Introduction to R

for Quantitative Social Science

Dr. Adrian Stanciu & Dr. Ranjit K. Singh

gesis
Leibniz-Institut
für Sozialwissenschaften

Day 1

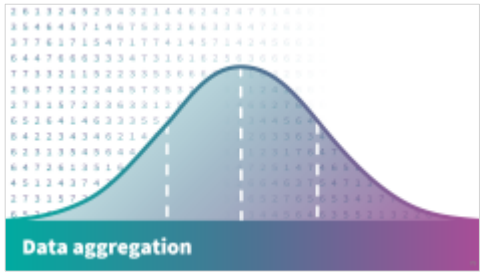
Diving into R

Data Structures & Import

Labelled Data

Data Wrangling

Day 2



Data aggregation

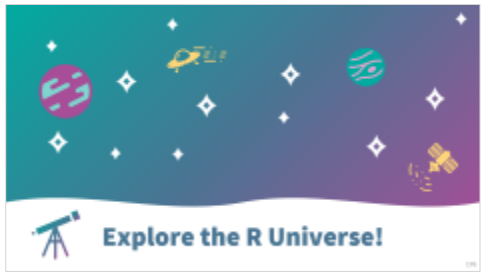
Data exploration

Data analysis

Day 3

Text and Tables

Plots



What's next?

click images to jump to each lesson!

Dr. Adrian Stanciu

mail adrian.stanciu@gesis.org

twitter @adrianvstanciu

web https://adrianstanciu.eu

orcid 0000-0001-8149-7829

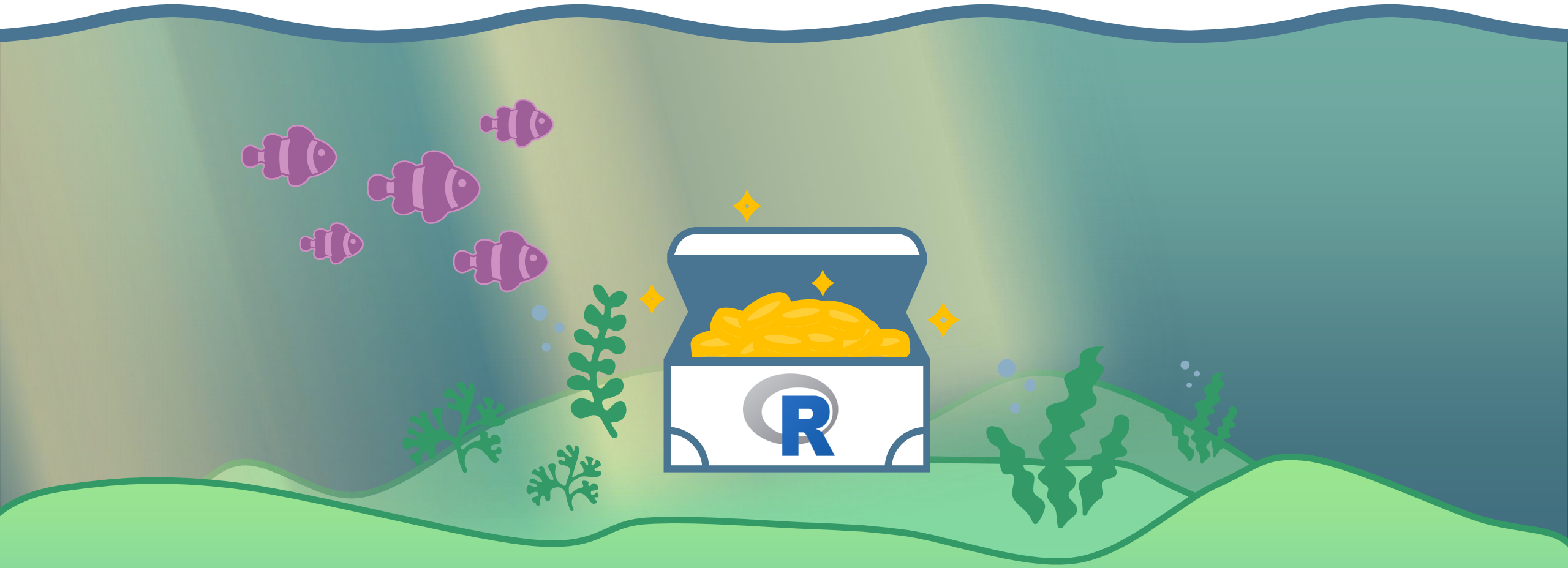
Dr. Ranjit K. Singh

mail ranjit.singh@gesis.org

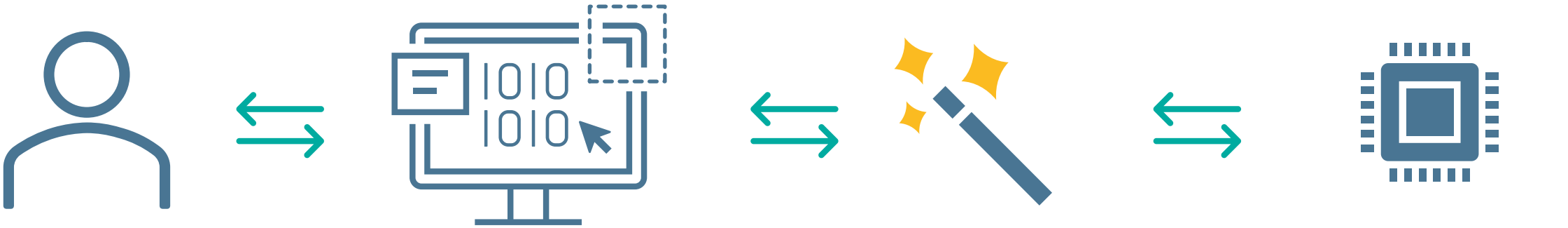
twitter @_R_K_Singh

web GESIS Profile

Diving into R!



R, **RStudio**, and the **tidyverse**



User **Interface** **Magic** **Processing**

RStudio®

tidyverse

rmarkdown

R, **RStudio**, and the **tidyverse**

- **R** is a statistical programming language
  - It is more programming language than SPSS and STATA
  - It is more statistical program than classical programming languages
- **RStudio** is an Integrated Development Environment (IDE) for R
  - It provides a graphical user interface
  - Many convenience features and keyboard shortcuts
  - Built in version control (e.g., via Git)
- **Tidyverse** is a collection of packages with a shared design philosophy
  - The tidyverse fundamentally changes how R syntax looks
  - It is (imho) more human readable, consistent, and convenient

Functions and **Packages**

- **Functions** tell R to „do something“.
  - e.g., `mean(df$age)` computes the average age of people in the dataset df

- **Packages** are collections of functions (and sometimes also data)
  - They extend the functionality of R enormously.
  - e.g., lavaan provides R with CFA capabilities simiar to MPlus
  - Packages have to be installed once:

    `install.packages("tidyverse")` installs the tidyverse on a new computer

    but loaded in every script that uses the package:

    `library(tidyverse)` tells R to prepare the script

    Alternatively, call functions directly with: `some_package::some_function()`

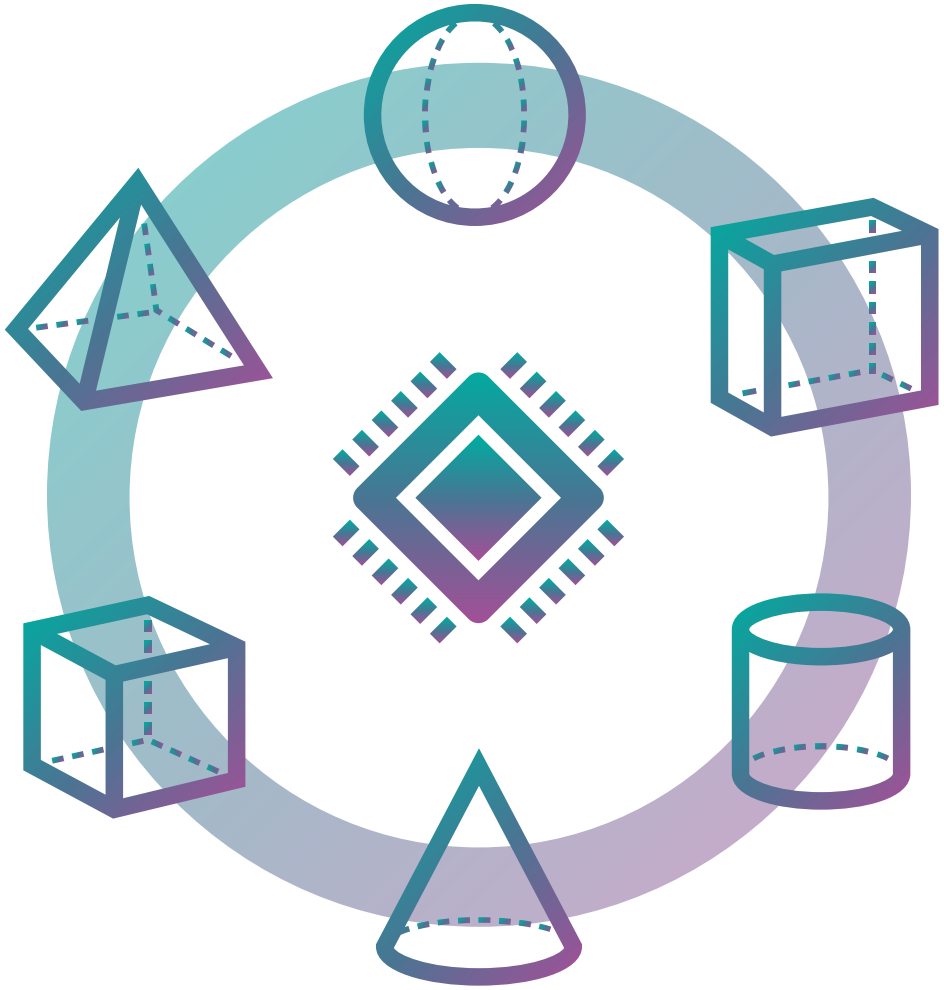
R **Console**, **Scripts**, and **Markdown**

R code can be entered in R in different ways.

- The **R console** allows you to give R short commands directly

- An **R Script** (file.R) contains pure R Syntax and can be run in parts or as a whole

- **R Markdown** (file.RMD) contains text document and R script parts.
  - Markdown allows us to prepare scripts that can be converted to html, word, or pdf documents
  - Think: Reports, papers, presentations, websites etc.

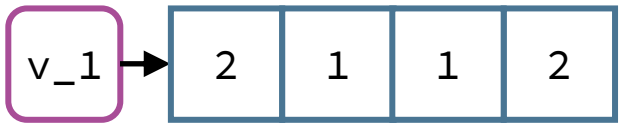


R Data Structures and Data Import

Data in R

Programming Variables vs. **Data** Variables

Variable:

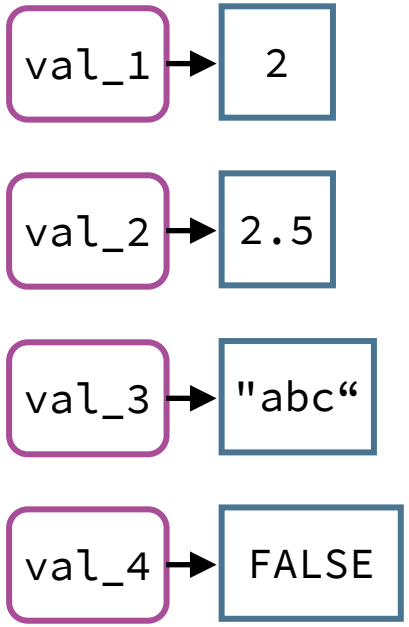


name object containing values

Types of data **structures**

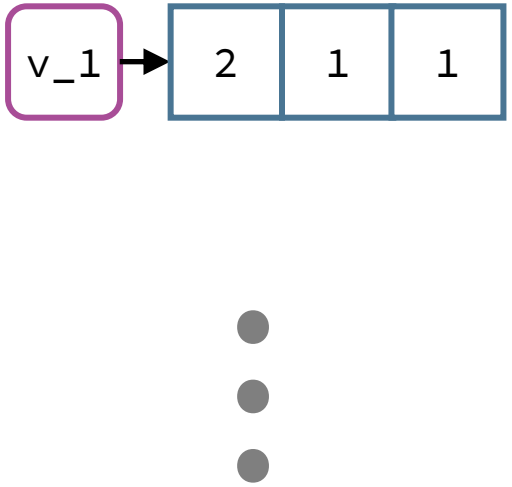
Values

Building blocks of data
(e.g., a response)



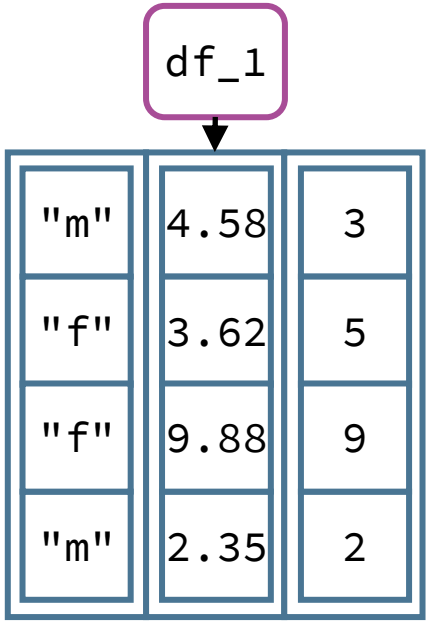
Vectors

Sequences of values
(e.g., a variable)



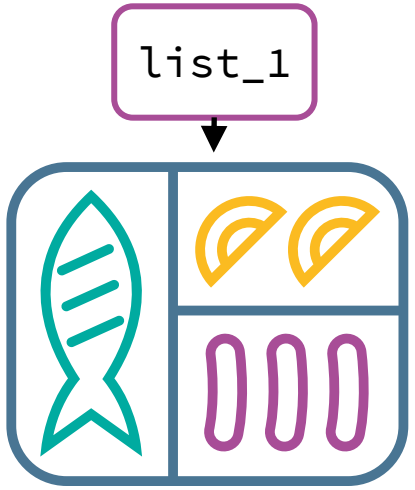
Data frames

A table structure
(e.g., a data set)



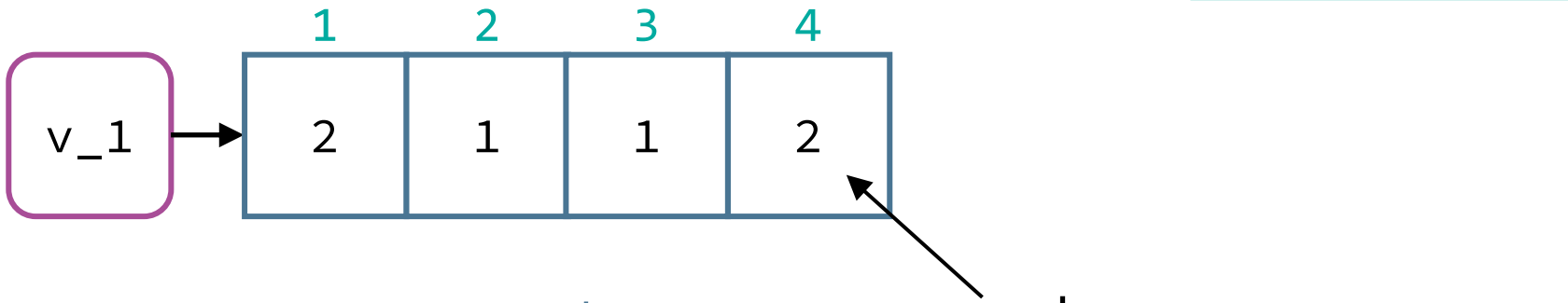
Lists

Complex data structures
which can contain
anything in any structure



Vector anatomy

(!)
Unlike many programming languages, R begins indices at 1, not 0.



Create this vector in R:

```
v_1 <- c(2, 1, 1, 2)
```

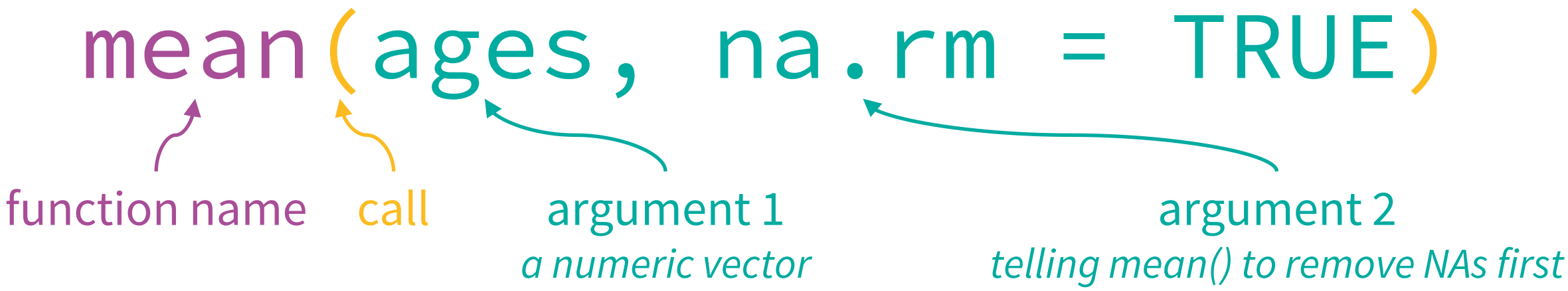
name `<-` `c()`

assign right side to left name

`combine values to vector`

Excursion: **Function** anatomy

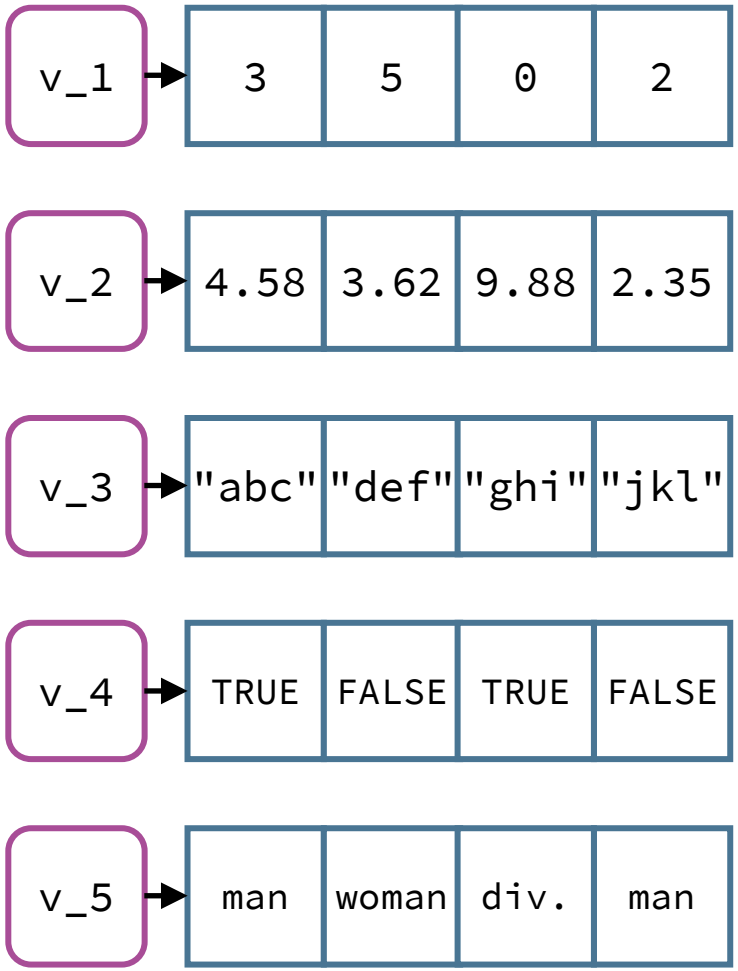
```
ages <- c(22, 34, 56, NA, 19, 89)
mean(ages, na.rm = TRUE)
# [1] 44
```



(!) Arguments can be provided by: position argument name

Tipp: Look up function documentation for mean with `?mean` or by pressing F1 in RStudio
Function definition: `mean(x, trim = 0, na.rm = FALSE, ...)`

Data types (~ Vector Types)



`Integer`

`Double`

`Numerical`

Metric values

`Character`

Arbitrary text content (e.g., open answers)

