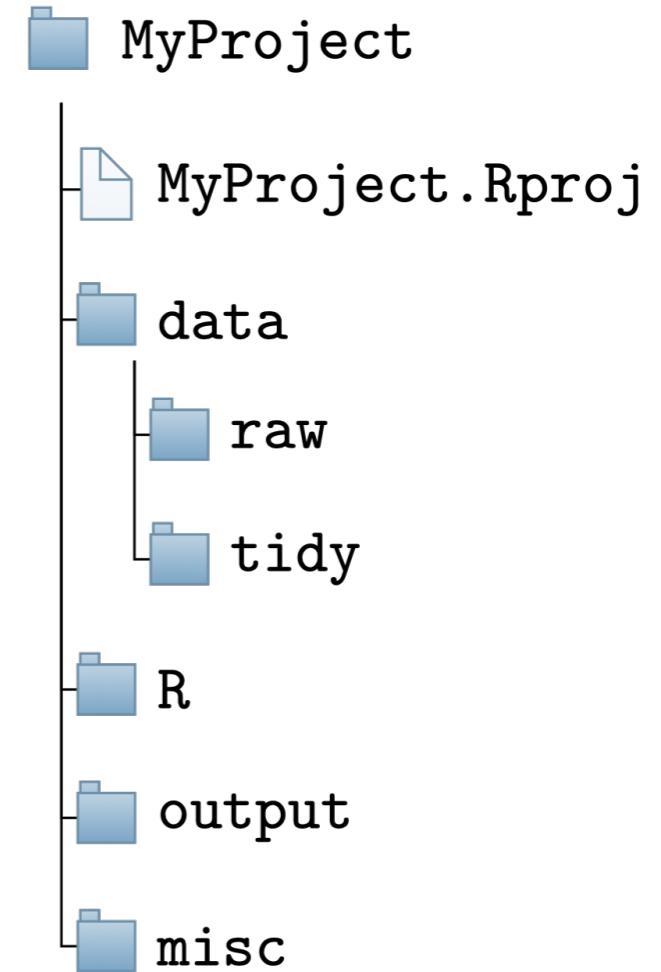
# Import data

# Preparation: set up your working environment

See separate session on project management

- Raw data should be saved in `data/raw`
- Tidied up data should be saved in `data/tidy`  
→ keep it separate

See upcoming topic on data preparation!



## Your turn:

Set up a working environment, download the data sets `fread_expls.zip` from the homepage and save them in the right folder.

# Import functions



- Once set up the project environment you can import data
  - Assume that your raw data is stored in the folder `data/raw`
- The right function to import a data set depends on the file type:



`csv/tsv files`

`.Rds/RData files`

`Excel files`

`Specific formats`

`data.table::fread()`

`readRDS()  
load()`

`readxl::read_xls()  
readxl::read_xlsx()`    `haven::read_dta()  
haven::read_sas()  
haven::read_spss()`

- Basic procedure the same in all cases → focus on reading `csv` files here

# How to import data

- Good practice: save path to file in a vector:

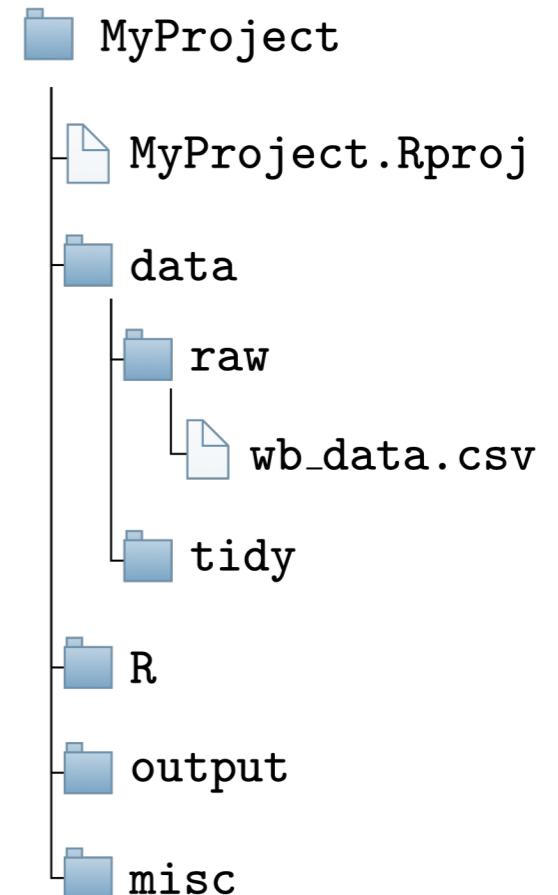
```
data_path <- here::here("data/raw/wb_data.csv")
```

- csv file → `data.table::fread()`:

```
data.table::fread(file = data_path)  
data.table::fread(  
  file = here::here("data/raw/wb_data.csv"))
```