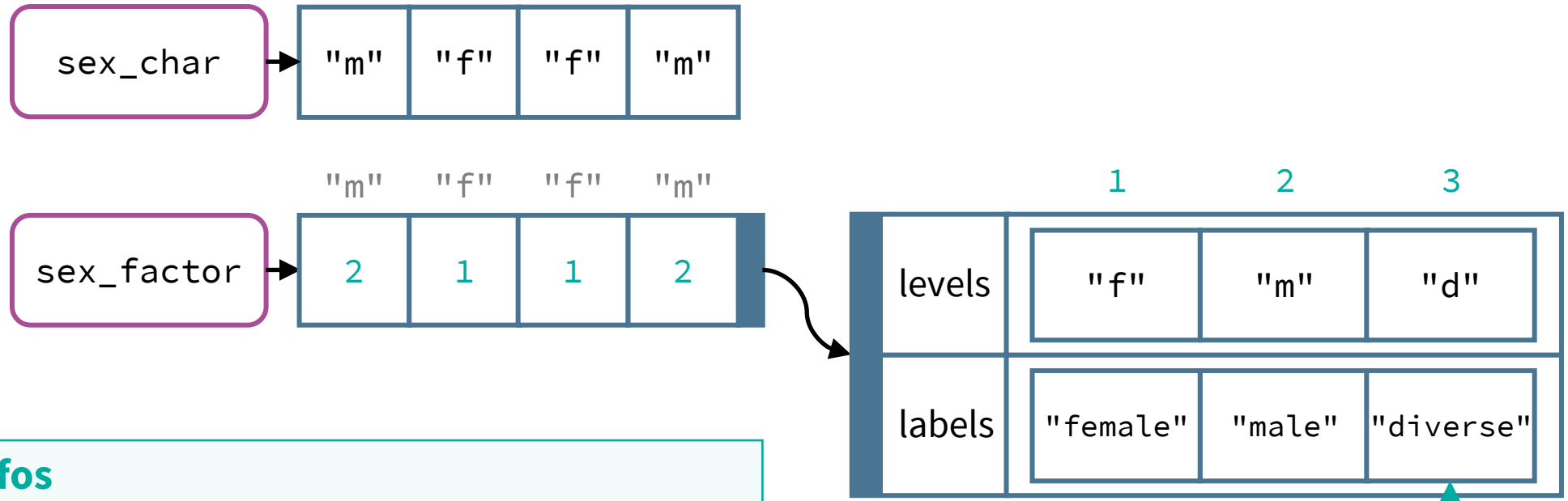
`Logical`

Dichotomous concepts

`Factor`

Concepts with finite and known possible values (e.g., ordinal or nominal variables)

Factor Vectors

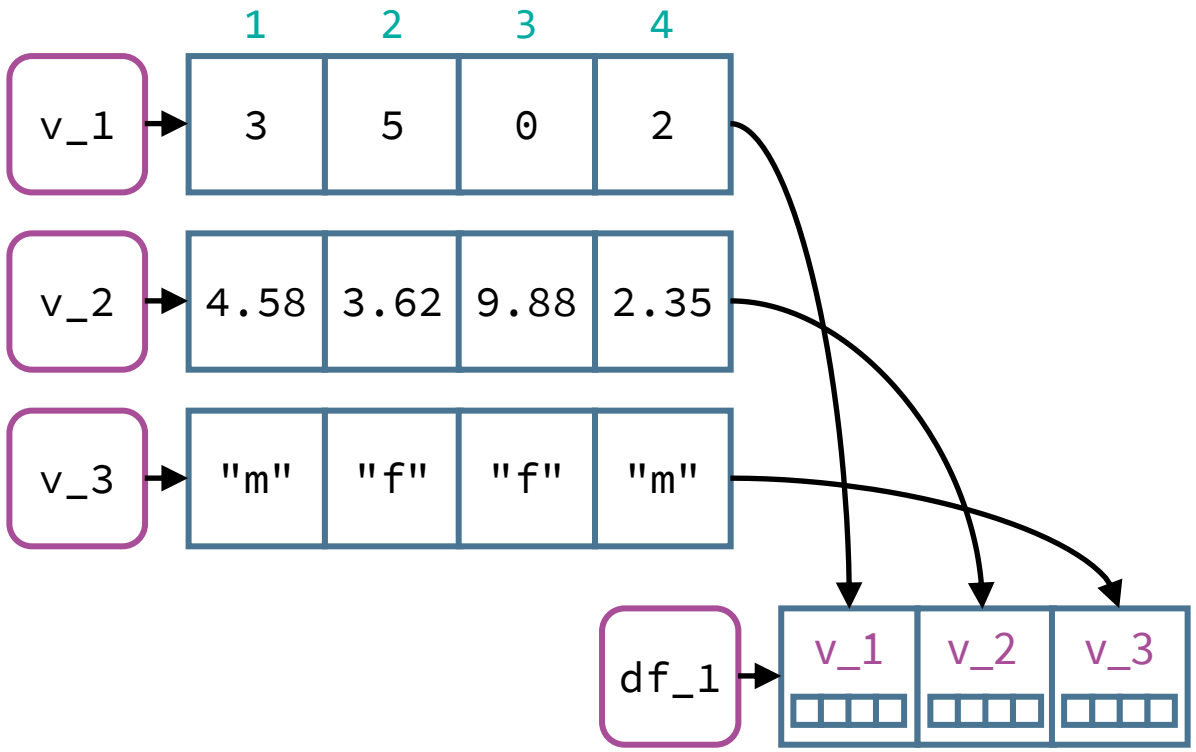


Factor infos

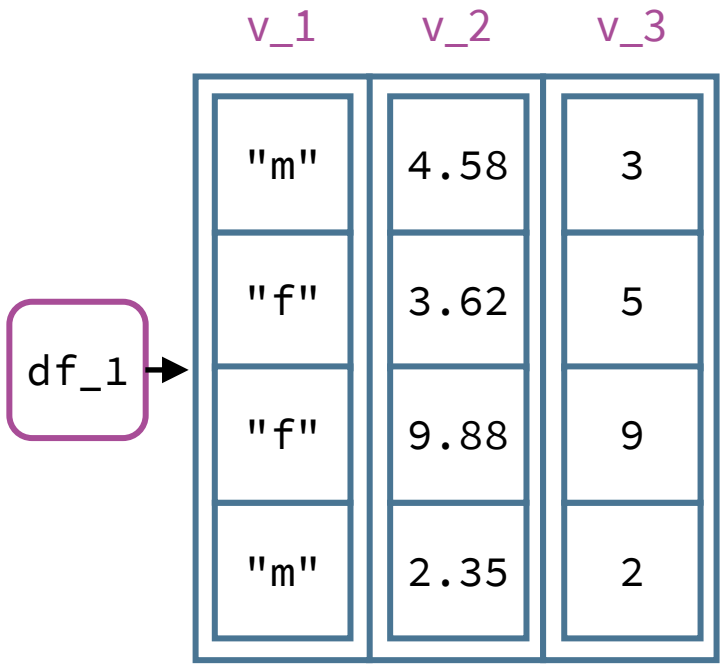
- Factors have **levels** (and optional labels)
- **Empty levels** are possible and a feature
- The **order of levels** is important (especially for plotting graphics and analyses)
- Many **analysis functions** treat factors specially

Data Frames…

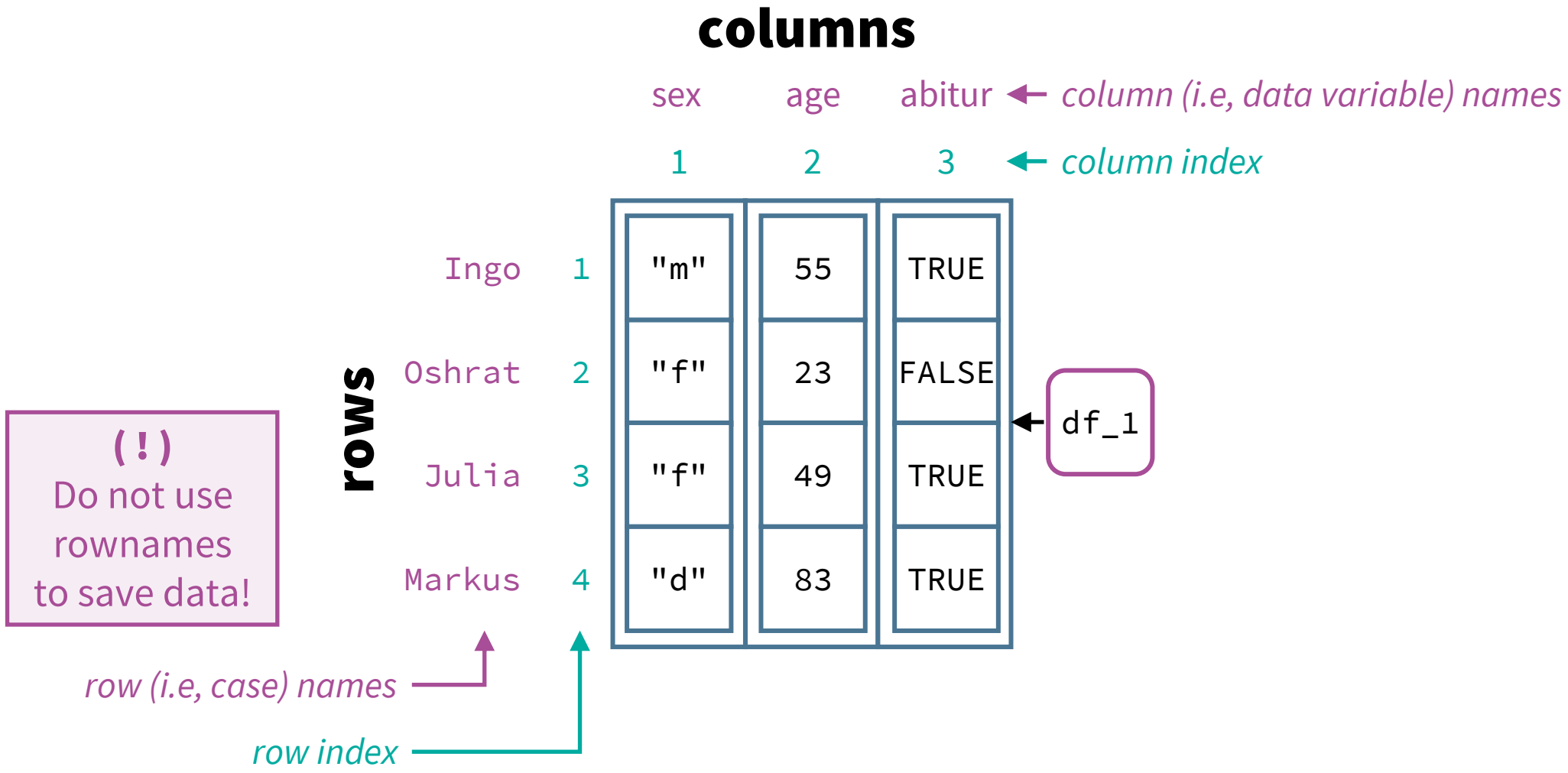
…are collection of vectors where all vectors have the same length



…are „tables“ with rows (observations) and columns (variables)



Data frame anatomy



Basic data frame **commands** (for a data frame „df“)

1. Names and dimensions

`names(df)` list all variable names in „df“ `length(df)` number of columns/variables

`nrow(df)` number of rows/observations

2. Inspecting a dataframe

`df` prints an excerpt `view(df)` opens Rstudio‘s graphical data viewer

`glimpse(df)` list variables and a preview of their content vertically

Construct a data frame (~tibble)

Sometimes, we want to create small data frames on the fly.
Treat **tibble** synonymously to **data frame** for now.

Column wise creation

`tibble()`

```
df <- tibble(
  a = c(1, 2, 3, 4),
  b = c(3.4, 4.6, 6.9, 9.2),
  c = c(TRUE, FALSE, FALSE, TRUE)
)
```

Row wise creation

`tribble()`

```
df <- tribble(
  ~a, ~b, ~c,
  1,  3.4, TRUE,
  2,  4.6, FALSE,
  3,  6.9, FALSE,
  4,  9.2, TRUE
)
```

Import data files to R with packages

`readr::` comma separated (**csv**) and tab separated (**tsv**) files

`readxl::` xls and xlsx Microsoft **Excel** files

`haven::` **STATA** .dat, **SPSS** .sav, and **SAS** .sas7bdat .sas7bcat

haven::

- The **Haven** package imports SPSS (and STATA / SAS) data.
- It preserves variable labels, value labels, and user missing values
- However, those structures are not native to R (!)
- The next session will go into detail of how to deal with this efficiently

```
library(haven)
new_df <- read_spss("some_data.sav", user_na = TRUE)
```

(!)
user_na = TRUE
means that custom SPSS missings are preserved
Without the argument, haven replaces them with R NAs

Working directories (how does R find files?)

The console and R scripts look for files in the current working directory

`getwd()` prints the current working directory

`setwd()` sets a new working directory

File paths (on your hard drive)

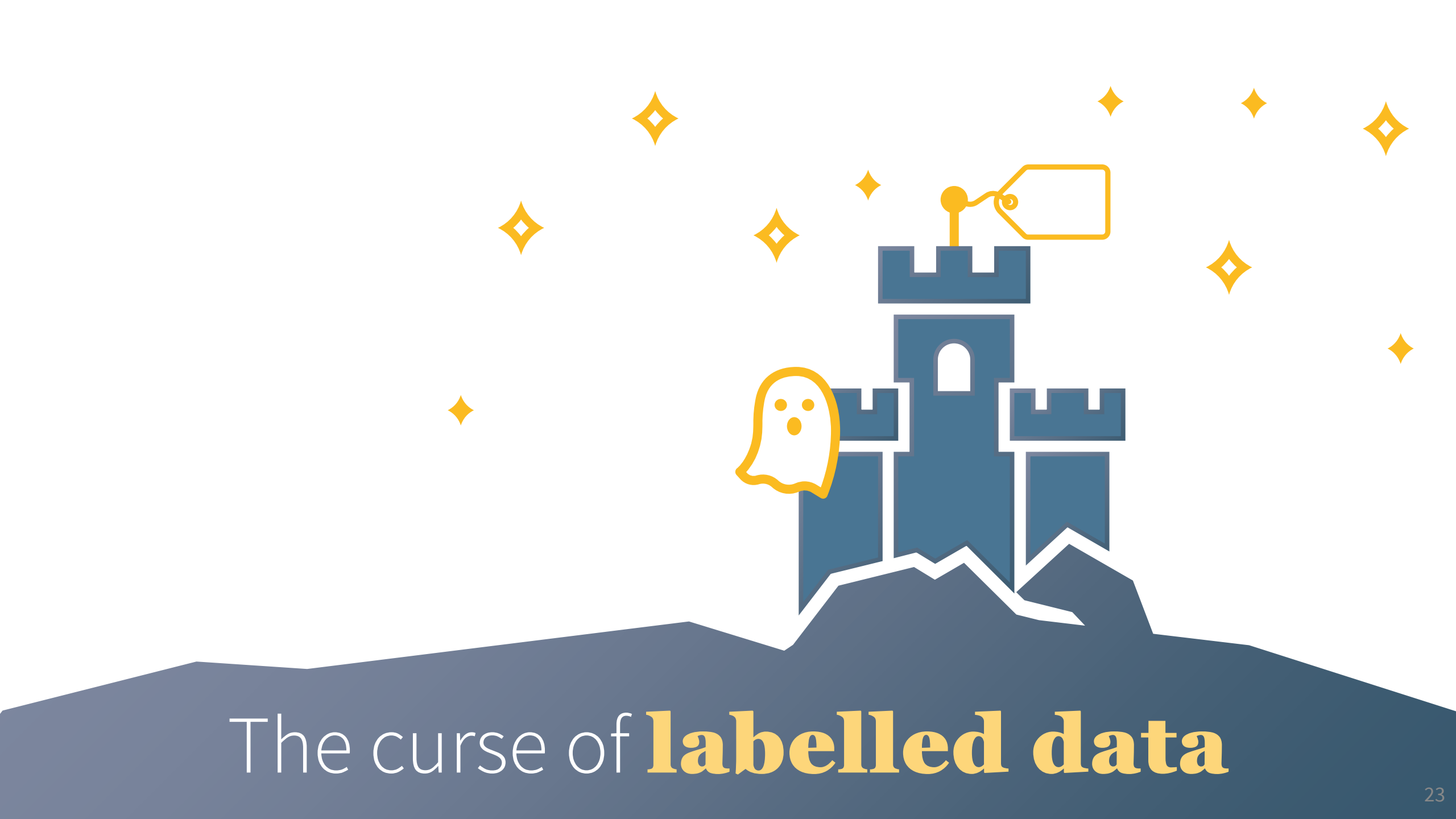
Different operating systems use different syntax. Thus, use file.path()

```
file.path("C", "Dropbox", "R_Scripts", "some_file.R")
```

```
#>[1] "C/Dropbox/R_Scripts/some_file.R"
```

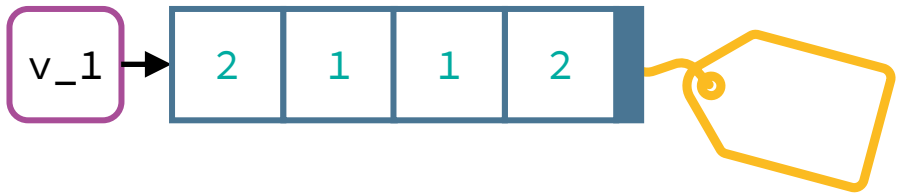
File Paths in **R Markdown**

(!) R Markdown looks for files in the same directory as the markdown file!



The curse of **labelled data**

„Labelled“ Data



- Variables in SPSS and STATA are often augmented („labelled“) with **documentation information**
- Such **metadata** might include:
  - **Variable label**, describing the variable (e.g., the question wording or the concept measured)
  - **Value labels**, describing the „meaning“ of some or all possible data values (e.g., the response option labels)
  - **User defined missing values**, which indicate that some data values should be treated as missing.