- This uses default options to import the file

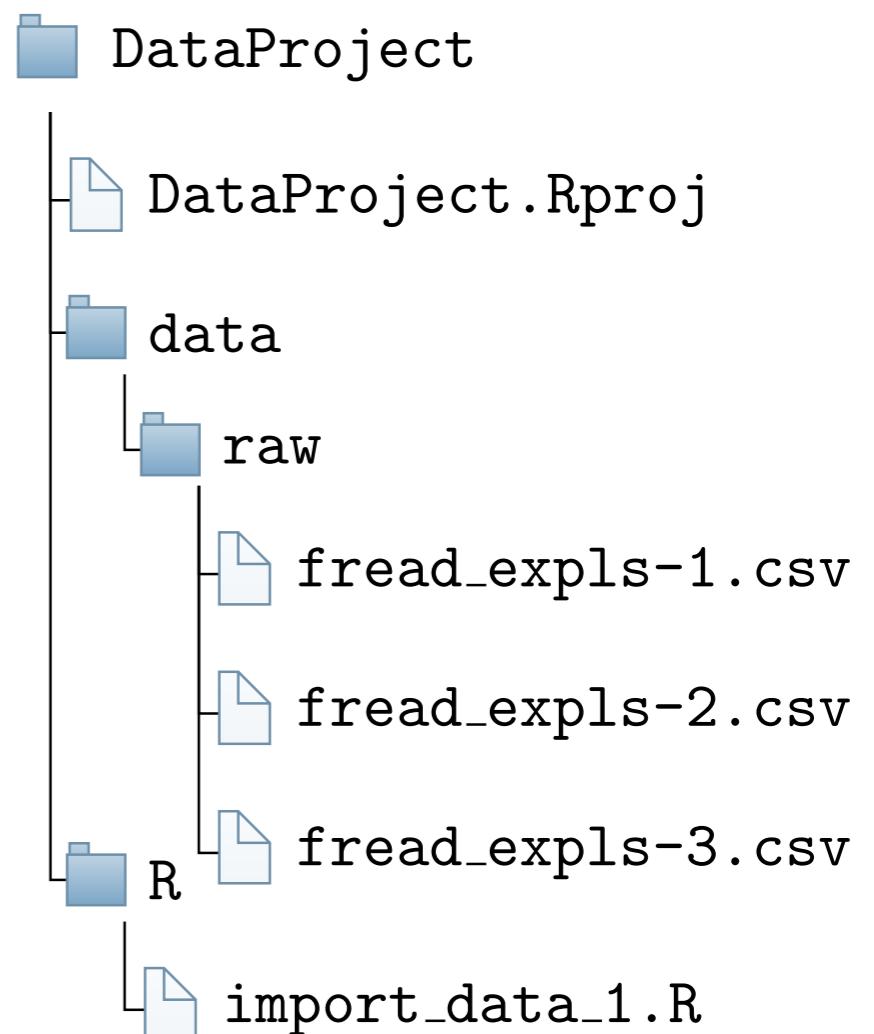
- Works often for clean data files
  - For not so clean data files we must specify several optional arguments



**Note:** there are many alternatives to `data.table::fread()`, including from base R or the tidyverse. But given its speed and flexibility I recommend to always use `fread()` and then transform the resulting `data.table` into a `tibble` or `data.frame`.

# Data import: exercise 1

- Download the zip file **fread\_expls.zip** from the course homepage
- Extract the zip file within the folder **data/raw/** in your R project
- Write a script that imports the data set saved in the file **fread\_expls-1.csv** into your session



# How to use `data.table::fread()`

- See also the tutorial on *data import*
- The following arguments of `data.table::fread()` are often helpful:
  - `file`: the relative path to the csv file you want to read → use `here::here()`
  - `sep`: symbol that separates columns
  - `dec`: symbol used as decimal sign
  - `colClasses`: what object type should be used for the columns?
- For other widely used commands check the tutorial and do the exercises
  - But note that there are even more specification options → `help(fread)`

# How to use `data.table::fread()`

## Specify column separator

- See also the tutorial on *data import*
- The following arguments of `data.table::fread()` are often helpful:
  - `file`: the relative path to the csv file you want to read → use `here::here()`
  - `sep`: **symbol that separates columns**
  - `dec`: symbol used as decimal sign
  - `colClasses`: what object type should be used for the columns?
- For other widely used commands check the tutorial and do the exercises
  - But note that there are even more specification options → `help(fread)`

# How to use `data.table::fread()`

## Specify column separator

```
c_code; year; exports; unemployment  
AT; 2013; 53.44; 5.34  
AT; 2014; 53.39; 5.62  
DE; 2013; 45.4; 5.23  
DE; 2014; 45.64; 4.98
```

- Especially in Germany, columns are often separated via ; instead of ,
- We can pass a string to `sep` indicating how the columns are separated
  - In the above case: `sep = ";"`

# How to use `data.table::fread()`

## Specify column separator

- See also the tutorial on *data import*
- The following arguments of `data.table::fread()` are often helpful:
  - `file`: the relative path to the csv file you want to read → use `here::here()`
  - `sep`: symbol that separates columns
  - `dec`: **symbol used as decimal sign**
  - `colClasses`: what object type should be used for the columns?
- For other widely used commands check the tutorial and do the exercises
  - But note that there are even more specification options → `help(fread)`

# How to use `data.table::fread()`

## Specify decimal separator

```
c_code; year; exports; unemployment  
AT; 2013; 53,44; 5,34  
AT; 2014; 53,39; 5,62  
DE; 2013; 45,4; 5,23  
DE; 2014; 45,64; 4,98
```

- Again in Germany, decimal places are often separated via , instead of .
- We can pass a string to `dec` indicating how the columns are separated
  - In the above case: `dec = ", "`

## Exercise 2

- Write a script that imports the data set `fread_expls-2.csv` into your session such that the following `tibble` results:

```
# A tibble: 4 × 4
  c_code  year exports unemployment
  <chr>   <int>    <dbl>        <dbl>
1 AT      2013     53.4       5.34
2 AT      2014     53.4       5.62
3 DE      2013     45.4       5.23
4 DE      2014     45.6       4.98
```