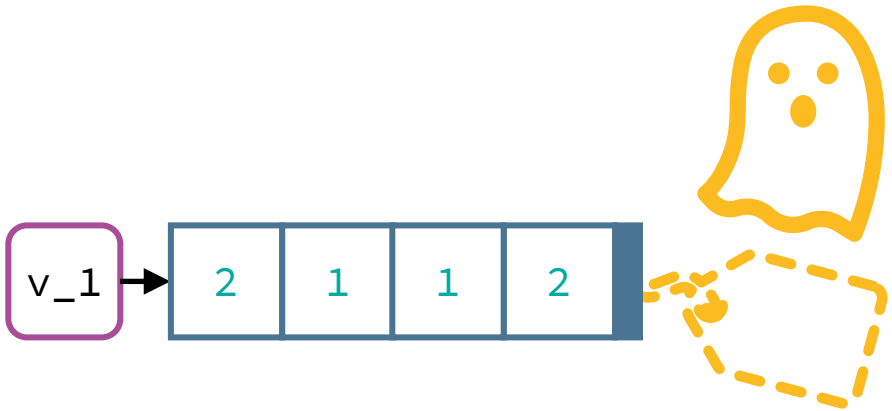
Conceptual problems

- **Data-documentation chimeras**
  Labelled variables combine data, documentation, and even paradata (e.g., „don‘t know“)

- **Ordinal-metric hybrids**
  Variables are often treated both as ordinal and metric:

  e.g., likert-scales (e.g., fully agree … not agree)
  - Ordinal: „34% of respondents at least somewhat agree that…“
  - Metric: Regression of income on agreement

Technical problems

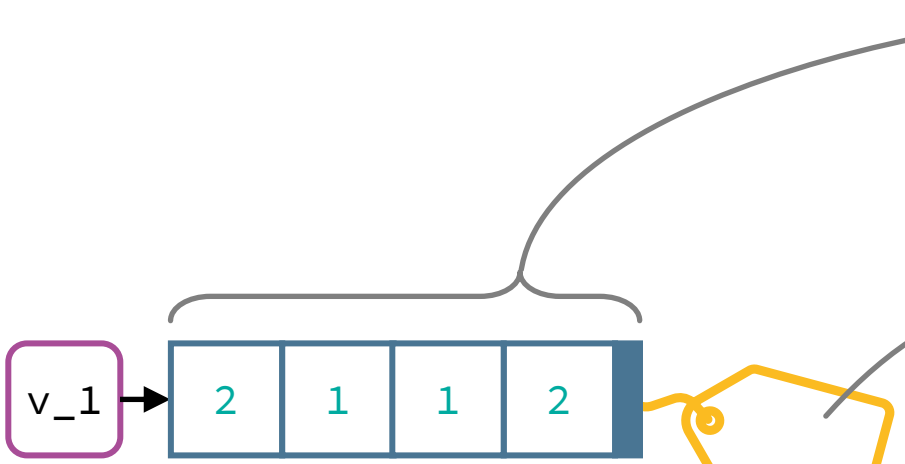


SPSS/STATA Labels are ***ephemeral*** in R (i.e, easily lost when working with the variable)

Most **R functions and packages** were not designed for foreign labelled data

We lose **context sensitive R behavior**, that depends on standard R data types

Transforming labelled data into **native R**



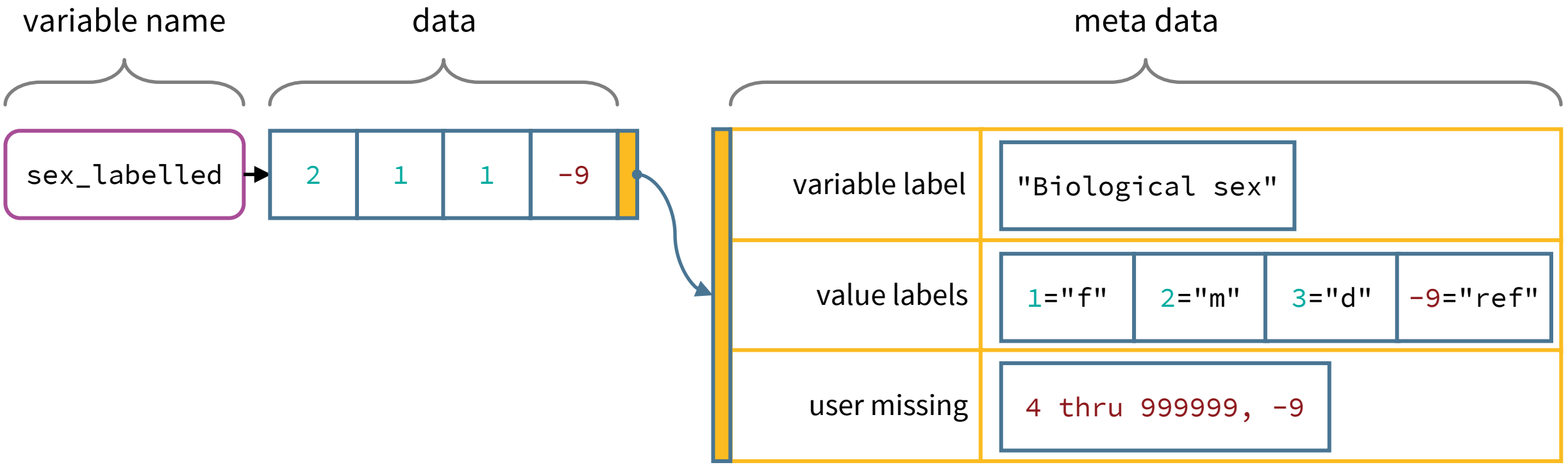
Save the data
Convert to (numeric) native R vector
with `zap_labels()`

Save the value labels
Convert to native R factor
with `as_factor()`

Save the variable labels
Extract a **data frame** with
all variable names and labels

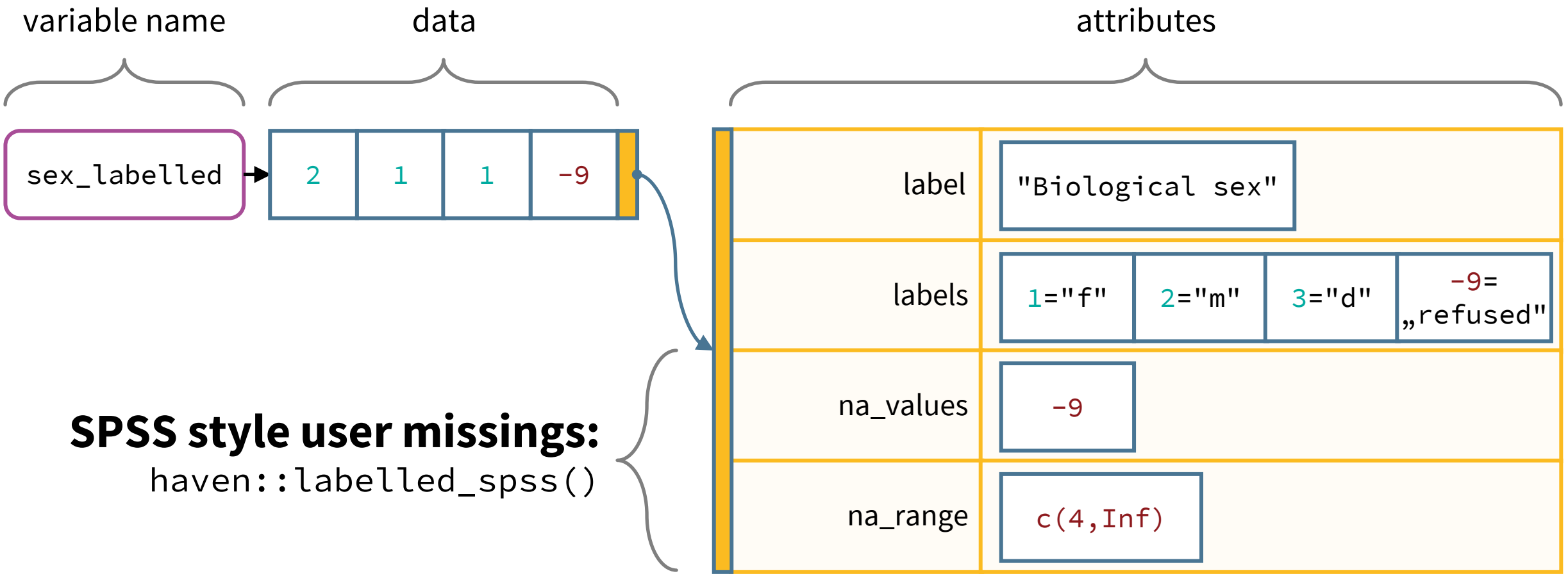
The structure of labelled data (SPSS)

BONUS ☺



Labelled data in R with `haven::labelled()`

BONUS ☺

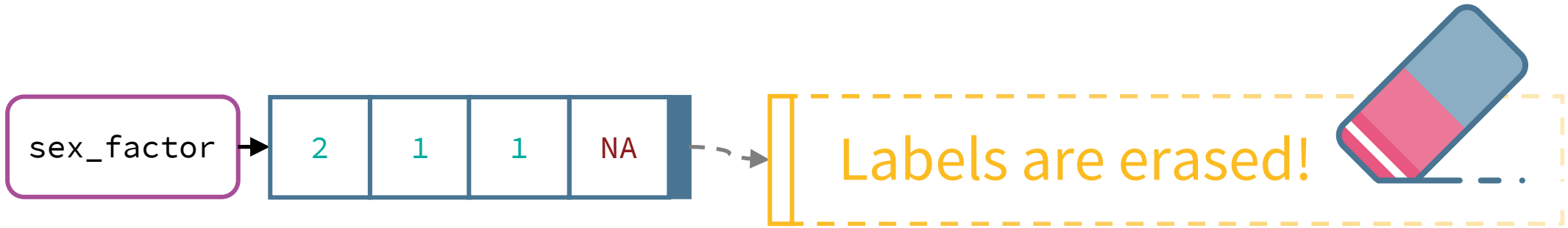


SPSS style user missings:

`haven::labelled_spss()`

Native R: numeric vectors

`haven::zap_labels(labelled_var)`

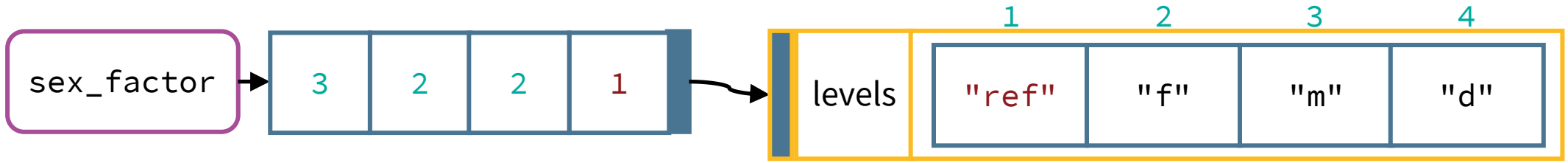


Numerical values are preserved (→ metric analyses)

However:

- Value labels are removed completely
- User defined missing turned to native R NAs

Native R: factor vectors
haven::as_factor(labelled_var)



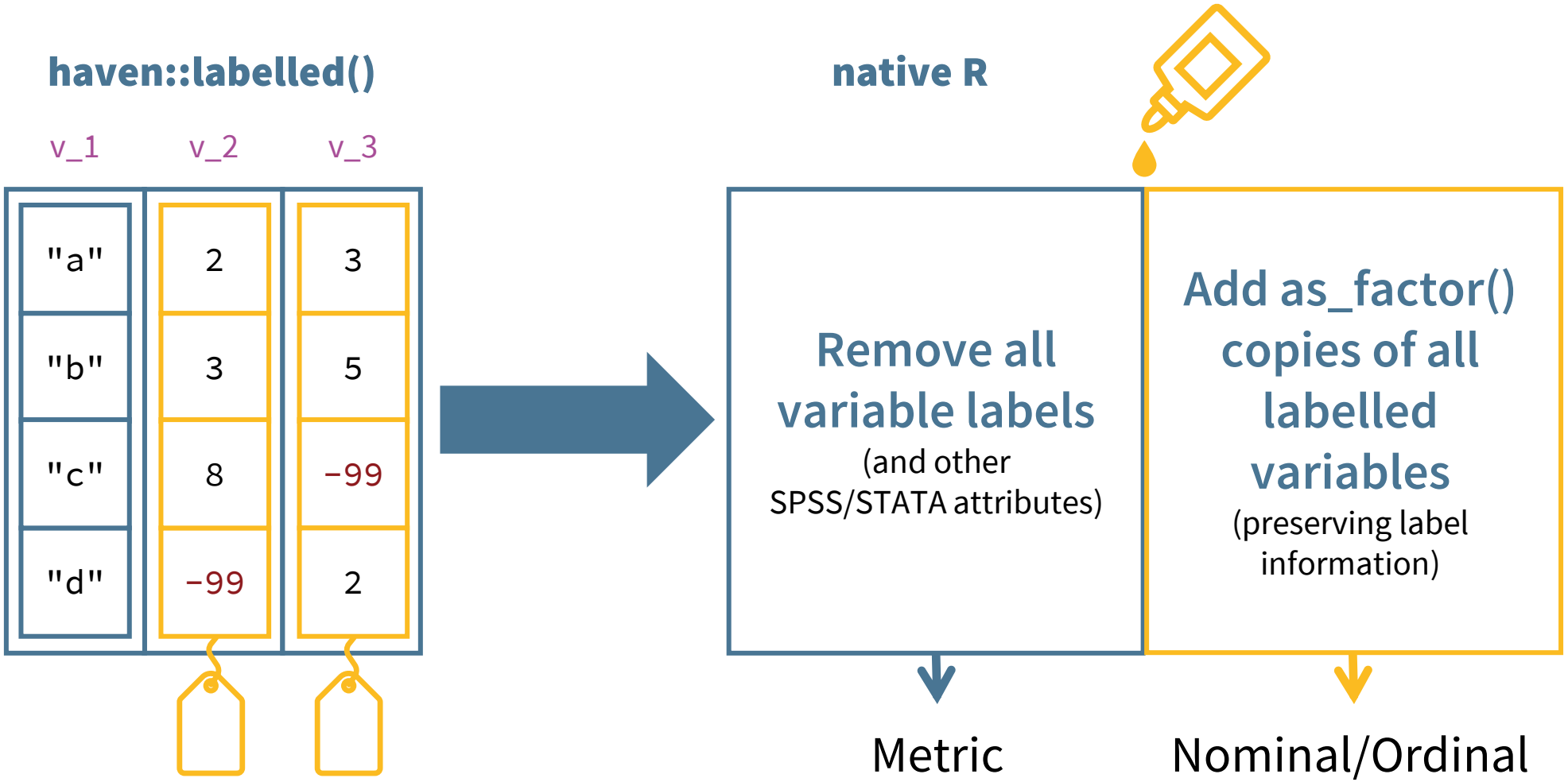
Value labels and label order are preserved (→ nominal and ordinal analyses)

- Numerical information is standardized to 1, 2, …, n (note: -9 is now 1)
  - ⚠ Take care when converting factors back to numbers!

```
as.numeric(labelled_var)              #> [1] 2  1  1 -9
as.numeric(as_factor(labelled_var))   #> [1] 3  2  2  1
```

- User defined missing lose their „treat as missing“ information
- R factors cannot be used as metric variables in analyses and plots

Transform SPSS/STATA Data into **native R** (but preserving value label information)



Transform SPSS/STATA Data into **native R**

```
allbus_df <- naturalize_labelled_df(ALLBUS_2018_labelled)
```

haven::labelled()

v_1	v_2	v_3
"a"	2	3
"b"	3	5
"c"	8	-99
"d"	-99	2

native R

v_1	v_2	v_3	v_2_**fct**	v_3_**fct**
"a"	2	3	lab2	lab3
"b"	3	5	lab3	lab5
"c"	8	NA	lab8	Lab-99
"d"	NA	2	Lab-99	lab2

Text variables remain unchanged

Labelled variables become **numeric** with R **NA**s

A **factor copy** of every labelled variable is added to preserve labels

Note that the factor copies get an added **_fct** suffix.

_fct factor copies

naturalize_labelled_df() can also add suffixes to the numeric or the text variables, if desired.

The right variable version for the job

Metric analyses with unlabelled vars (e.g., mean, sd, linear models)

```
summary(allbus_df$pa02a)
```

```
#>  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#> 1.000   2.000   3.000   2.696   3.000   5.000       2
```

_fct Nominal or ordinal analyses (e.g., Frequency tables or bar charts)

```
summary(allbus_df$pa02a_fct)
```

```
#> KEINE ANGABE       SEHR STARK            STARK           MITTEL
#>            2              385              943             1605
```

Data frame with **variable and value labels**: `register_labels(df)`

```
allbus_labels <- register_labels(ALLBUS_2018_labelled)
```

var_name	variable_label	value_labels
za_nr	STUDY NUMBER	[5271] ALLBUS COMPACT 2018
doi	DIGITAL OBJECT IDENTIFIER	[]
german	GERMAN CITICENSHIP?	[1] YES; [2] NO
pa02a	POLITICAL INTEREST (RESP.). (ORDINAL)	[-9] NO ANSWER; [1] VERY STRONGLY; [2] STRONGLY; [3] MIDDLING; [4] SOMEWHAT; [5] NOT AT ALL

Fetch **labels**

Variable Labels

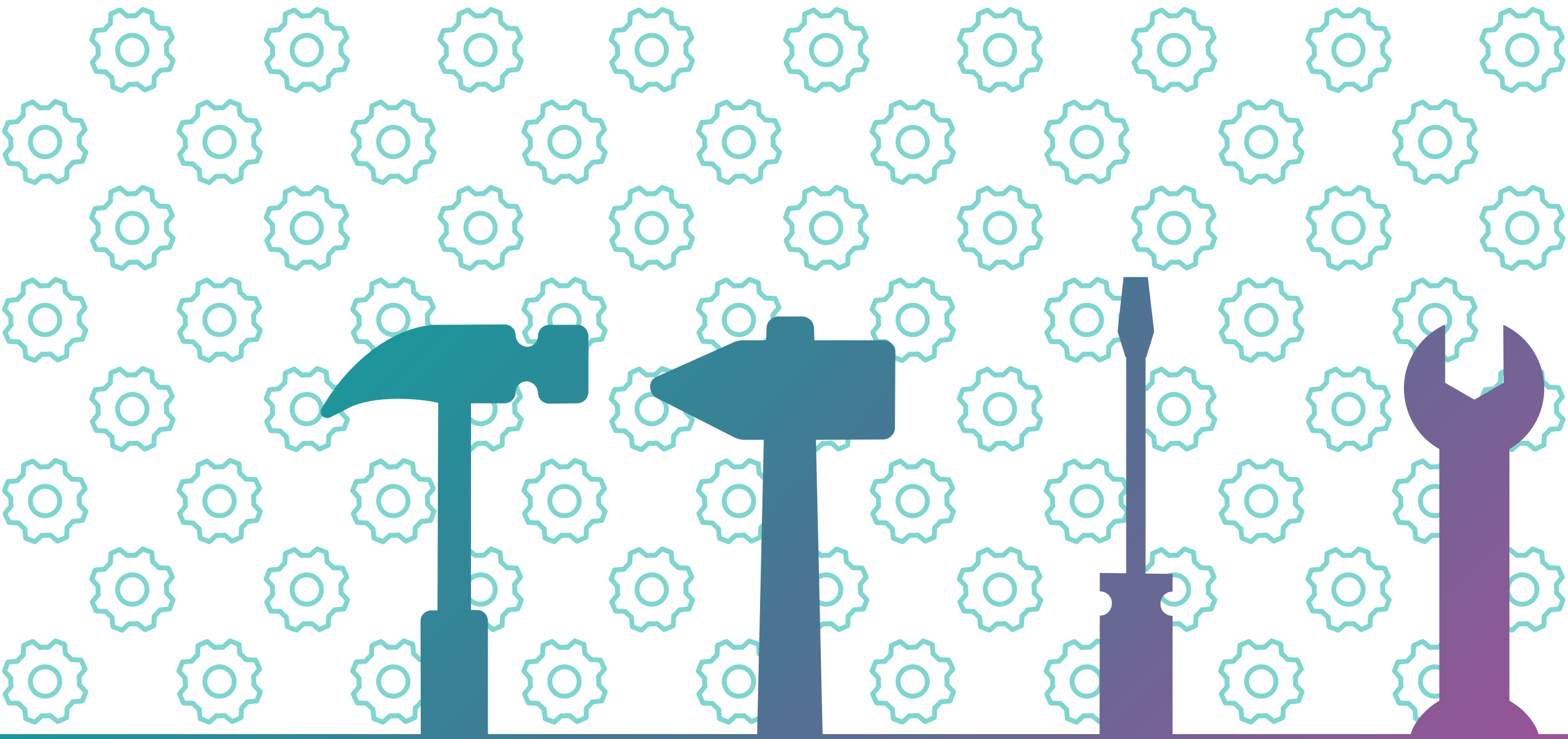
```
fetch_var_lab("pa02a", allbus_labels)
```

```
#> [1] "POLITISCHES INTERESSE, BEFR. (ORDINAL)"
```

Value Labels

```
fetch_val_lab("pa02a", allbus_labels)
```

```
#> "[-9] KEINE ANGABE; [1] SEHR STARK; [2] STARK
```



Data Wrangling

Pipes

Chaining multiple actions

Shaping Dataframes

Choosing variables and cases

-

Clean missing values

-

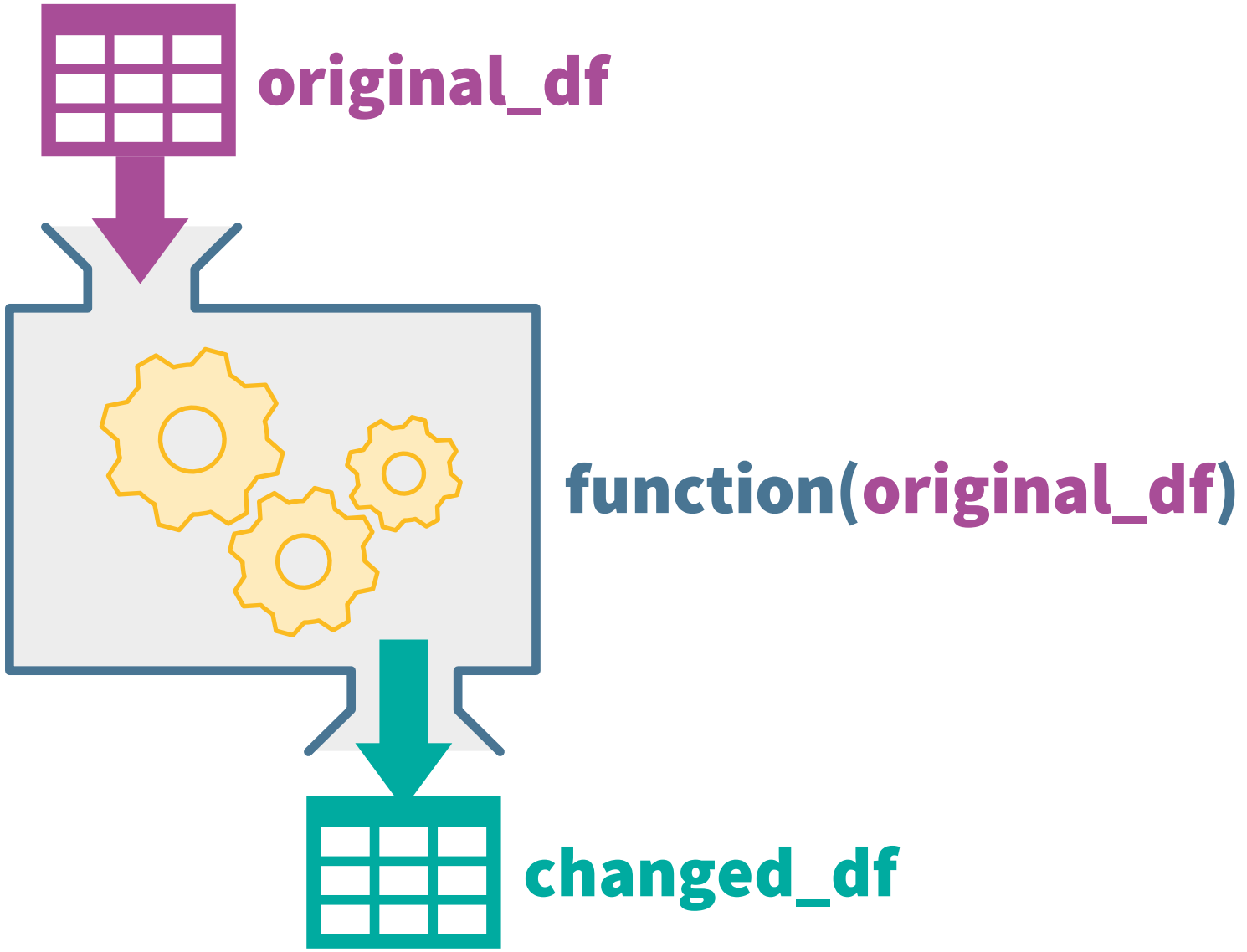
Order variables and cases

Transforming Data

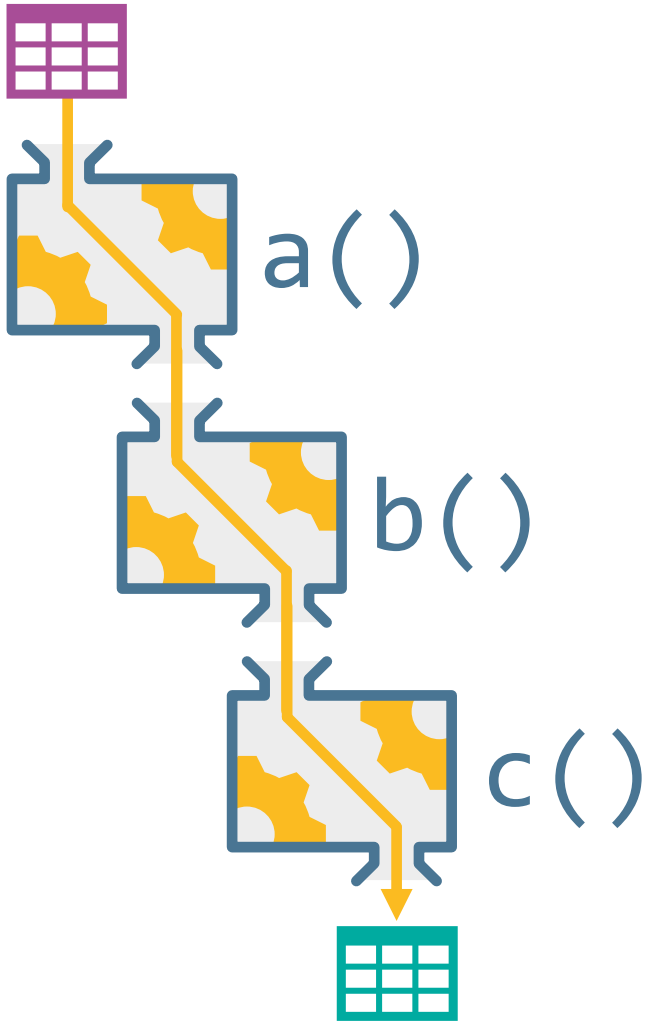
Recode variables

-

Computing and transforming values

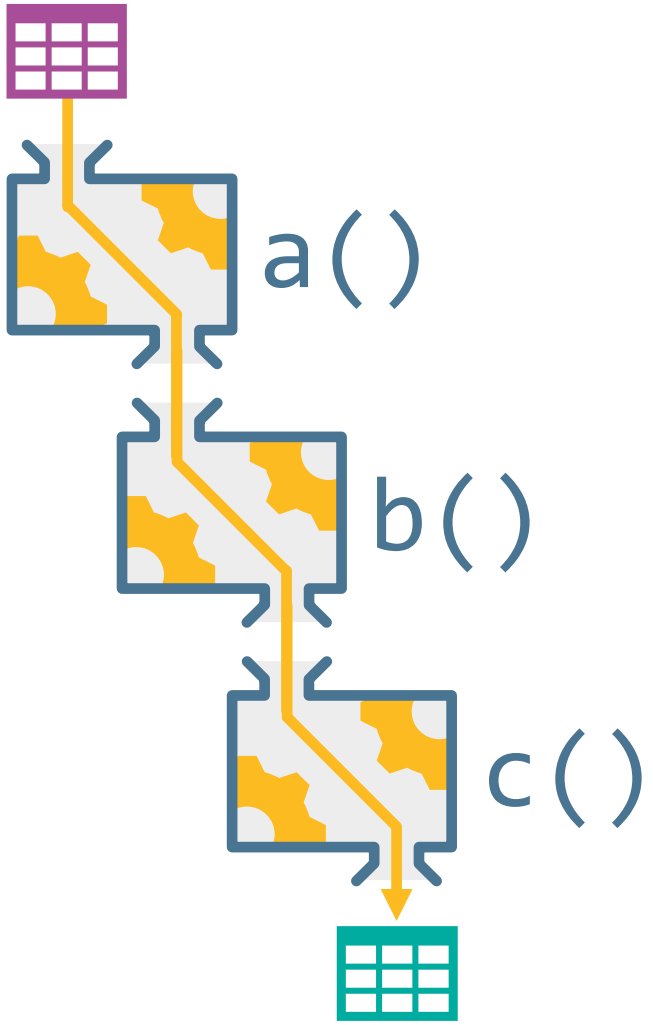

original_df
function(original_df)
changed_df

Applying many changes to df in a row



Applying many changes to df in a row

BONUS ☺



1. Nested functions

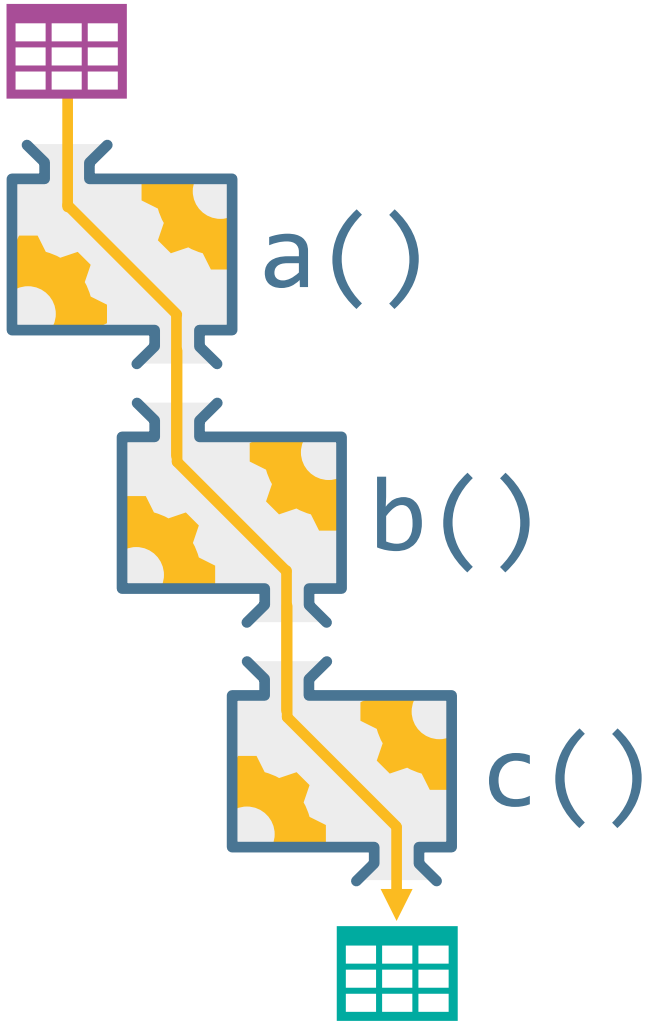
```
# nested functions
new_df <- c(b(a(df)))

# with arguments
new_df <- c(b(a(df, some_argument = "x"),
                some_argument = „y"),
              some_argument = "x")
```

Note: Functions are written in inverted order of execution.
It also becomes hard to tell which argument belongs to which function.

Applying many changes to df in a row

BONUS ☺

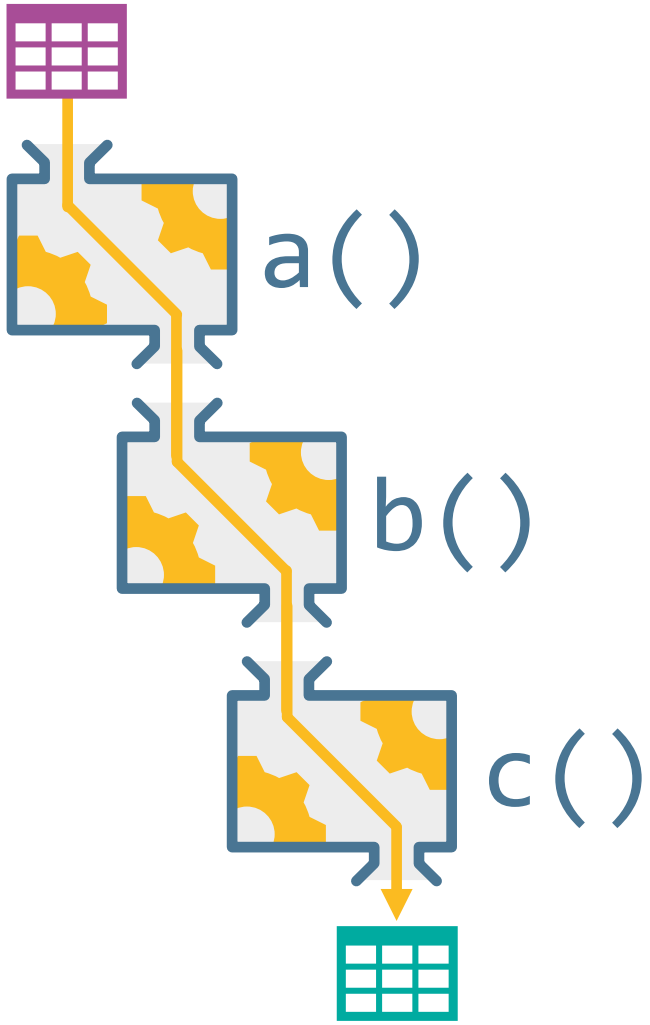


2. Intermediate objects

```
df_a <- a(df, some_argument = "x")
df_b <- b(df_a, some_argument = "y")
new_df <- c(df_b, some_argument = "z")
```

Note: This approach is more readable, and the order is correct.
However, we have to type out two data frames for each process step.

Applying many changes to df in a row



3. Magrittr Pipe

```
df %>%
  a(some_argument = "x") %>%
  b(some_argument = "y") %>%
  c(some_argument = "z") -> new_df
```

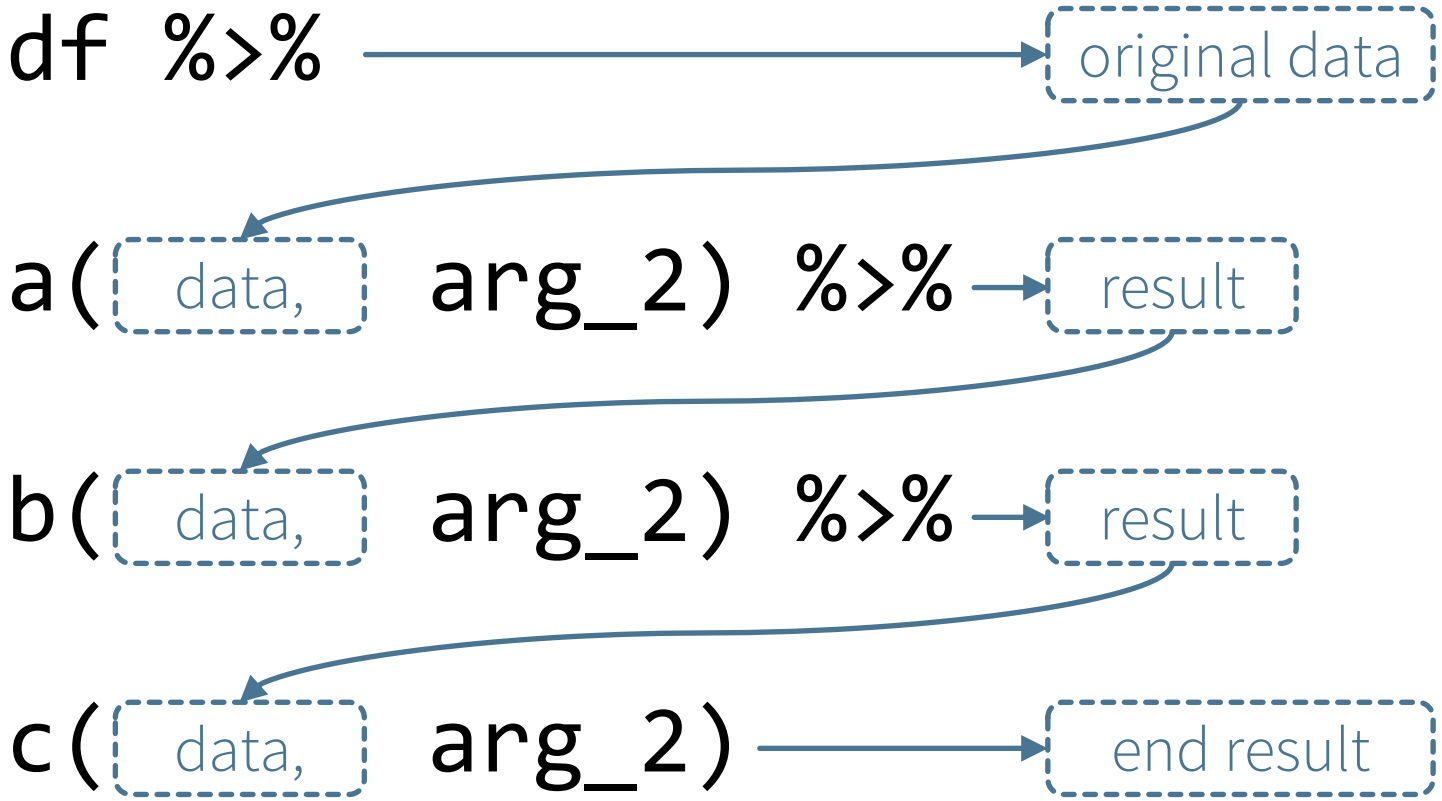
The pipe syntax is in correct order and we type start and end dataframe only once.

Tipp:
Do not type the piper operator %>%
Instead you can use a shortcut in RStudio

Ctrl + ⇧Shift + M

Using the **magrittr pipe** %>%

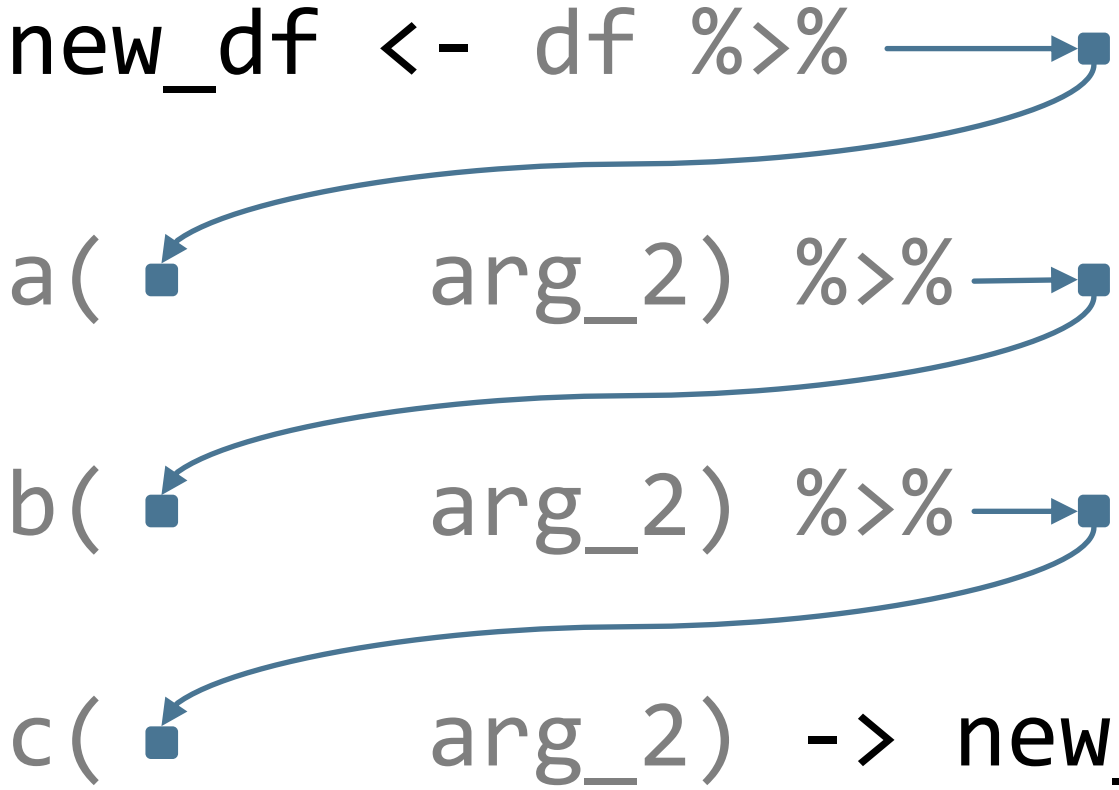
Ctrl + ⇧Shift + M



Using the **magrittr pipe** %>%

Ctrl + ⇧Shift + M

Save results with assignment here…

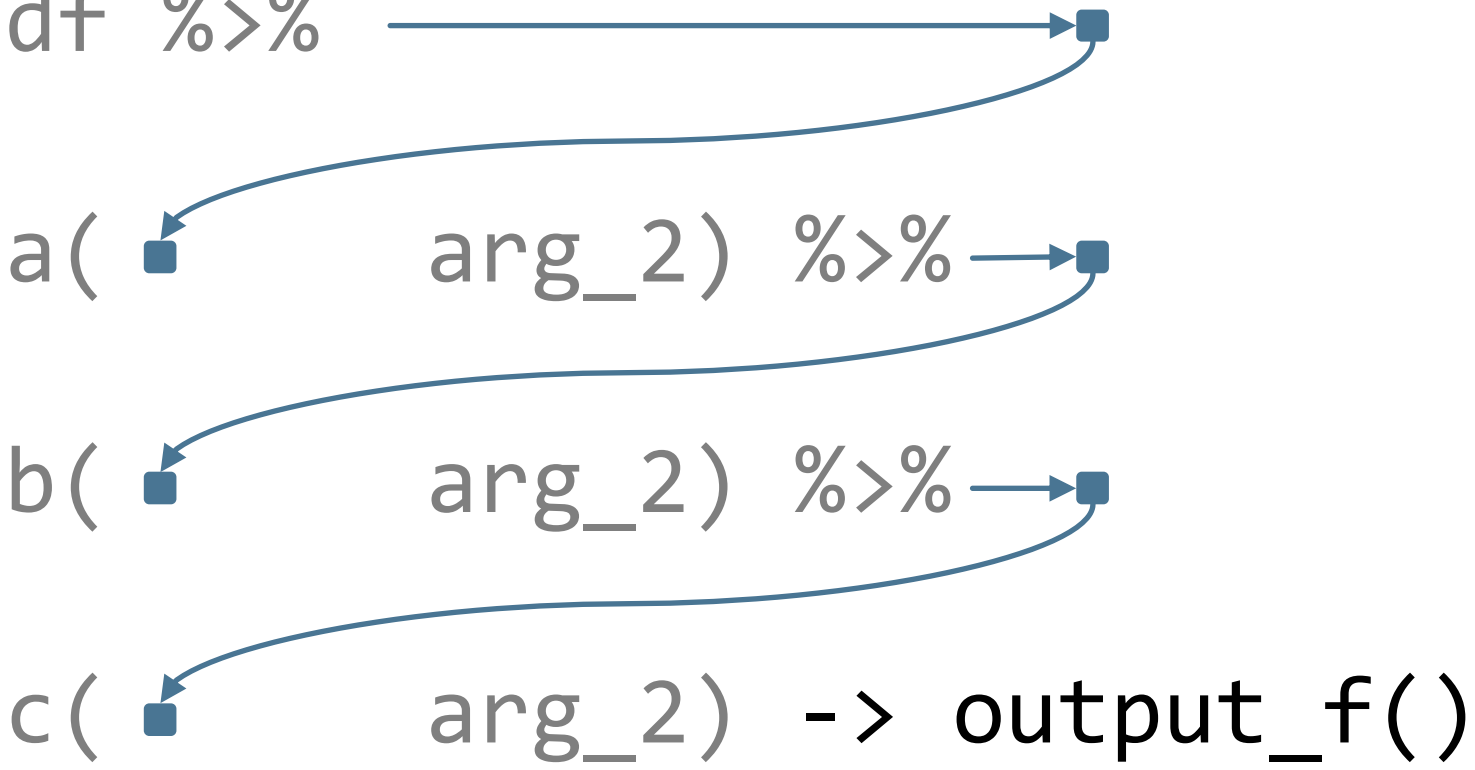


…or here (but not both!)