# How to use `data.table::fread()`

## Specifying column types using `colClasses`

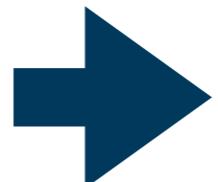
- See also the tutorial on *data import*
- The following arguments of `data.table::fread()` are often helpful:
  - `file`: the relative path to the csv file you want to read → use `here::here()`
  - `sep`: symbol that separates columns
  - `dec`: symbol used as decimal sign
  - **colClasses: what object type should be used for the columns?**
- For other widely used commands check the tutorial and do the exercises
  - But note that there are even more specification options → `help(fread)`

# How to use `data.table::fread()`

## Specifying column types using `colClasses`

- Whenever numbers should be saved as character, the guessing algorithm of `data.table::fread()` often fails:

```
c_code,year,exports, PROD_CODE  
AT, 2013, 53.44, 0011  
AT, 2014, 53.39, 0011  
DE, 2013, 45.4, 0011  
DE, 2014, 45.64, 0011
```

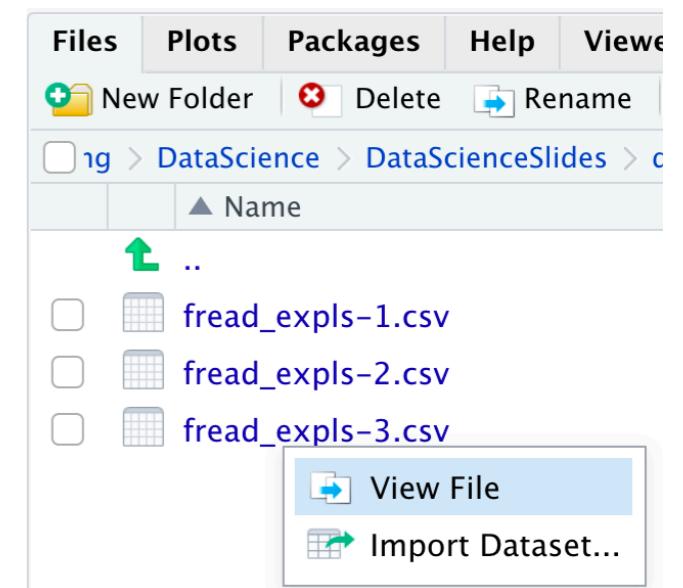


	c_code	year	exports	PROD_CODE
	<chr>	<int>	<dbl>	<int>
1	AT	2013	53.4	11
2	AT	2014	53.4	11
3	DE	2013	45.4	11
4	DE	2014	45.6	11

- We can specify the column types explicitly by passing a vector to `colClasses`:
  - `colClasses = c("character", rep("double", 2), "character")`
  - Usually, this is often a good idea to make your code more transparent
  - You can also combine it with `select` and only read selected columns (see tutorial)

# Exercise 3

- Now read in the file `fread_expls-3.csv` and use all the arguments you consider to be necessary
- Hint:
  - To get an idea about the raw data, click on the file and select “View File” to see it in its raw form → helps you to choose the right arguments:
  - Infeasible for very large files → use `nrows` and `select` to read a representative subset (see tutorial)



# And what about saving data?

- Saving data is much easier than reading data
- The only relevant question is about the format
  - If there are no good arguments for using a different format, go for csv
- This can be achieved by `data.table::fwrite()` with the main arguments:
  - `x`: the name of the object to be saved
  - `file`: the file name under which the object should be saved
- Example: save object `exp_tab` to file `data/exp_tab.csv`:

```
data.table::fwrite(  
  x = exp_tab,  
  file = here::here("data/exp_tab.csv")  
)
```

# Data import - the general idea

**Make yourself comfortable before reading in data -  
expect frustration!**

- General idea: you import the data and bind it to an R object - usually a `data.frame` or whatever aligns with your preferred dialect
- Then you proceed with transforming this `data.frame` until it satisfies the demands for tidy data
- Then you save the data under a new name, save the script, and celebrate yourself 🎉🍾🥂



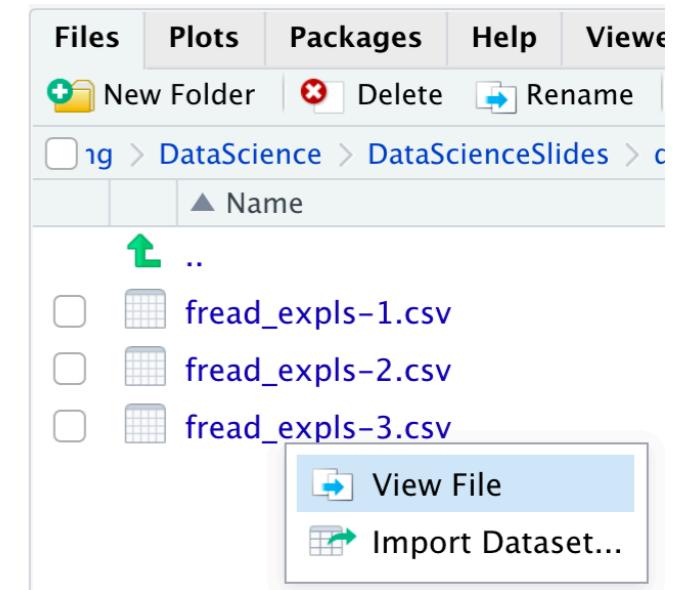
# Summary and conclusion

- You learned how to import data into R
- Main focus: importing csv data files using `data.table::fread()`
  - Other **functions** for csv provided, e.g., via the `tidyverse` packages
  - Other **formats**: specialised functions available, esp. in the `haven` package
- Importing standard data often works well with **default options**
- In other cases, optional arguments must be used → check **function documentation**
- If **speed or memory restrictions** are an issue, comparing import functions is advisable

# **Appendix: Typical challenges and their solution**

# Some general hints and suggestions I

- Always check whether the imported data set looks as expected
  - If in doubt, always look at the plain file using a plain text editor and compare!
  - Try to avoid csv data in Excel, Numbers or any other “advanced” program



```
> data_ex3 <- fread(file = data_path)
> head(data_ex3)
  cgroup commoditycode product_complexity exp_share
  <char>      <int>           <num>      <num>
1: Core countries        101     0.06424262 0.01312370
2: Periphery countries   101     0.06424262 0.04639794
3: Core countries        102    -0.49254290 0.05162508
4: Periphery countries   102    -0.49254290 0.03700469
5: Core countries        103     0.51082386 0.05324995
6: Periphery countries   103     0.51082386 0.04082251
```

```
1  cgroup,commoditycode,product_complexity,exp_share
2  Core countries,0101,0.0642426217529412,0.0131236961169874
3  Periphery countries,0101,0.0642426217529412,0.0463979373363288
4  Core countries,0102,-0.492542902941177,0.0516250772043892
5  Periphery countries,0102,-0.492542902941177,0.0370046875606996
6  Core countries,0103,0.510823855941177,0.0532499520404157
```

# Some general hints and suggestions II

- **Problem:** your data set is very large and every import takes very long
- **Solution:** use `nrows=n` imports only the first `n` rows

```
data_raw <- fread(  
  file = data_path, nrows = 10  
)
```

- **Problem:** you are irritated by the standard `data.table` format of the data sets imported by `data.table::fread()`
- **Solution:** use `data.table=FALSE` to get a `data.frame` or use `tibble::as_tibble()` immediately

```
data_raw <- fread(  
  file = data_path, data.table = FALSE  
)
```

```
data_raw <- as_tibble(fread(  
  file = data_path))  
)
```

# Some general hints and suggestions III

- **Problem:** the column names of the imported data set are V1, V2, ...
  - Note: this happens when the column names are numbers (e.g., years)
- **Solution:** set header=TRUE

```
country,Growth,year
Germany,-4.5696,2020
Germany,1.0555,2019
Germany,1.086,2018
Germany,2.6802,2017
Germany,2.23,2016
Germany,1.4919,2015
```

```
> data_raw <- as_tibble(fread(file = data_path))
> data_raw
# A tibble: 2 × 7
      V1        V2        V3        V4        V5        V6        V7
      <chr>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
1 country 2020    2019    2018    2017    2016    2015
2 Germany -4.57    1.06    1.09    2.68    2.23    1.49

> data_raw <- as_tibble(fread(file = data_path, header = TRUE))
> data_raw
# A tibble: 1 × 7
  country `2020` `2019` `2018` `2017` `2016` `2015`
  <chr>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
1 Germany -4.57    1.06    1.09    2.68    2.23    1.49
```

# Some general hints and suggestions IV

- **Problem:** there are one or two additional columns named V\*

  - Note: this happens when the underlying csv file is corrupted and ends with a ,

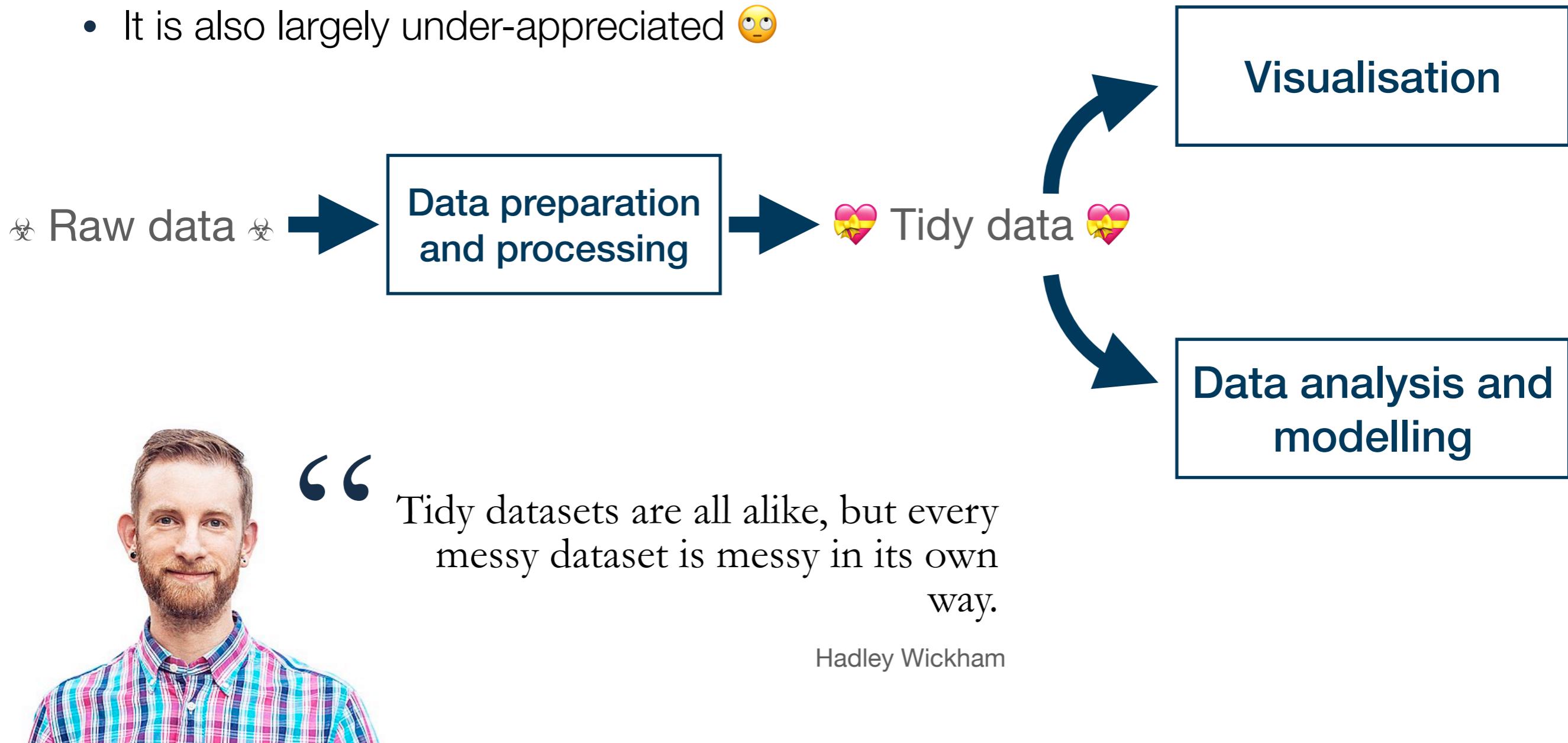
- **Solution:** check for the column names with `names()` and remove the corrupted columns

```
> data_raw <- as_tibble(fread(file = data_path, header = TRUE))
> data_raw
# A tibble: 1 × 5
country `2020` `2019` `2018` V5
<chr>    <dbl>   <dbl>   <dbl> <lgl>
1 Germany -4.57    1.06    1.09 NA
> names(data_raw)
[1] "country" "2020"     "2019"     "2018"     "V5"
> data_raw <- select(data_raw, -all_of(c("V5")))
> data_raw
# A tibble: 1 × 4
country `2020` `2019` `2018` 
<chr>    <dbl>   <dbl>   <dbl>
1 Germany -4.57    1.06    1.09
```