Using the **magrittr pipe** %>%

Ctrl + ⇧Shift + M

Piping directly into an output function




```
e.g.,
ggplot()
kable()
lm(y~x, data =.)
```

If data is not the first argument, you can supply data with a period (.)

Shaping data frames

We will use „verbs“ from the dplyr package

`filter()`		Choose which rows (i.e., observations) to keep
`drop_na()`		Remove rows with missing values
`sample_n()`	etc.	Choosing a certain number of rows randomly
`select()`		Choose which columns (i.e., variables) to keep
`rename()`		Rename columns (i.e., variables)
`arrange()`		Reoder rows (i.e., observations)

Lastly, merging and pivoting data frames will be teasered.

Choosing **rows** (i.e., observations, cases etc.)

filter() – choosing rows

filter() allows us to choose rows with conditional expressions
(filter keeps where the expression is TRUE)

df

age	uni_degr	sex
22	FALSE	"m"
34	TRUE	"f"
46	FALSE	"m"
58	FALSE	"f"
67	TRUE	"f"

df_new

age	uni_degr	sex
22	FALSE	"m"
34	TRUE	"f"
46	FALSE	"m"
58	FALSE	"f"
67	TRUE	"f"

filter() – choosing rows

filter() allows us to choose rows with conditional expressions
(filter keeps where the expression is TRUE)

„Keep respondents with university degree:“

df

age	uni_degr	sex
22	FALSE ✗	"m"
34	TRUE	"f"
46	FALSE ✗	"m"
58	FALSE ✗	"f"
67	TRUE	"f"

```
df %>%
  filter(
     uni_degree
  ) ->  df_new
```

df_new

age	uni_degr	sex
34	TRUE	"f"
67	TRUE	"f"

filter() – choosing rows

filter() allows us to choose rows with conditional expressions
(filter keeps where the expression is TRUE)

„Keep respondets with female sex:“

df

age	uni_degr	sex
22	FALSE	"m" ✗
34	TRUE	"f"
46	FALSE	"m" ✗
58	FALSE	"f"
67	TRUE	"f"

```
df %>%
  filter(
     sex == "f"
  ) ->  df_new
```

df_new

age	uni_degr	sex
34	TRUE	"f"
58	FALSE	"f"
67	TRUE	"f"

filter() – choosing rows

filter() allows us to choose rows with conditional expressions
(filter keeps where the expression is TRUE)

„Keep respondets who are at least 40 years old.“

df

age	uni_degr	sex
22	FALSE	"m" ✗
34	TRUE	"f" ✗
46	FALSE	"m"
58	FALSE	"f"
67	TRUE	"f"

```
df %>%
  filter(
    age >= 40
    ) ->  df_new
```

df_new

age	uni_degr	sex
46	FALSE	"m"
58	FALSE	"f"
67	TRUE	"f"

BONUS ☺

filter() – choosing rows

filter() allows us to choose rows with conditional expressions
(filter keeps where the expression is TRUE)

1. Relational operators

`==` equal `!=` not equal `is.na(x)` check for NA `near(x,y)` equal w. tolerance

`<` less than `>` greater than `<=` less or equal `>=` greater or equal

2. Logical operators

`&` and (all must be true)

`|` or (some must be true)

`!` not (the opposite please)

A	B	A&B	A|B	!A
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

filter() – choosing rows

BONUS ☺

filter() allows us to choose rows with conditional expressions
(filter keeps where the expression is TRUE)

3. String operators for character vectors and factors

`==` equal `!=` not equal

`str_detect(var, „string“)` checks for parts of strings

Example:

```
chr_var <- c("apple_pie", "cherry_pie", "blueberry_muffin")
str_detect(chr_var, "_pie")
```

```
# [1] TRUE TRUE FALSE
```

`drop_na()` – convenient NA filtering

`drop_na()` is a special filter for excluding rows with missing values:

- either in any variable:
  ```
  df %>% drop_na()
  ```
- or in specific variables:
  ```
  df %>% drop_na(age, uni_degree)
  ```

Alternatively, use `filter()` with `is.na()`:

```
df %>% filter(!is.na(age), !is.na(uni_degree))
```

Choosing n rows

`sample_n(df, n)`
Choose n random rows

`sample_n()` is great to get a feeling for a dataset

`head(df, n)`
Choose the first n rows

`tail(df, n)`
Choose the last n rows

`df[n:k,]`
Choose rows n to k

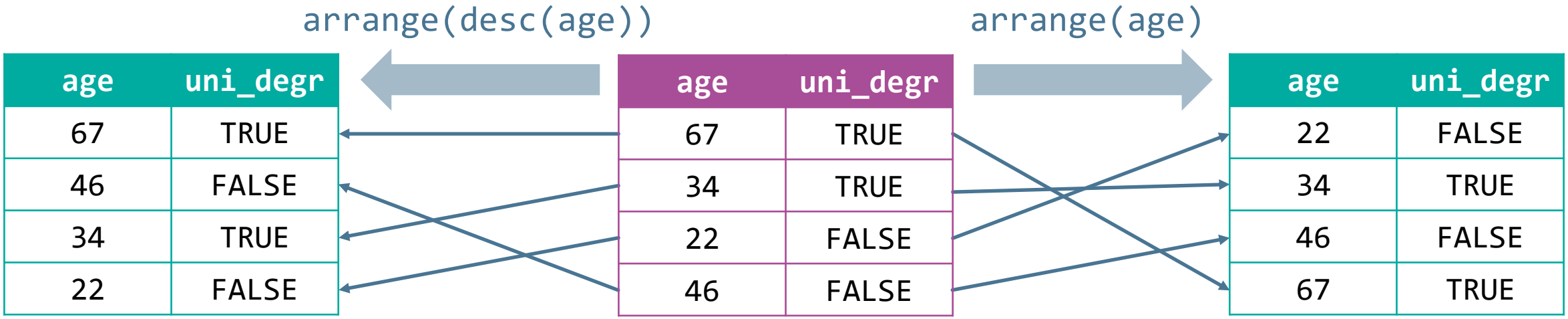
Note the comma!
`df[rows, columns]`

Without a comma, we only select columns:
`df[columns]`

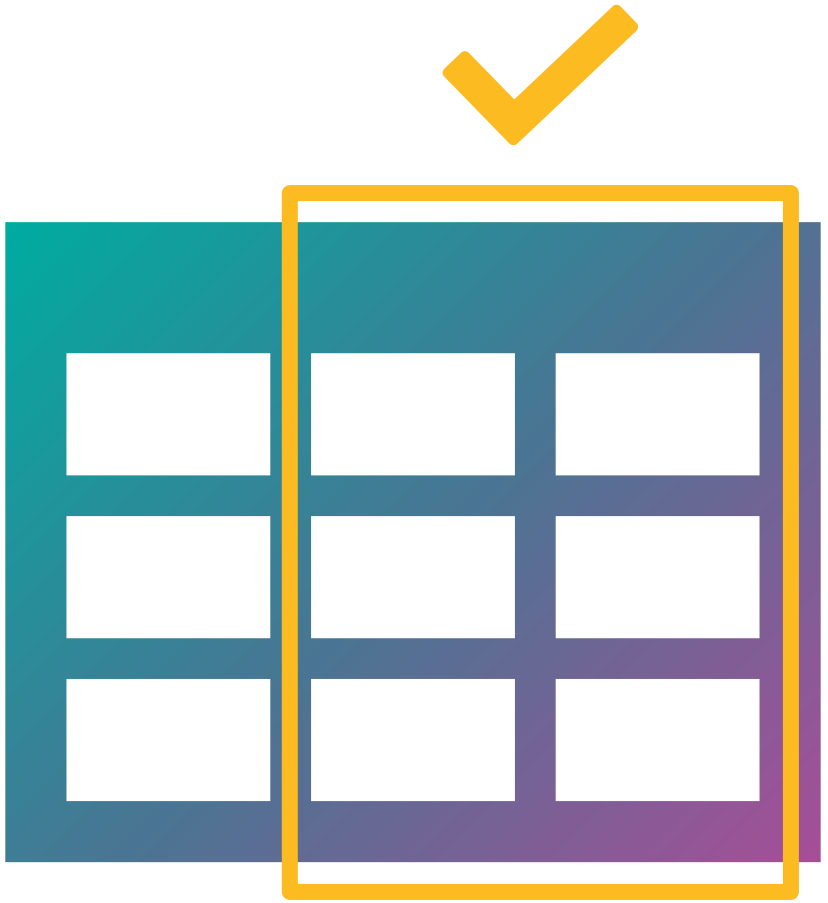
arrange() – sorting data frame rows

arrange() sorts data frame rows

- By default, arrange() sorts in ascending order (e.g., 1, 2, 3)
  - Change this by wrapping a variable in desc()
    arrange(desc(age))
- If several variables are named, arrange() will sort after the first, then break ties with the second, then third and so on.



Choosing **columns** (i.e., variables)



`select()` – choosing columns

select() allows us to choose columns with many convenient helpers

df

dem_age	dem_uni	dem_sex	scale_01	scale_02	scale_03	scale_04	scale_05
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

df_new

dem_age	dem_uni	dem_sex	scale_01	scale_02	scale_03	scale_04	scale_05
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

`select()` – choosing columns

select() allows us to choose columns with many convenient helpers

df

dem_age	dem_uni	dem_sex	scale_01	scale_02	scale_03	scale_04	scale_05
⋮	✗	⋮	✗	✗	✗	✗	✗

„Keep the variables I list explicitly.“

```
df %>%
  select(dem_age, dem_sex) ->
  df_new
```

df_new

dem_age	dem_sex
⋮	⋮

`select()` – choosing columns

select() allows us to choose columns with many convenient helpers

df

dem_age	dem_uni	dem_sex	scale_01	scale_02	scale_03	scale_04	scale_05
✗	✗	✗	⋮	⋮	⋮	⋮	⋮

„Only keep the „scale_“… variables.

```
df %>%
  select(starts_with("scale_")) ->
  df_new
```

df_new

scale_01	scale_02	scale_03	scale_04	scale_05
⋮	⋮	⋮	⋮	⋮

BONUS ☺

`select()` – choosing columns

select() allows us to choose columns with many convenient helpers

1. Basic operators and helpers

`v1, v2` Select v1 and v2 `age:polint` columns age, polint, an all in between

`everything()` All (remaining) columns; e.g., to reorder `select(x,y, everything())`

2. Text matching

`starts_with("")` `ends_with("")` `contains("")` `matches("regular expression")`

`num_range ("string", 3:25)`

3. Logical expressions

`&` and (all must be true) `|` or (some must be true) `!` not (the opposite please)

e.g., `select(starts_with("dem_") & ends_with("_fctr"))`

`rename()`

`rename()` chances the column names (i.e., variable names)

- Basic pattern: new name = old name
  ```
  df %>% rename(new_1 = old_1, new_2 = old_2)
  ```
- Non-syntactic names (e.g., with spaces): use back quotes
  ```
  `new name` = `old name`
  ```
- `rename()` does not change the order of variables
  Use `relocate()` instead.
- You can also rename with `select()`, but be careful not to unintentionally drop variables.

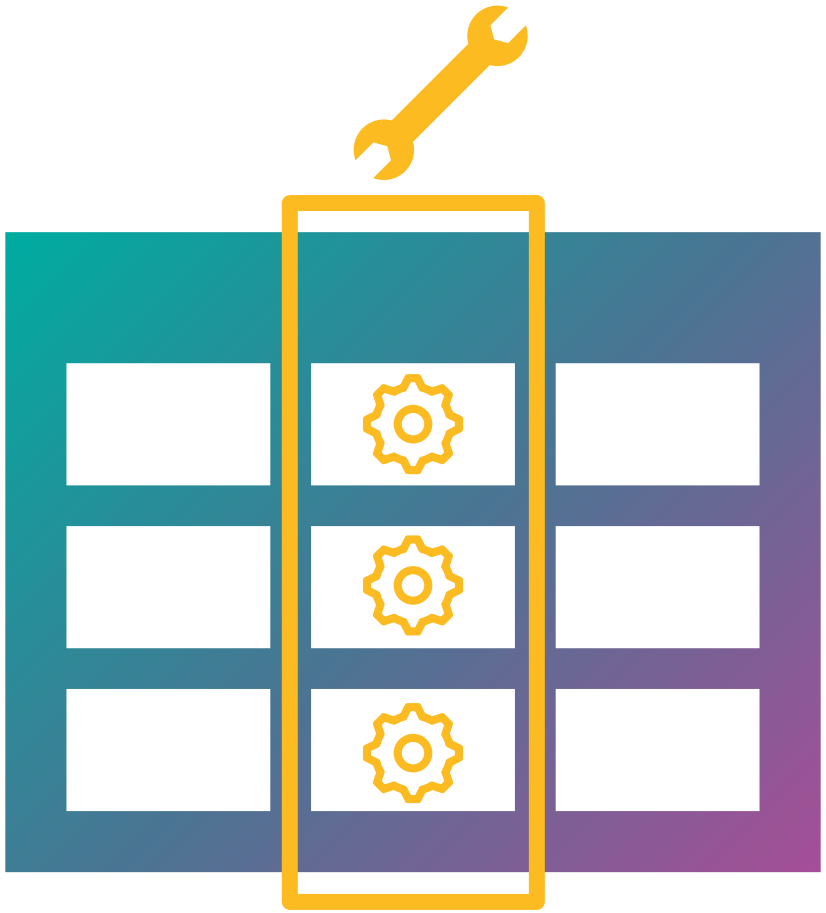
Chaining with the **Pipe %>%**

Ctrl + ⇧Shift + M

```
df %>%
  filter(dem_age > 40) %>%
  select(!starts_with("scale_") %>%
  rename(`university degree` = dem_uni ) ->
  df_new
```

Note:
`filter()` before `select()`

Transforming (i.e., mutating) variables



Mutating data frames

dplyr::mutate() transforms or generates variables

transmute() does the same, but discards all untransformed variables

Transforming or generating variables with:

- Numerical and logical operators
- Recoding & Helper functions
- Outlook:
  - Factor recoding via `forcats::`
  - String recoding via `stringr::`

`mutate()` basics

Generate new variables:

```
df %>% mutate(
  new_var = some_transformation(old_var)
)
```

Transform existing variable:

```
df %>% mutate(
  old_var = some_transformation(old_var)
)
```

`mutate()` basics

Multiple variables in one go:

```
df %>% mutate(
   v1_n = v1 * 2,
   v2_n = v2 - 5,
   v3 = v1_n + v2_n
)
```

You can immediately use newly created or transformed variables within the same `mutate()`

Vectorization

```
a <- c(1, 2, 3, 4)
b <- c(5, 6, 7, 8)
c <- a + b
```

a	+	b	=	c
1	+	5	=	6
2	+	6	=	8
3	+	7	=	10
4	+	8	=	12

(!) a, b, c all have the same **`length()`** which is equal to **`nrow(df)`** in a data frame

and vector recycling

```
a <- c(1, 2, 3, 4)
b <- 2
c <- a + b
```

a	+	b	=	c
1	+	2	=	3
2	+	2	=	4
3	+	2	=	5
4	+	2	=	6

b is „**recycled**“; i.e., repeated by **`length(a)`** or **`nrow(df)`** in a dataframe

`mutate()` – numerical operations

BONUS ☺

mutate() can perform numerical operations on variables

1. Mathematical operators

`+` plus `-` minus `*` times `/` divded `^` power `**` power (alt.)

`%%` modulus `%/%` integer division

2. Mathematical functions with single value results

`mean` `sd` `sum` `min` `max` **(!)** All return NA if any value is NA unless `(x, na.rm=TRUE)`

3. Mathematical functions with vector results

`scale(x)` z-score standardization `scale(x, scale = FALSE)` only mean centering

`cumsum` cumulative sum `cummean` `cummin` `cummax`

`mutate()` –relational and logical operators

BONUS ☺

mutate() can use relational and logical operators just like filter.
The result is a logical vector containing TRUE, FALSE and NA values.

1. Relational operators

`==` equal `!=` not equal `is.na(x)` check for NA `near(x,y)` equal w. tolerance

`<` less than `>` greater than `<=` less or equal `>=` greater or equal

2. Logical operators

`&` and (all must be true)

`|` or (some must be true)

`!` not (the opposite please)

A	B	A&B	A|B	!A
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

`mutate()` –recoding

1. `recode()`

```
recode(variable, old_value1 = new_value1 , old2 = new2)
```

(!)
Mind the order:
Old value then new value!

recode() numerical values with back quotes for the new values:

```
recode(num_var, `4` = 1, `3` =  2, `2` =  3, `1` = 4)
```

2. `if_else()` for vectorized conditional recoding

```
if_else(condition, value_when_true, value_when_false)
```

e.g., windsorizing a variable to a max_value

```
if_else(num_var > max value, max_value, num_var)
```

Note: `if_else()` is dichotomous, see **`case_when()`** for a polytomous solution

BONUS ☺

`mutate()` – helpers

There are helpers for mutate() to facilitate common tasks.