# Data preparation

# The role of data preparation

- Importing and preparing is the most fundamental task in data science
  - It is also largely under-appreciated 😳



# The goal: tidy data

Every **column** corresponds to one and only one **variable**

Every **row** corresponds to one and only one **observation**

	c_code	year	exports	unemployment
1	AT	2013	53.4	5.34
2	AT	2014	53.4	5.62
3	DE	2013	45.4	5.23
4	DE	2014	45.6	4.98

Every **cell** corresponds to one and only one **value**

- Every data set that satisfies these three demands is called tidy
- Excellent start for basically every further task – but maybe not the best way to represent data to humans

# The goal: tidy data

Every **row** corresponds to one and only one **observation**



Every **column** corresponds to one and only one **variable**



Every **cell** corresponds to one and only one **value**



```
# A tibble: 2 × 4
c_code variable `2013` `2014`
<chr> <chr>   <dbl>   <dbl>
1 AT   unemployment 5.34    5.62
2 DE   unemployment 5.23    4.98
```

```
# A tibble: 4 × 4
c_code year exports variable
<chr> <int>   <dbl> <chr>
1 AT     2013    53.4 exports
2 AT     2014    53.4 exports
3 DE     2013    45.4 exports
4 DE     2014    45.6 exports
```

```
# A tibble: 2 × 2
country important_industries
<chr> <chr>
1 DE   Cars (25%) and meat (10%)
2 AT   Wine (5%) and milk (2%)
```

- Goal of data preparation: turn such untidy data into tidy data
  - Includes data manipulation (?) and data wrangling

# Short recap

- Look at the following data sets: how would a tidy version look like?

```
# A tibble: 6 × 4
  continent year variable     value
  <fct>    <int> <chr>      <dbl>
1 Africa     2002 lifeExp      53.3
2 Africa     2002 pop        16033152.
3 Africa     2002 gdpPercap   2599.
4 Africa     2007 lifeExp      54.8
5 Africa     2007 pop        17875763.
6 Africa     2007 gdpPercap   3089.
```

```
# A tibble: 1 × 7
  country `2020` `2019` `2018` `2017` `2016` `2015`
  <chr>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Germany -4.57   1.06   1.09   2.68   2.23   1.49
```

```
# A tibble: 10 × 5
  continent year lifeExp       pop gdpPercap
  <fct>    <int> <dbl>      <dbl>      <dbl>
1 Africa     2002  53.3  16033152.  2599.
2 Africa     2007  54.8  17875763.  3089.
3 Americas   2002  72.4  33990910.  9288.
4 Americas   2007  73.6  35954847. 11003.
5 Asia       2002  69.2  109145521. 10174.
```

```
# A tibble: 6 × 3
  country Growth year
  <chr>    <dbl> <int>
1 Germany -4.57  2020
2 Germany  1.06  2019
3 Germany  1.09  2018
4 Germany  2.68  2017
5 Germany  2.23  2016
6 Germany  1.49  2015
```

# Further recap questions

- What are the three demands a data set needs to fulfil to count as ‘tidy’?
- Why do we care about tidy data at all?
- Are there plausible reasons for transforming a tidy data set into a non-tidy data set?

# The way to tidy data

“ Tidy datasets are all alike, but every messy dataset is messy in its own way.