- `coalesce()` combines several variables by taking the first non-NA value. Do this for experimental data.

condition	A_resp	B_resp	coalesce(A_resp, B_resp)
A	3	~~NA~~	3
A	2	~~NA~~	2
B	NA	4	4
B	NA	6	6

- `na_if(variable, value_to_replace_with_NA)`
  Replace specific values with missing (NA)

`mutate()` – for strings and factors

BONUS ☺

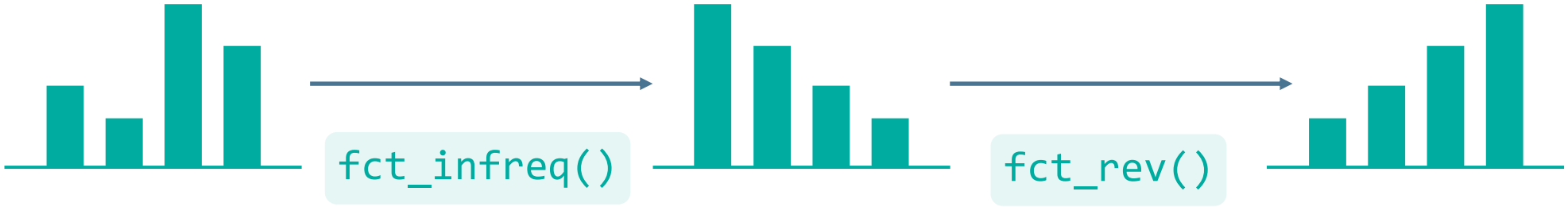
stringr:: package for string manipulation

`str_detect()` find substrings

`str_replace_all()` replace substrings

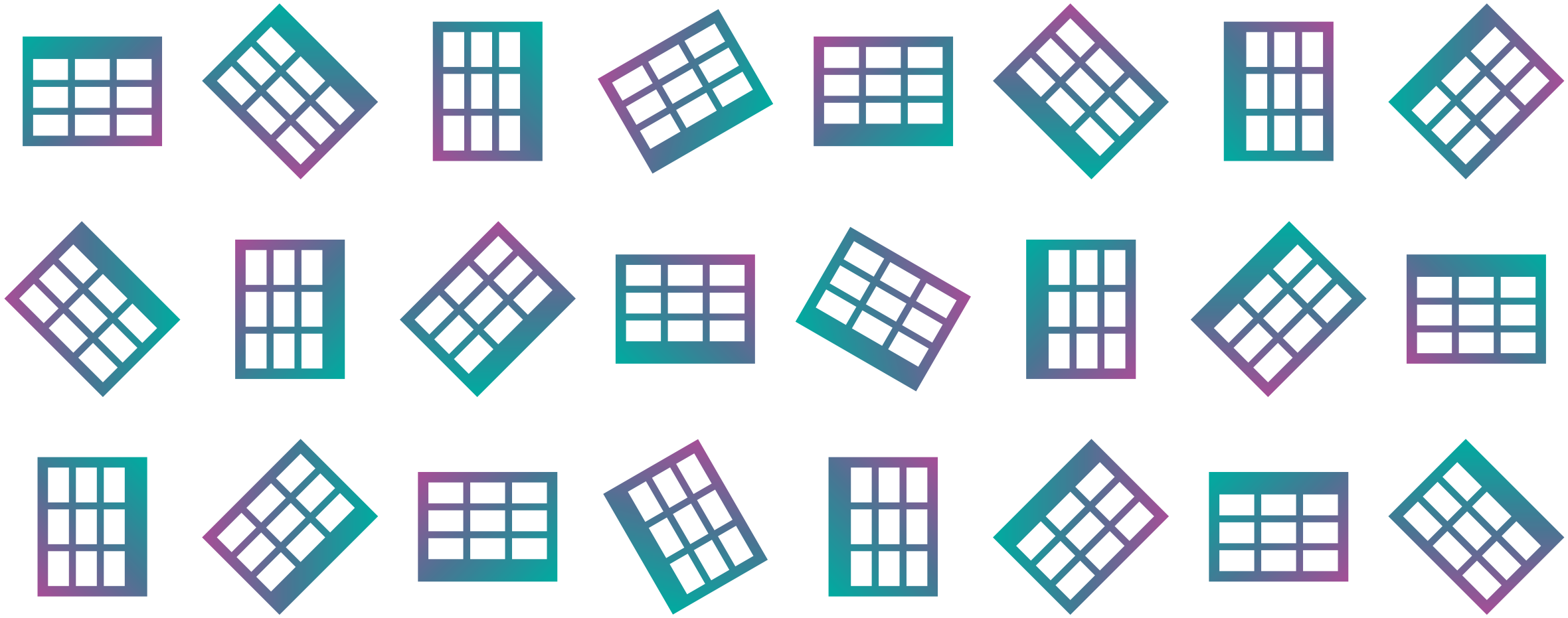
forcats:: package for factor manipulation

Recoding, but also rearranging factor levels (which, for example, influcences plots)



Bonus: Pivoting and Joining

BONUS ☺



BONUS ☺

Wide and long format: pivoting data frames

- Wide and long format are ways to structure a data table.
- E.g., multi-level data structures, such as panel surveys

Wide format – One row per subject

respondent	wave_1	wave_2	wave_3
A	1	2	3
B	4	5	6
C	7	8	9

Long format – One row per observation

respondent	wave	response
A	wave_1	1
A	wave_2	2
A	wave_3	3
B	wave_1	4
B	wave_2	5
B	wave_3	6
C	wave_1	7
C	wave_2	8
C	wave_3	9

BONUS ☺

Wide and long format: pivoting data frames

To transform a data frame from one form into the other, use:

- pivot_wider() to transform **long to wide**

```
long_df %>%
  pivot_wider(names_from = wave, values_from = response)
```

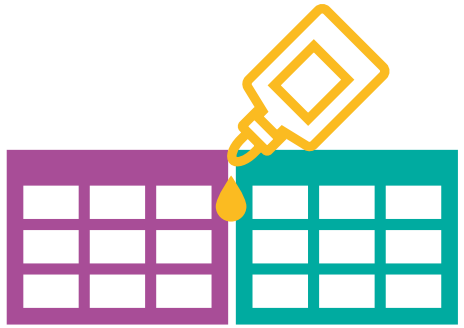
- pivot_longer() to transform **wide to long**

```
wide_df %>%
  pivot_longer(wave_1:wave_3, names_to = "wave", values_to = "response")
```

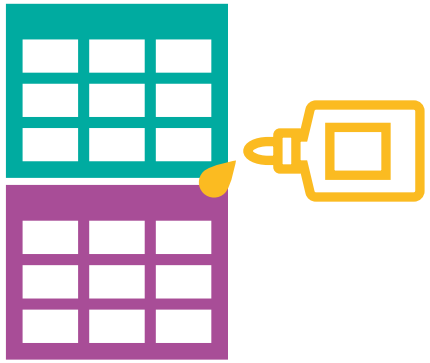
R for Data Science — Pivoting

BONUS ☺

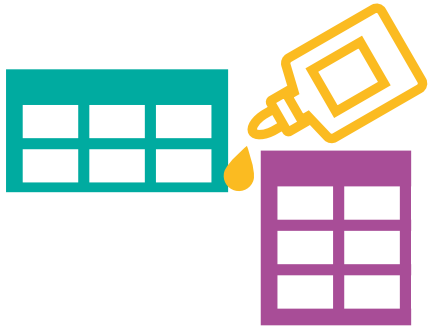
Adding columns, rows, and merging



`add_column(df_1, df_2)`
Glues columns together
- No duplicate variable names
- Same number of rows
- Rows are matching observations



`add_row(df_1, df_2)`
Glues rows together
- Same variable names
- Adds more observations

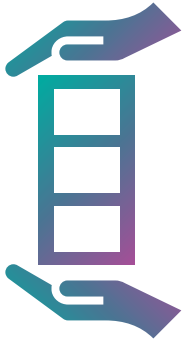


`Mutating joins`
inner_join(), left_join(), right_join(), full_join()

Merges dataframes of different sizes;
optionally via key variable

↗ R for Data Science — Relational Data

Data aggregation

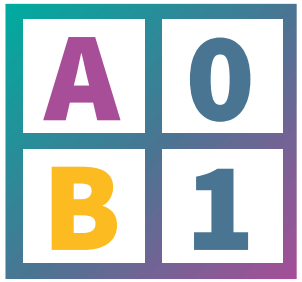


Aggregating Vectors

Calculating an aggregated value from a sequence of values

Grouping Dataframes

Structuring a dataframe based on one or more variable

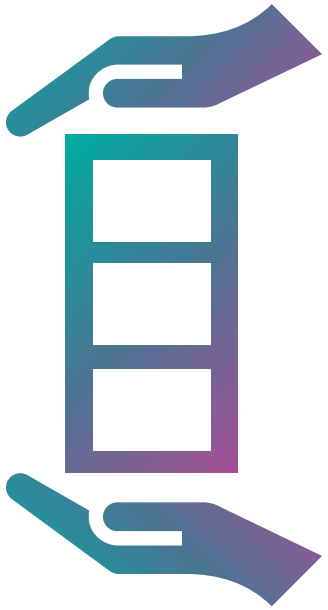


Aggregating Grouped Dataframes

Aggregating data down to a single value per group combination

(e.g., mean values for demographic groups)

Aggregating a vector



`summary()` – exploratory aggregation

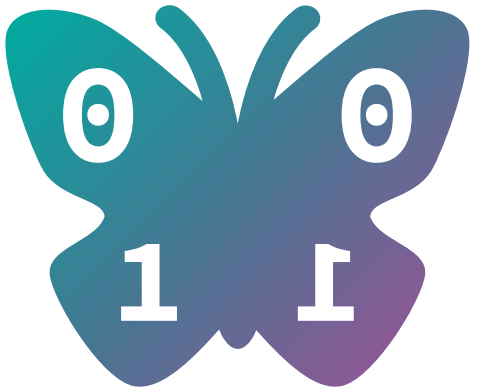
Vector	Code	Output
Numerical	`summary(` `  c(1, 2, 3, 4, NA, NA))`	`Min. 1st Qu. Median Mean 3rd Qu. Max. NA's` `1.00 1.75 2.50 2.50 3.25 4.00 2`
Logical	`summary(` `  c(TRUE, FALSE, FALSE, FALSE))`	`Mode FALSE TRUE` `logical 3 1`
Character	`summary(` `  c("f", "m", "f", "d", "m"))`	`Length Class Mode` `5 character character`
Factor	`summary(` `  factor(c("f", "m", "f", "d", "m")))`	`d f m` `1 2 2`

Tipp: For vectors in data frames use the $ selector: **`summary(df$some_variable)`**

BONUS ☺

Polymorphism

- `summary()` behaves differently for different input classes (types)
- e.g., summary() can also process result objects from analyses, such as regression results from lm() or glm() or ANOVA results from aov()
- This feature is called **polymorphism**:
  A common interface but class specific implementation “methods”.
- Packages can add new behaviors to such **generic functions** by writing their own **methods** (i.e., functions for a class) to them.
- Other generic function **examples**:
  - `print()` – Text output for different objects
  - `anova()` – Model comparison for regressions, confirmatory factor analyses etc.



Mathematical Aggregation Functions

- Vectors can be aggregated to a single value via many mathematical functions
- We already saw many of them while discussing mutate()

`mean` `sd` `sum` `min` `max` **(!)** All return NA if any value is NA unless `(x, na.rm=TRUE)`

- Additionally:

`median` `quantile(x, probs = c(0.25, .5, 0.75))` outputs a vector of quantiles

- And even more via Packages; e.g., psych::

`psych::geometric.mean()` `psych::harmonic.mean()`

Grouping data

dplyr::group_by()

- **group_by()** groups the data frame by the content of one or more of its variables

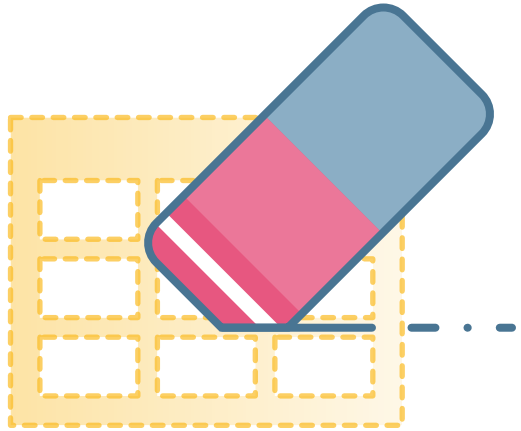
```
df %>% group_by(edu)
```

- After grouping, nothing has visibly changed exept for grouping information when we print the data frame
- However, grouping **changes how the follwoing functions work!**
  - **arrange()** can sort within groups

```
df %>% group_by(edu) %>% arrange(age, .by_groups = TRUE)
```

  - **mutate()** applies aggregation functions by group (e.g., group means)
  - **summarise()** can condense groups down to aggregated values

ungroup()



ungroup() removes **grouping**

group_by() – basics

- **Grouping** can be done with different **types of vectors** (numeric, logical, string, factor …)
- **Unique values** become groups
- Grouping **does not sort** the rows!

```
df %>% group_by(college)
```

	college	class	age
1	TRUE	middle class	33
2	FALSE	lower class	26
3	FALSE	lower class	49
4	TRUE	middle class	80
5	TRUE	upper class	75

grouping information:

groups — where in df

college	.rows
FALSE	2, 3
TRUE	1, 4, 5

`group_by()` – multiple grouping variables

- **Several variables** can be used for groups!
- This creates all combinations of unique values

	college	class	age
1	TRUE	middle class	33
2	FALSE	lower class	26
3	FALSE	lower class	49
4	TRUE	middle class	80
5	TRUE	upper class	75
	…	…	…

college	class	.rows
FALSE	lower	…
FALSE	middle	…
FALSE	upper	…
TRUE	lower	…
TRUE	middle	…
TRUE	upper	…

Aggregating grouped data

`summarise()` – aggregate (grouped) data

- **`dplyr::summarise()`** aggregates a data frame down to one row per group combination.
- To do so, it applies one or more **vector aggregation functions**.
- The **result** is a dataframe with the group structure and one or several aggregation result variables.
- **Aggregation variables** contain exactly one value per group combination

`summarise()` – basics

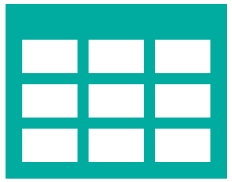
- **`dplyr::summarise()`** aggregates a data frame down to one row per group combination.

`group_by()` → **`summarise()`**

```
df %>%
group_by(college_degree)
```

%>%

```
summarise(
mean_age = mean(age, na.rm = TRUE),
sd_age = sd(age, na.rm = TRUE)
)
```



college
FALSE
TRUE

college	mean_age	sd_age
FALSE	37.5	16.3
TRUE	62.7	25.8

Grouping determines **rows** of summarised data frame!

summarise() – **multiple** grouping variables

group_by() → **summarise()**

```
df %>%
group_by(college_degree,
         class)
```

%>%

```
summarise(
mean_age = mean(age, na.rm = TRUE),
sd_age = sd(age, na.rm = TRUE)
)
```

college	class
FALSE	lower
FALSE	middle
FALSE	upper
TRUE	lower
TRUE	middle
TRUE	upper

college	class	mean_age	sd_age
FALSE	lower	32.3	9.2
FALSE	middle	43.3	15.4
FALSE	upper	47.8	18.3
TRUE	lower	38.7	2.1
TRUE	middle	42.2	12
TRUE	upper	42.8	9.2

`summarise()` – **without** grouping

`df()` → **`summarise()`**

```
df %>%
summarise(
mean_age = mean(age, na.rm = TRUE),
sd_age = sd(age, na.rm = TRUE)
)
```

mean_age	sd_age
52.6	24.3

Note
Without grouping, everything is reduced to a single row

Helpeful functions within `summarise()`

BONUS ☺

`n()` returns the **current group size**

- With `n()` we can add the **number of rows** that were condensed down into a single row

`n_distinct(var)` counts the unique values of var in a group

`first()` returns the first value of a variable in each group

- If a variable does not vary in a group, `first()` preserves the variable in the `summarise()` result.

`any()` and **`all()`** condense logical vectors into a single logical value.

- `any()` is TRUE if at least one is TRUE, `all()` is TRUE if all are TRUE

Example: Calculating **relative frequencies**

calculate absolute frequencies: n

```
gss %>%
  group_by(college, class) %>%
  summarise(n = n()) %>%
  ungroup() %>%
  mutate(rel_freq = n / sum(n),
         percentage = rel_freq * 100,
         across(where(is.numeric), round, 3))
```

college	class	n
no degree	lower class	17
no degree	working class	196
no degree	middle class	109
no degree	upper class	4
degree	lower class	3
⋮	⋮	⋮

Example: Calculating **relative frequencies**

`ungroup()`
We want to remove grouping, because otherwise the grouping would interfere with the `sum()` in the `mutate()` below.

```
gss %>%
  group_by(college, class) %>%
  summarise(n = n()) %>%
  ungroup() %>%
  mutate(rel_freq = n / sum(n),
         percentage = rel_freq * 100,
         across(where(is.numeric), round, 3))
```

college	class	n
no degree	lower class	17
no degree	working class	196
no degree	middle class	109
no degree	upper class	4
degree	lower class	3
⋮	⋮	⋮

Example: Calculating **relative frequencies**

Calculate **relative frequency**: Take the absolute frequency n and divide it by the sum of all absolute frequencies (i.e., all cases).

```
gss %>%
  group_by(college, class) %>%
  summarise(n = n()) %>%
  ungroup() %>%
  mutate(rel_freq = n / sum(n),
         percentage = rel_freq * 100,
         across(where(is.numeric), round, 3))
```

college	class	n	rel_freq
no degree	lower class	17	0.034
no degree	working class	196	0.392
no degree	middle class	109	0.218
no degree	upper class	4	0.008
degree	lower class	3	0.006
⋮	⋮	⋮	⋮

Example: Calculating **relative frequencies**

calculate percentages

…and round all numeric columns with `across()` magic ☺

```
gss %>%
  group_by(college, class) %>%
  summarise(n = n()) %>%
  ungroup() %>%
  mutate(rel_freq = n / sum(n),
         percentage = rel_freq * 100,
         across(where(is.numeric), round, 3))
```

college	class	n	rel_freq	percentage
no degree	lower class	17	0.034	3.4
no degree	working class	196	0.392	39.2
no degree	middle class	109	0.218	21.8
no degree	upper class	4	0.008	0.8
degree	lower class	3	0.006	0.6
⋮	⋮	⋮	⋮	⋮

BONUS ☺

Example2: **Quantiles**

Note: quantiles() takes a vector of relative frequencies and returns a vector of quantiles

calculate quantiles →

pivot_wider() to have quantiles in columns →

```
iris %>%
  group_by(Species) %>%
  summarise(p = c(.05, .25, .5, .75, .95),
   quantile = quantile(Petal.Length, p)
  ) %>%
 pivot_wider(names_from = p,
             values_from = quantile,
             names_prefix = "p_")
```

Species	p_0.05	p_0.25	p_0.5	p_0.75	p_0.95
setosa	1.2	1.4	1.5	1.575	1.7
versicolor	3.39	4	4.35	4.6	4.9
virginica	4.845	5.1	5.55	5.875	6.655

Aggregation by **nesting**

Cells in data frames are not restricted to single values. Instead, each cell can contain a spearate vector or even a whole data frame.

This is called „nesting“ in computer science.

```
df %>%
  group_by(fct) %>%
  summarise(
    mean_a = mean(a),
    backup_a = list(a)
  )
```

saves the group segment of x as a nested vector for later use

fct	mean_a	mean_backup	n
A	-0.041	<dbl [5]>	5
B	0.283	<dbl [7]>	7

BONUS ☺

Nesting use case: **Regression by Group**

Regression for each group →

Unfolds the results (which are nested data frames) into columns →

```
iris %>%
  group_by(Species) %>%
  summarise(
    reg = lm(Petal.Length ~ Sepal.Width) %>% glance()
  ) %>%
  unnest(reg)
```

Model summary via `broom::glance()`

Species	adj.r.squared	p.value	statistic	nobs
setosa	0.011	0.217	1.565	50
versicolor	0.300	0.00	21.990	50
virginica	0.143	0.004	9.200	50

Nesting use case: **Regression by Group**

BONUS ☺

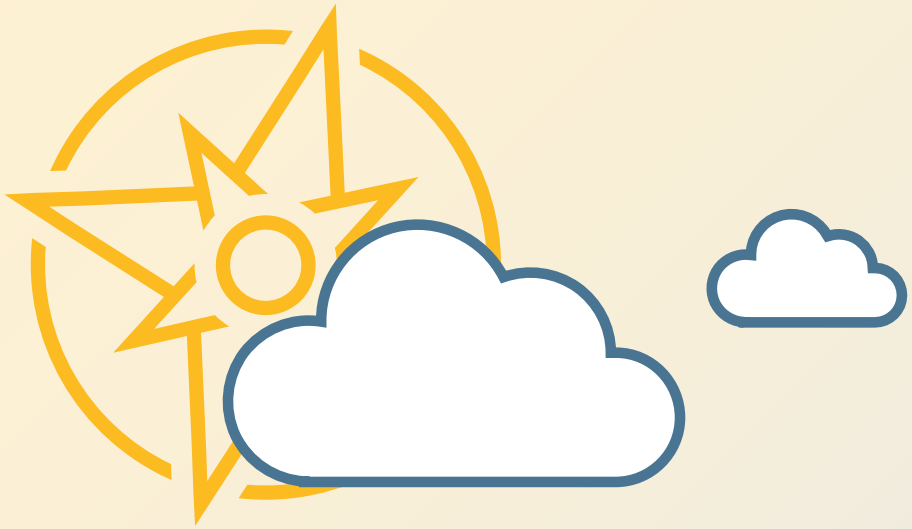
Regression for each group →

Unfolds the results (which are nested data frames) into columns →

```
iris %>%
  group_by(Species) %>%
  summarise(
    reg = lm(Petal.Length ~ Sepal.Width) %>% tidy()
  ) %>%
  unnest(reg)
```

Coefficients via `broom::tidy()`

Species	term	estimate	std.error	statistic	p.value
setosa	(Intercept)	1.183	0.224	5.271	0
setosa	Sepal.Width	0.081	0.065	1.251	0.217
versicolor	(Intercept)	1.935	0.499	3.878	0
versicolor	Sepal.Width	0.839	0.179	4.689	0
virginica	(Intercept)	3.511	0.677	5.187	0
virginica	Sepal.Width	0.686	0.226	3.033	0.004



Descriptive statistics and **visual exploration**

Descriptive Statistics (in contrast to inferential)

- `group_by()` and `summarise()` can help here!
- Additionally, we can use **simple bivariate exploratory analyses**
  - Correlation for metric variables

```
cor(df$var_1, df$var_2, use = "complete.obs")
```
Person's correlation

  - Correlation for non-normal or ordinal variables

```
cor(method = "kendall")
```
or method = "spearman"

  - Cohen's d via **`psych::cohen.d()`**

```
psych::cohen.d(df$var_1, df$var_2, na.rm = TRUE)
```
outputs a vector of quantiles

Example: Global and group correlation

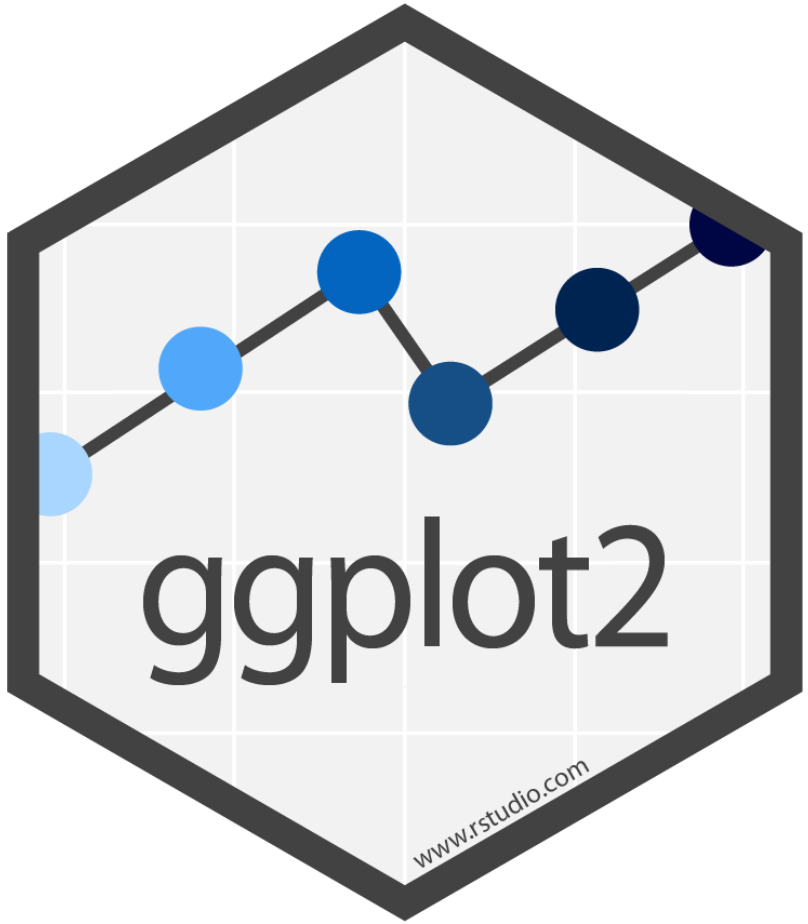
Correlation between sepal length and petal length:

Once overall and once by species!

```
iris %>%
  mutate(sepal_petal_cor = cor(Sepal.Length, Petal.Length)) %>%
  group_by(Species) %>%
  summarise(sepal_petal_cor = first(sepal_petal_cor),
            group_cor = cor(Sepal.Length, Petal.Length))
```

Species	sepal_petal_cor	group_cor
setosa	.872	.267
versicolor	.872	.754
virginica	.872	.864

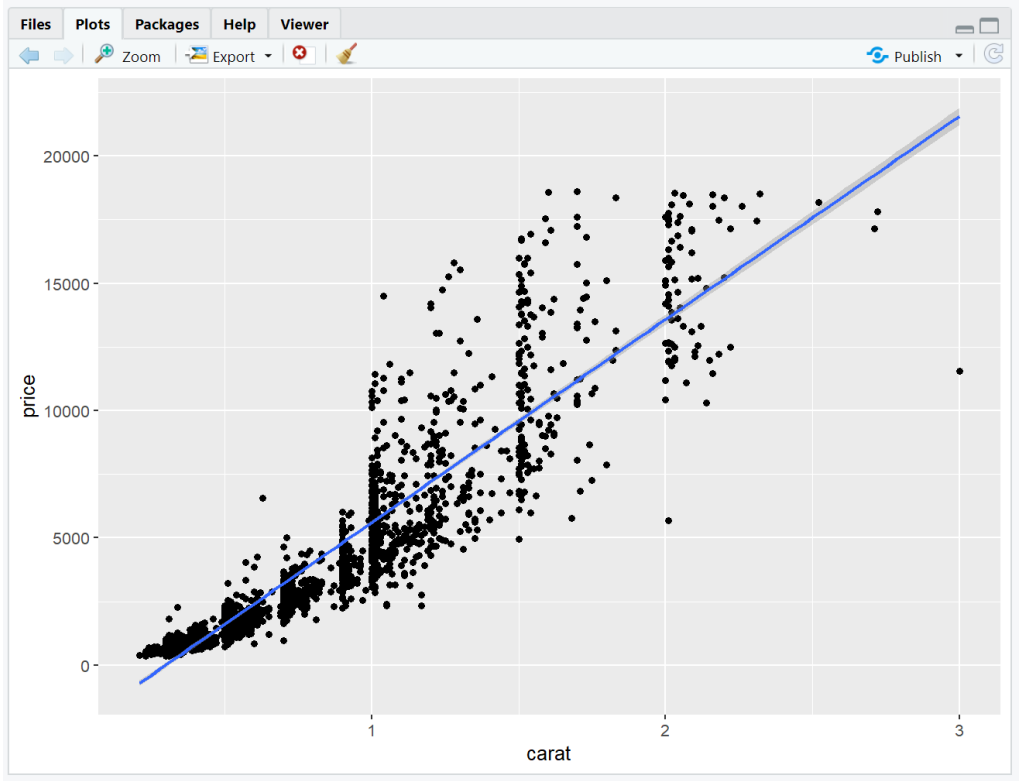
Visual exploration with **`ggplot2::`**



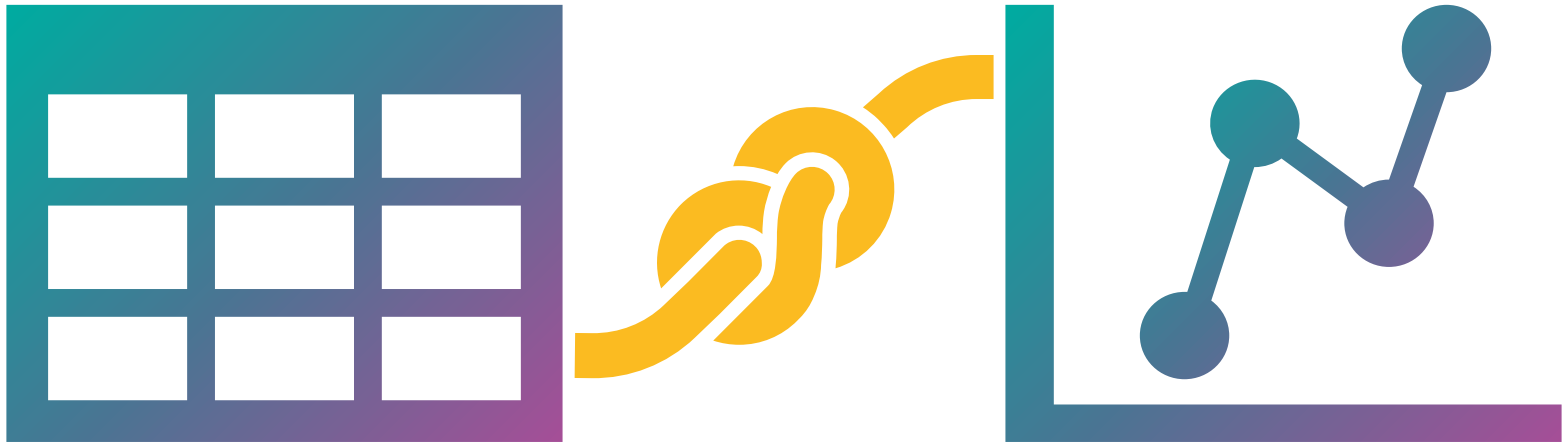
ggplot2:: – Basics

- ggplot2:: is a a powerful and flexible engine for generating plots and visualizations

```
diamonds %>%
  sample_n(2000) %>%
  ggplot(aes(carat, price))+
  geom_point()+
  geom_smooth(method = "lm")
```



`ggplot()` – anatomy

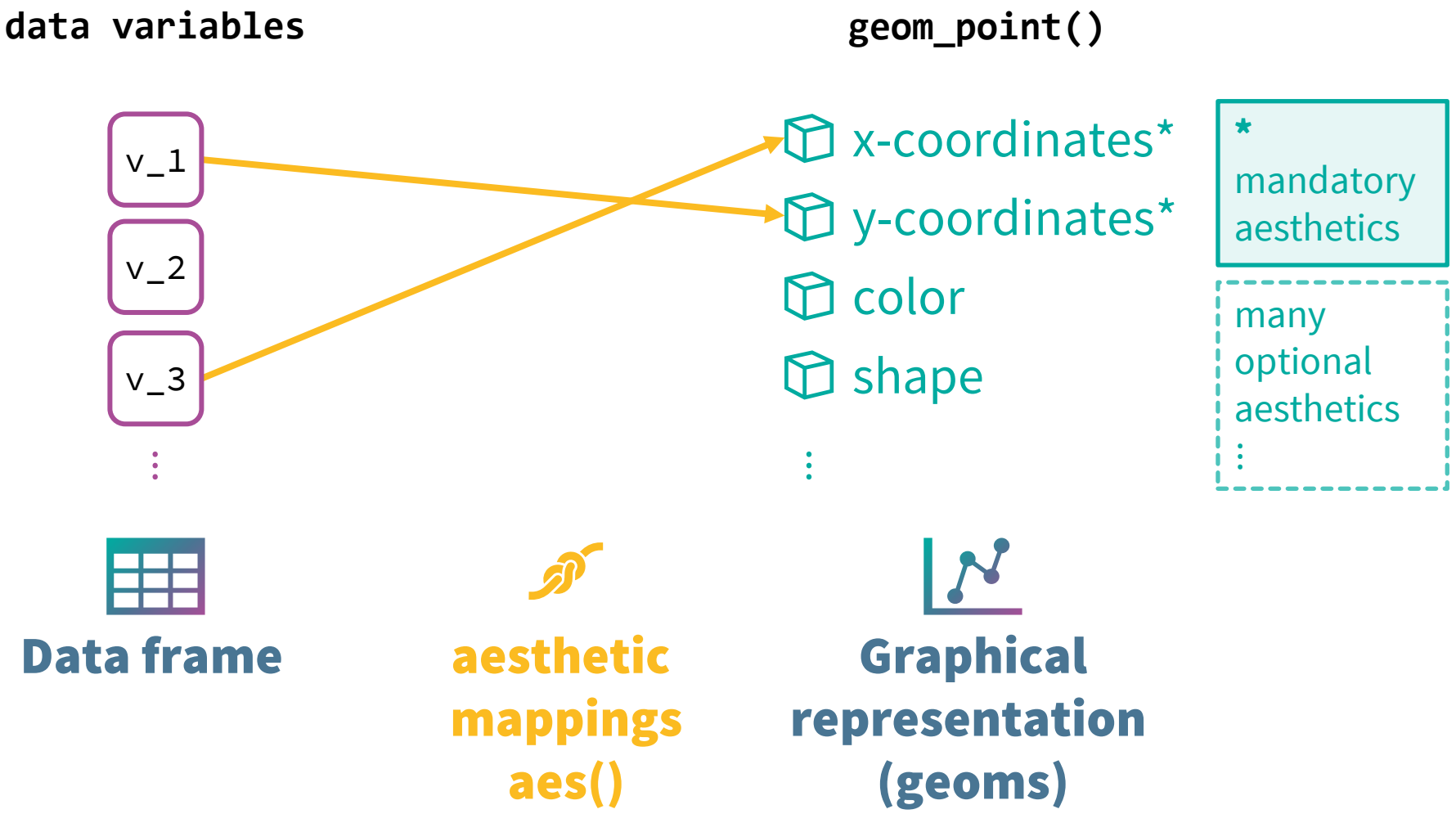


Data frame(s)

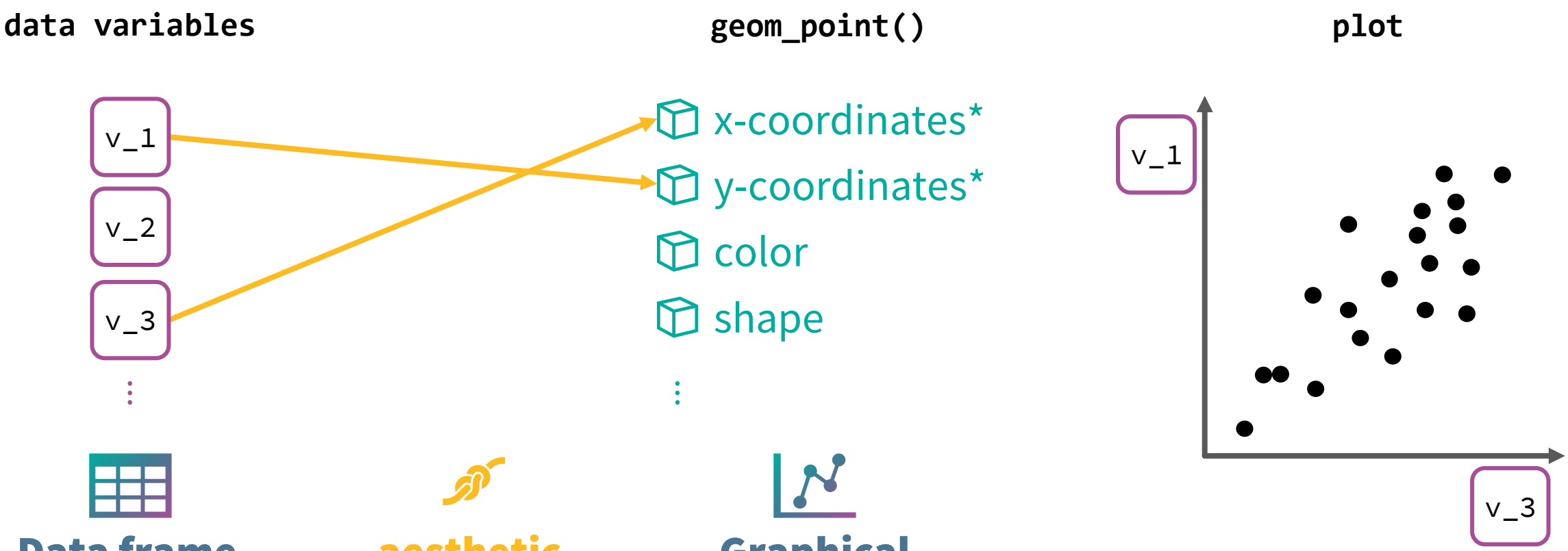
aesthetic mappings aes()

Graphical representation (geoms)

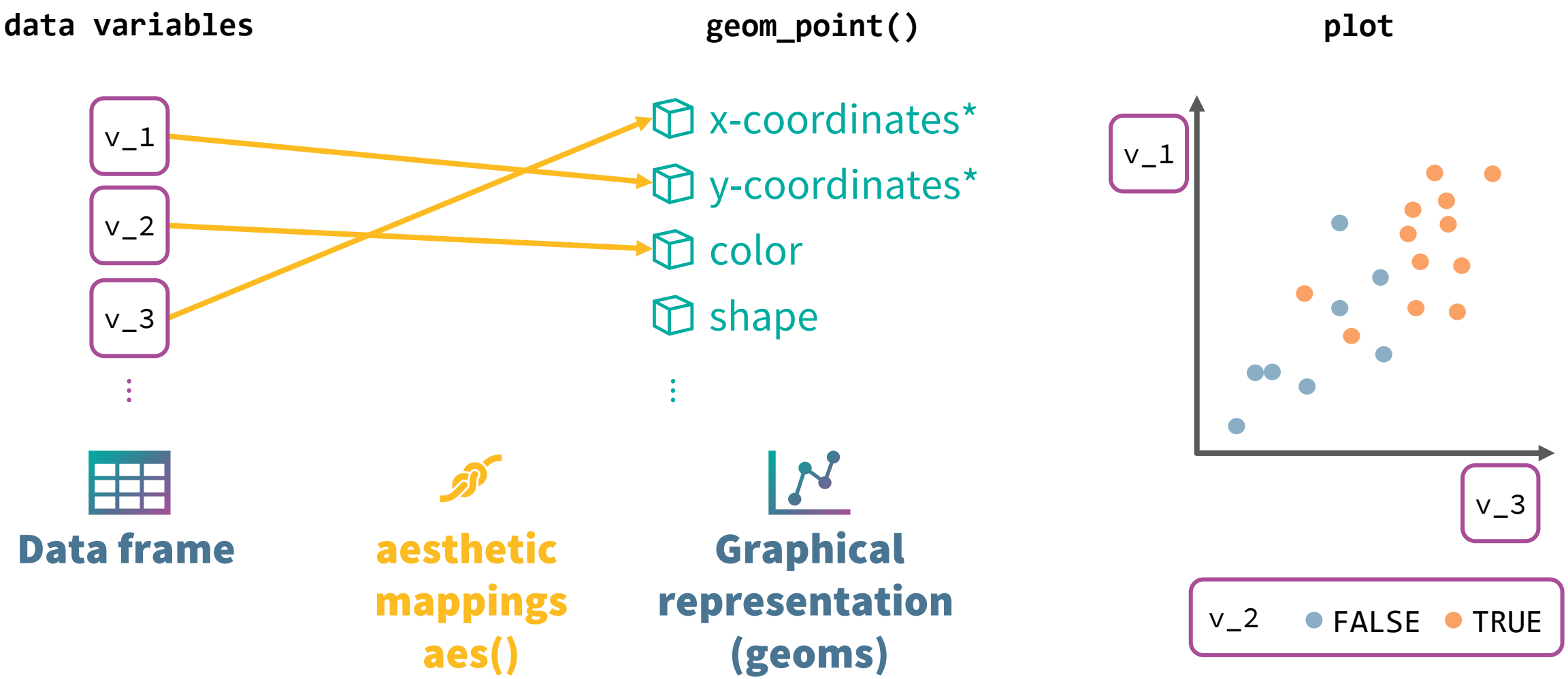
`ggplot()` – anatomy



`ggplot()` – anatomy



`ggplot()` – anatomy

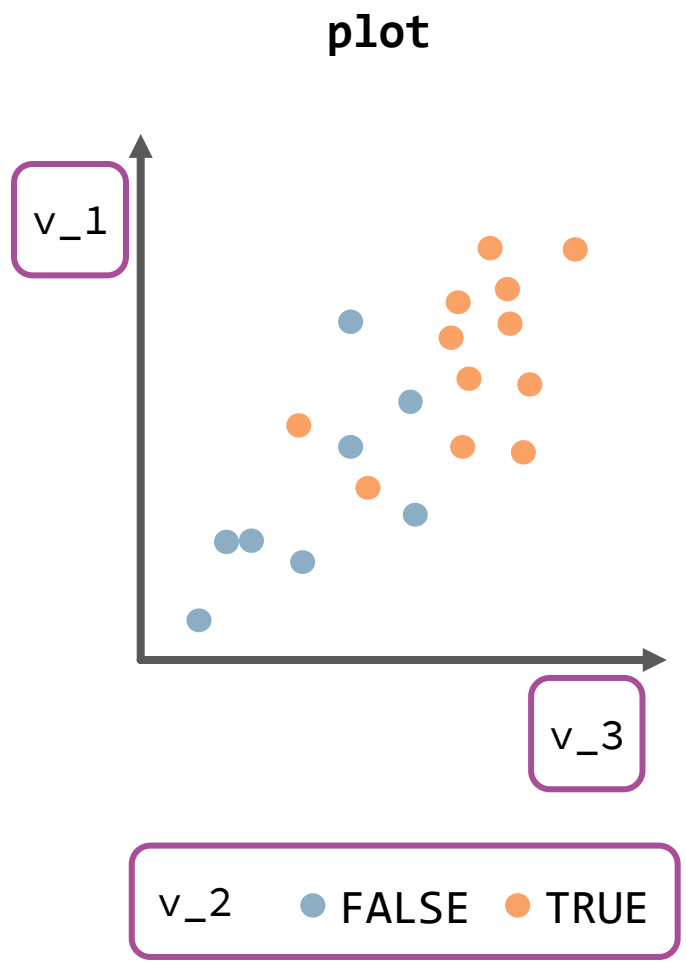


`ggplot()` – anatomy

x-coord.
y-coord.
color

```
df %>%
  ggplot(aes(
    x = v_3,
    y = v_1,
    color = v_2))+
  geom_point()
```

v_3
v_1
v_2



`ggplot()` – anatomy

```
df %>%
  ggplot(aes(
    x = v_3,
    y = v_1,
    color = v_2))+
  geom_point()+
  geom_smooth(
      aes(color = var_a),
      data = other_df))
```

geoms can be **layered**
first geom gets drawn first, then second on top etc.

geoms inherit data and aes() from ggplot()

But **new aes() mappings** or **new data sources** can be provided as well.