Hadley Wickham

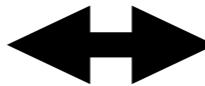


- The starting point to tidy data is always different
  - The goal is always the same → so are the steps: **six main routines**
  - Often useful: make a physical note of how the tidy data set should look like
- Two main packages are relevant:
  - `tidyverse` provides functions for **reshaping data** into tidy format ('wrangling')
  - `dplyr` provides functions for **manipulating data** to extract desired information

# The six routines of data preparation

**Reshaping** data from long to wide format (and vice versa)

```
# A tibble: 4 × 4
  c_code year exports unemployment
  <chr>  <int>   <dbl>      <dbl>
1 AT     2013    53.4       5.34
2 AT     2014    53.4       5.62
3 DE     2013    45.4       5.23
4 DE     2014    45.6       4.98
```

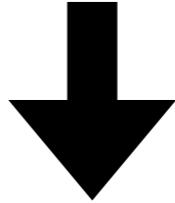


```
# A tibble: 8 × 4
  c_code  year variable   value
  <chr>  <int> <chr>     <dbl>
1 AT     2013  exports  53.4
2 AT     2013  unemployment 5.34
3 AT     2014  exports  53.4
4 AT     2014  unemployment 5.62
5 DE     2013  exports  45.4
6 DE     2013  unemployment 5.23
7 DE     2014  exports  45.6
8 DE     2014  unemployment 4.98
```

# The six routines of data preparation

**Filter** rows according to conditions

```
# A tibble: 4 × 4
  c_code   year exports unemployment
  <chr>   <int>   <dbl>        <dbl>
1 AT       2013    53.4        5.34
2 AT       2014    53.4        5.62
3 DE       2013    45.4        5.23
4 DE       2014    45.6        4.98
```

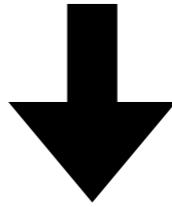


```
# A tibble: 2 × 4
  c_code   year exports unemployment
  <chr>   <int>   <dbl>        <dbl>
1 DE       2013    45.4        5.23
2 DE       2014    45.6        4.98
```

# The six routines of data preparation

Select columns/variables

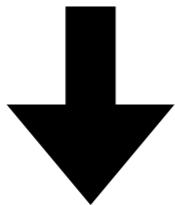
```
# A tibble: 4 × 4
  c_code   year exports unemployment
  <chr>   <int>   <dbl>        <dbl>
1 AT       2013    53.4       5.34
2 AT       2014    53.4       5.62
3 DE       2013    45.4       5.23
4 DE       2014    45.6       4.98
```



```
# A tibble: 4 × 3
  c_code   year exports
  <chr>   <int>   <dbl>
1 AT       2013    53.4
2 AT       2014    53.4
3 DE       2013    45.4
4 DE       2014    45.6
```

# The six routines of data preparation

```
# A tibble: 2 × 4
  c_code variable   `2013` `2014`
  <chr>   <chr>     <dbl>   <dbl>
1 AT      unemployment  5.34    5.62
2 DE      unemployment  5.23    4.98
```

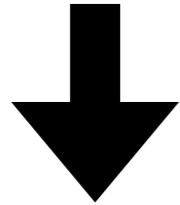


Mutate or create variables

```
# A tibble: 2 × 5
  c_code variable   `2013` `2014` change
  <chr>   <chr>     <dbl>   <dbl>   <dbl>
1 AT      unemployment  5.34    5.62    0.285
2 DE      unemployment  5.23    4.98   -0.25
```

# The six routines of data preparation

```
# A tibble: 4 × 4
  c_code  year exports unemployment
  <chr> <int>   <dbl>        <dbl>
1 AT      2013     53.4       5.34
2 AT      2014     53.4       5.62
3 DE      2013     45.4       5.23
4 DE      2014     45.6       4.98
```



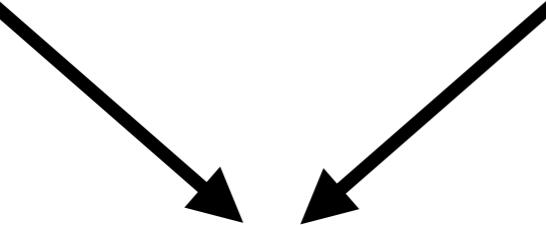
**Group and summarise data**

```
# A tibble: 2 × 3
  c_code exports_avg unemployment_avg
  <chr>      <dbl>           <dbl>
1 AT          53.4           5.48
2 DE          45.5           5.11
```

# The six routines of data preparation

```
# A tibble: 4 × 3
  c_code  year exports
  <chr> <int>   <dbl>
1 AT     2013    53.4
2 AT     2014    53.4
3 DE     2013    45.4
4 DE     2014    45.6
```

```
# A tibble: 4 × 3
  c_code  year unemployment
  <chr> <int>      <dbl>
1 AT     2013      5.34
2 AT     2014      5.62
3 DE     2013      5.23
4 DE     2014      4.98
```



```
# A tibble: 4 × 4
  c_code  year exports unemployment
  <chr> <int>   <dbl>      <dbl>
1 AT     2013    53.4      5.34
2 AT     2014    53.4      5.62
3 DE     2013    45.4      5.23
4 DE     2014    45.6      4.98
```

Merge several data sets

# The six routines of data preparation

- After having imported your data into R, you can usually make it tidy using a sequential combination of the following routines:

**Reshaping** data from long to wide format (and vice versa)

**Filter** rows according to conditions

**Select** columns/variables

**Mutate** or create variables

**Group** and **summarise** data

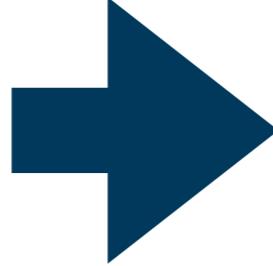
**Merge** several data sets

- With these six routines, you can prepare almost any messy data set
- This way you produce the inputs we used for visualisation...
  - ...and the inputs we will use for modelling

# Exercises on reshaping

- Take the data set `data_raw_long.csv` and transform it as follows:

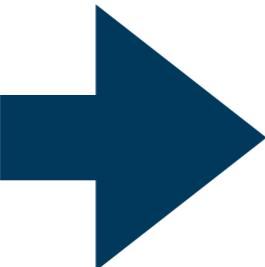
```
> data_raw_long
# A tibble: 8 × 4
  country year variable   value
  <chr>   <dbl> <chr>     <dbl>
1 Germany  2017  unemp      3.75
2 Germany  2017  gdp       53071.46
3 Germany  2018  unemp      3.38
4 Germany  2018  gdp       53431.39
5 Greece   2017  unemp     21.49
6 Greece   2017  gdp       28604.86
7 Greece   2018  unemp     19.29
8 Greece   2018  gdp       29141.17
```



```
# A tibble: 4 × 4
  year variable Germany Greece
  <dbl> <chr>     <dbl>   <dbl>
1 2017  unemp      3.75    21.5
2 2017  gdp       53071.  28605.
3 2018  unemp      3.38    19.3
4 2018  gdp       53431.  29141.
```

- Take the data set `data_raw_wide.csv` and transform it as follows:

```
> data_raw_wide
# A tibble: 2 × 3
  country `2017` `2018`
  <chr>    <dbl>   <dbl>
1 Germany   29.4    29.6
2 Greece    32.2    31.7
```



```
# A tibble: 4 × 3
  country year   gini
  <chr>   <dbl> <dbl>
1 Germany  2017  29.4
2 Germany  2018  29.6
3 Greece   2017  32.2
4 Greece   2018  31.7
```

# Exercises on manipulation basics

- Consider the data set `wine2dine` from the package `DataScienceExercises`
1. Filter the data set such that it only contains white wines
  2. Then remove the column ‘kind’
  3. Change the type of the column ‘quality’ into double
  4. Divide the values in the columns ‘alcohol’ and ‘residual sugar’ by 100
  5. Filter the data such that you only keep the wines with the highest quality score



# Exercises on summarising and grouping

- What is the difference between `dplyr::mutate()` and `dplyr::summarize()`?
  - Consider again the data set `wine2dine` from the package `DataScienceExercises`
1. Summarise the data by computing the mean alcohol, mean sugar, and mean quality of white and red wines
  2. Compute a variable indicating how the quality of each wine deviates from the average quality of all wines.



# Exercises on joining data sets

- Consider the data sets `join_x.csv` and `join_y.csv` and join them on the columns `time` and `id` using the functions `left_join()`, `right_join()`, and `full_join()`!
- Try for yourself what the function `inner_join()` does. How does it differ from `left_join()`, `right_join()`, and `full_join()`?
- Consider the data sets `join_x.csv` and `join_y.csv` and the function `dplyr::full_join()`. What is the difference of joining on columns `time` and `id` vs joining only on column `id`?



# Helpful tools I: Pipes



# Using pipes

- While not strictly necessary, you can improve the usability and readability of your code using so called pipes: `%>%`
- Pipes take the result from their left and ‘throw’ them on the right
  - The thrown result can be referred to via `.`
  - Usually they are used at the end of a line and ‘throw’ the result of one line into the next one

```
data_sub <- dplyr::select(  
  .data = data_raw,  
  country, year, unemp, gdp)
```



```
data_sub <- data_raw %>%  
  dplyr::select(  
    .data = .,  
    country, year, unemp, gdp)
```

# Using pipes

- While not strictly necessary, you can improve the usability and readability of your code using so called pipes: `%>%`
- Pipes take the result of one line and ‘throw’ them into the next line
  - The thrown result can be referred to via `.`
  - By default, the thrown result is used as the first argument of the function in the next line

```
data_sub <- dplyr::select(  
  .data = data_raw,  
  country, year, unemp, gdp)
```



```
data_sub <- data_raw %>%  
  dplyr::select(  
    .data = .,  
    country, year, unemp, gdp)
```



```
data_sub <- data_raw %>%  
  dplyr::select(  
    country, year, unemp, gdp)
```

# Using pipes

- A more practical example:

```
chain_1 <- tidyr::pivot_longer(  
  data = data_raw_wide,  
  cols = c("gdp", "gini", "unemp"),  
  names_to = "indicator",  
  values_to = "val")
```

```
chain_2 <- tidyr::pivot_wider(  
  data = chain_1,  
  names_from = "year",  
  values_from = "val")
```

```
chain_complete <- pipe_data_raw %>%  
  tidyr::pivot_longer(  
    data = .,  
    cols = c("gdp", "gini", "unemp"),  
    names_to = "indicator",  
    values_to = "val") %>%  
  tidyr::pivot_wider(  
    data = .,  
    names_from = "year",  
    values_from = "val")
```

- Pipes make code almost always easier to read → desired stage at the end
- But it is usually easier to make intermediate steps explicit during code development

# Short recap on piping

- Explain what the pipe `%>%` does.
- When can the pipe be useful?
- Should you develop code with pipes right from the start? Why? Why not?



- Rewrite the following code 🤝 using pipes (data available via course page)

```
pipedata_v1 <- data.table::fread(here("data/recap2.csv"))

pipedata_v2 <- tidyverse::pivot_longer(
  data = pipedata_v1,
  cols = c("lifeExp", "gdpPercap"),
  names_to = "Indicator",
  values_to = "Value")

pipedata_v3 <- tidyverse::pivot_wider(
  data = pipedata_v2,
  names_from = "year",
  values_from = "Value")
```

- Look at the introduction to the R package `magrittr`, which defines even more pipes: <https://magrittr.tidyverse.org/articles/magrittr.html>

# **Helpful tools II: Selection helpers**

# Digression: tidy selection helpers

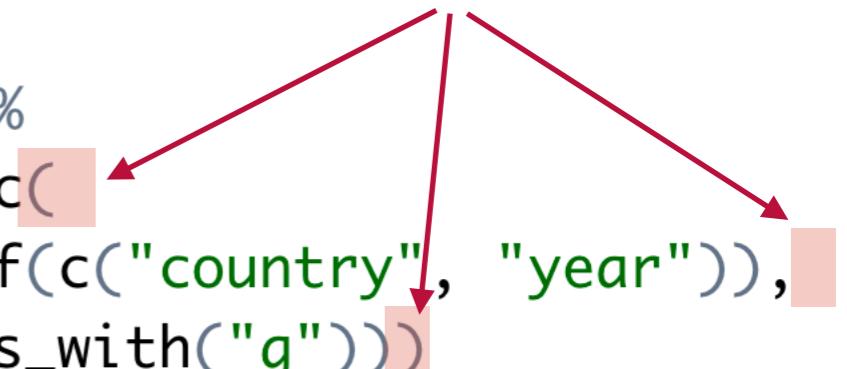
- It can become tedious to select many columns using explicit reference to their names
- The tidy selection helpers are a useful tool to select columns based on common criteria:

```
data_raw_wide %>%  
  dplyr::select(gdp, gini)
```

Combine multiple selectors

```
data_raw_wide %>%  
  dplyr::select(  
    tidyr::starts_with("g"))
```

```
data_raw_wide %>%  
  dplyr::select(c(  
    tidyr::all_of(c("country", "year")),  
    tidyr::starts_with("g"))))
```



- For a complete list of helpers see, e.g., [the official reference](#)

# Final exercises

# Final exercise 1: filtering and reshaping

- Use the data set `exercise_1.csv` contained in `wrangling_exercises_data.zip`

- Import the data and ...

- ...only consider data on Greece and Germany between 1995 and 2015

- ...make it wider (and tidy) 

- ...save it in the subfolder `data/tidy/`

# A tibble: 42 × 4				
	country	year	gdp	co2
	<chr>	<int>	<dbl>	<dbl>
1	Germany	1995	39366.	10.7
2	Germany	1996	39569.	11.0
3	Germany	1997	40219.	10.6
4	Germany	1998	41023.	10.5
5	Germany	1999	41770.	10.2
6	Germany	2000	42928.	10.1
7	Germany	2001	43577.	10.3
8	Germany	2002	43417.	10.1
9	Germany	2003	43089.	10.1
10	Germany	2004	43605.	9.95
# ... with 32 more rows				

# Final exercise 2: mutating, selecting & summarising

- Use the data set `exercise_2.csv` contained in `wrangling_exercises_data.zip`
- Import the data
  - Only keep the variables `gdp`, `share_indus`, and `co2`
  - Divide the industry share in GDP with 100
  - Only keep data between 2010 and 2018
  - Compute the averages over time for all countries

# A tibble: 12 × 3			
	country	indicator	time_avg
	<chr>	<chr>	<dbl>
1	Austria	co2	7.60
2	Austria	gdp	53322.
3	Austria	share_indus	0.254
4	Germany	co2	9.17
5	Germany	gdp	50781.
6	Germany	share_indus	0.272
7	Greece	co2	6.72
8	Greece	gdp	29169.
9	Greece	share_indus	0.144
10	Italy	co2	5.87
11	Italy	gdp	41326.
12	Italy	share_indus	0.213

# Summary & outlook

# Summary

- After importing raw data you usually must prepare them → make **tidy**
- Tidy data is the input to any visualisation/modelling task and defined as data where:
  - Every **column** corresponds to one and only one **variable**
  - Every **row** corresponds to one and only one **observation**
  - Every **cell** corresponds to one and only one **value**
- It is usually a good idea to write a script that imports raw, and saves tidy data
- Such script usually makes use of functions from the following packages:
  - **data.table**, **dplyr**, **tidyr**, and **here**

# Summary

- These packages provide functions that help you to address some wrangling challenges that regularly await you:
  - **Reshaping** data: `tidyr::pivot_longer()` and `tidyr::pivot_wider()`
  - **Filtering** rows: `dplyr::filter()`
  - **Selecting** columns: `dplyr::select()` and the select helpers
  - **Mutating** or **creating** variables: `dplyr::mutate()`
  - **Grouping** and **summarising**: `dplyr::group_by()` & `dplyr::summarise()`
  - **Merging** data sets: `dplyr::*_join()`

# General recap questions

- What are the three demands a data set needs to fulfil to count as ‘tidy’?
- Why do we care about tidy data at all?
- What is the relation between long and wide data sets?
- What are the six main routines of data preparation? What are they used for?
- What does ‘data wrangling’ mean?
- Which two packages are used most frequently in the context of data preparation? What are their respective areas of application?
- Explain what the pipe `%>%` does. When can the pipe be useful?

Data preparation is mainly about practice, so the practical exercises are particularly recommended