+

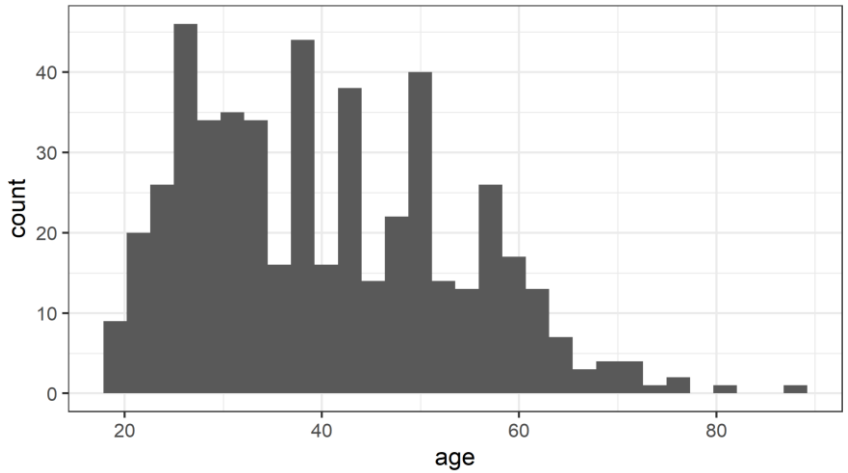
ggplot functions are linked by **plus signs**!

(Take care not to use %>%)

Univariate: **Histogram**

Numerical Variable

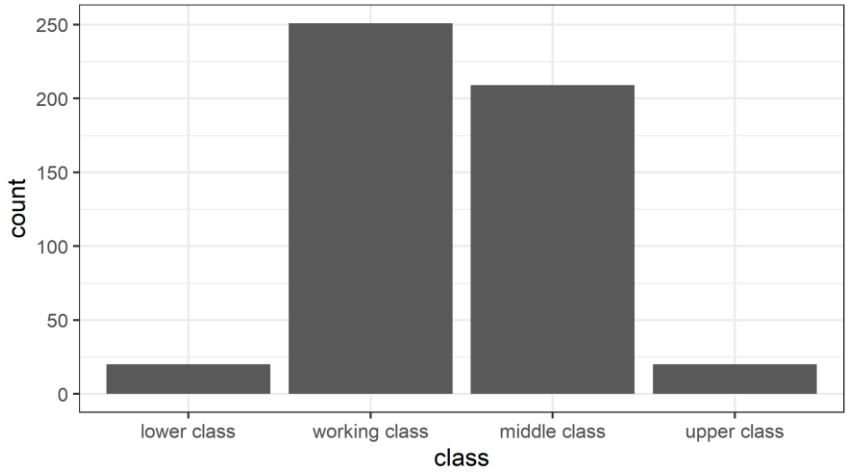
```
df %>%
  ggplot(aes(numerical_var))+
  geom_histogram()
```



granularity can be finetuned with the **bin** (number of bars) or **binwidth** arguments

Factor Variables

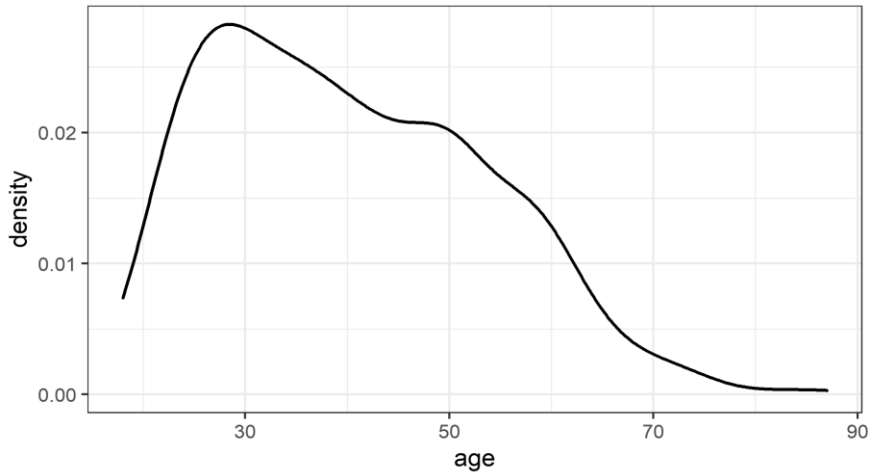
```
df %>%
  ggplot(aes(factor_var))+
  geom_histogram(stat = "count")
```



Univariate: **Density plot**

Numerical Variable

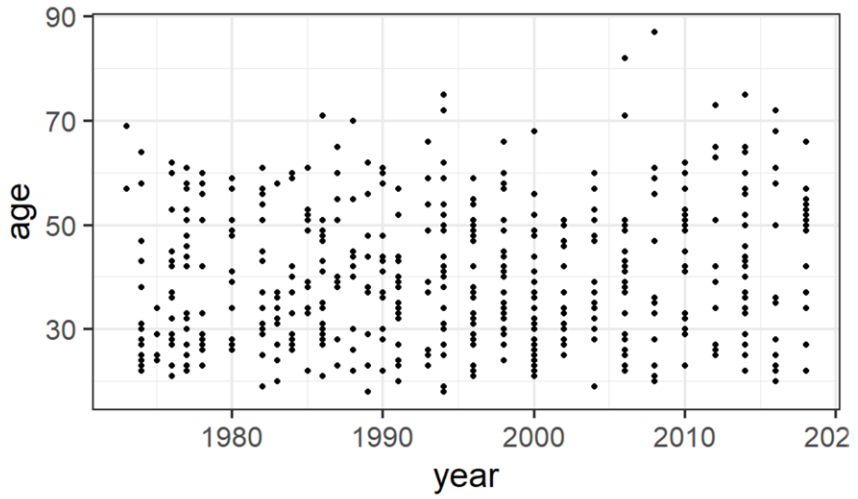
```
df %>%
  ggplot(aes(numerical_var))+
  geom_density()
```



Bivariate: **Scatterplot `geom_point()`**

Two Numerical Variables

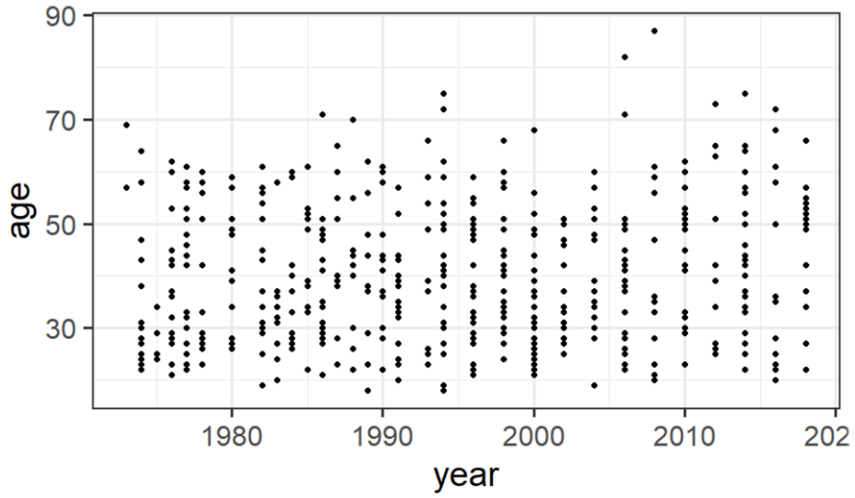
```
df %>%
  ggplot(aes(num_x, num_y))+
  geom_point()
```



Bivariate: **Scatterplot `geom_point()`**

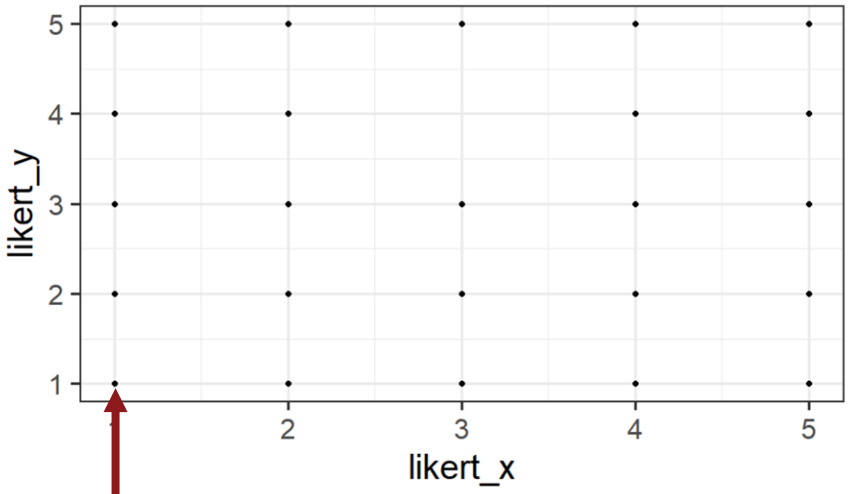
Two Numerical Variables

```
df %>%
  ggplot(aes(num_x, num_y))+
  geom_point()
```



Pseudo-metric (e.g., Likert items)

```
df %>%
  ggplot(aes(likert_x, likert_y))+
  geom_point()
```

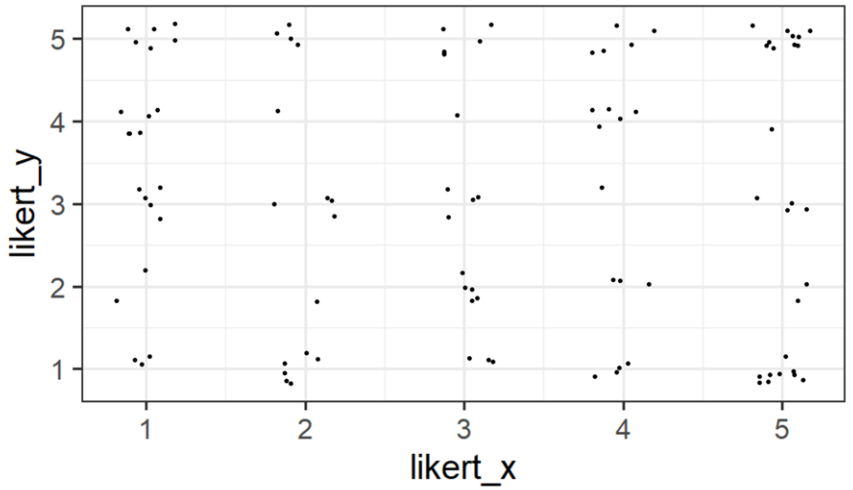


Each point might represent arbitrary many cases!

Bivariate: **Pseudo-metric** scatterplots

geom_jitter()

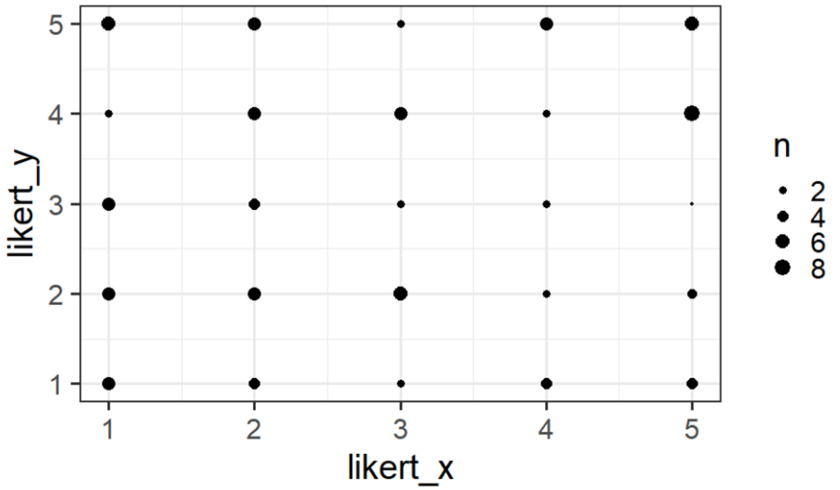
```
df %>%
  ggplot(aes(likert_x, likert_y))+
  geom_jitter(width=0.2, height=0.2)
```



„jitters“ points randomly
around their actual x and y

geom_count()

```
df %>%
  ggplot(aes(likert_x, likert_y))+
  geom_count()
```



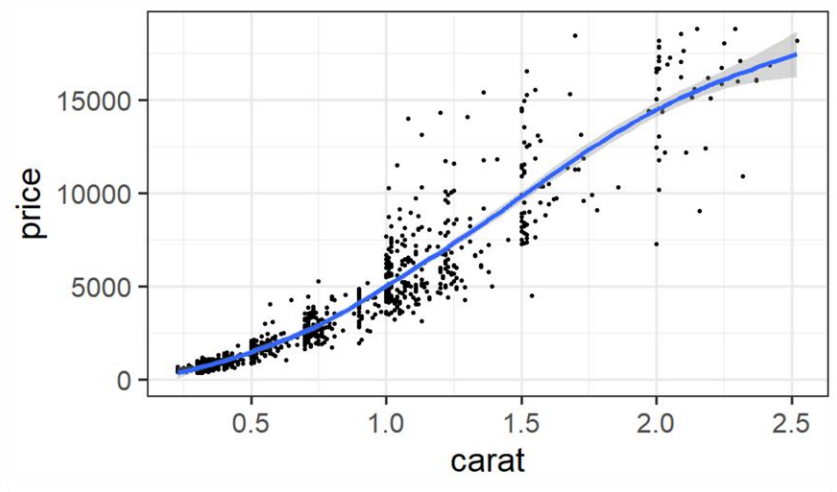
point radius represents
case count at each coordinate

Bivariate: **Numeric trends**

Tipp: Use `geom_point()` and then `geom_smooth()` to combine raw and modeled data in one plot

Smooth trend curve (LOESS or GAM)

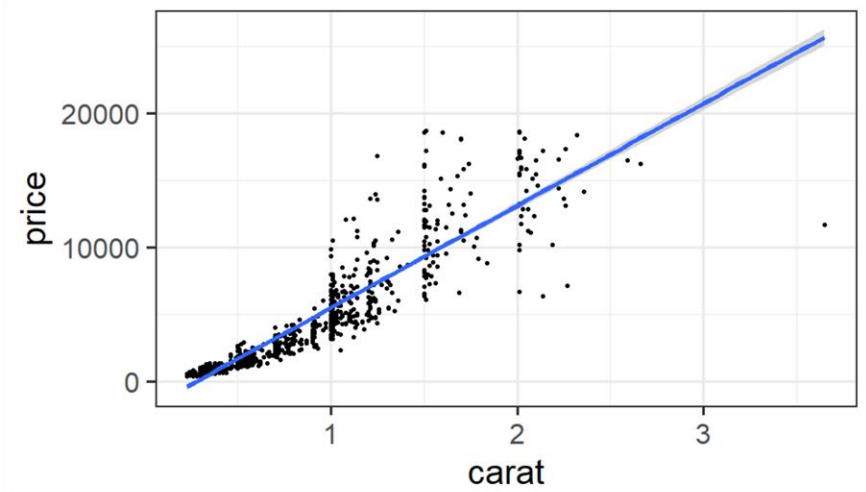
```
df %>%
  ggplot(aes(num_x, num_y))+
  geom_point()+
  geom_smooth()
```



n < 1000: LOESS - Local Polynomial Regression Fitting
n > 1000: GAM – Generalized additive model

Linear trend curve (lm)

```
df %>%
  ggplot(aes(num_x, num_y))+
  geom_point()+
  geom_smooth(method = "lm")
```

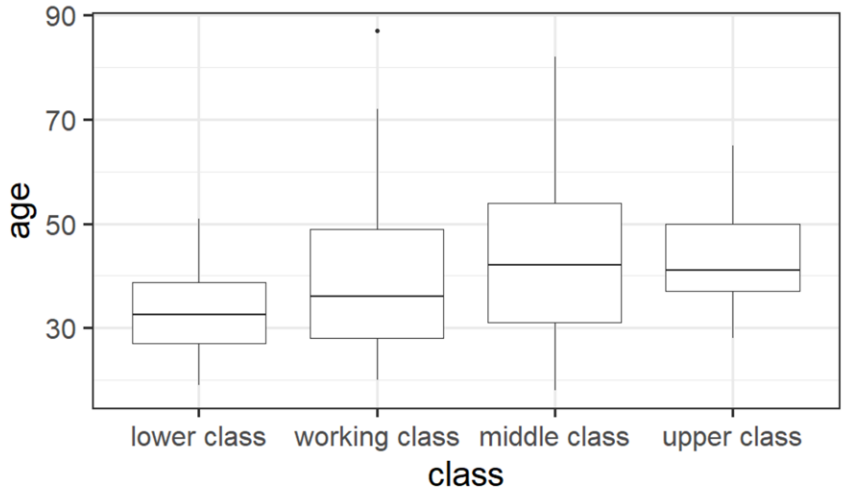


A mix of `geom_boxplot()` and `geom_density()`

Bivariate: **Factor & Numeric**

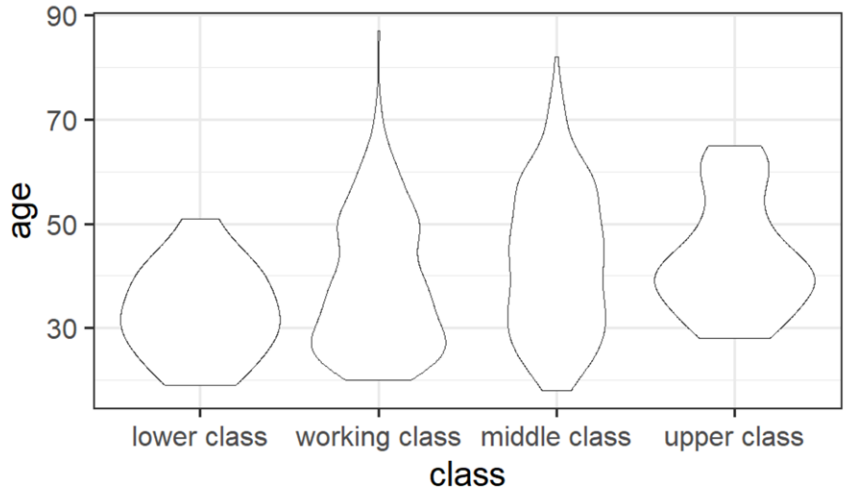
geom_boxplot()

```
df %>%
  ggplot(aes(factor_x, num_y))+
  geom_boxplot()
```

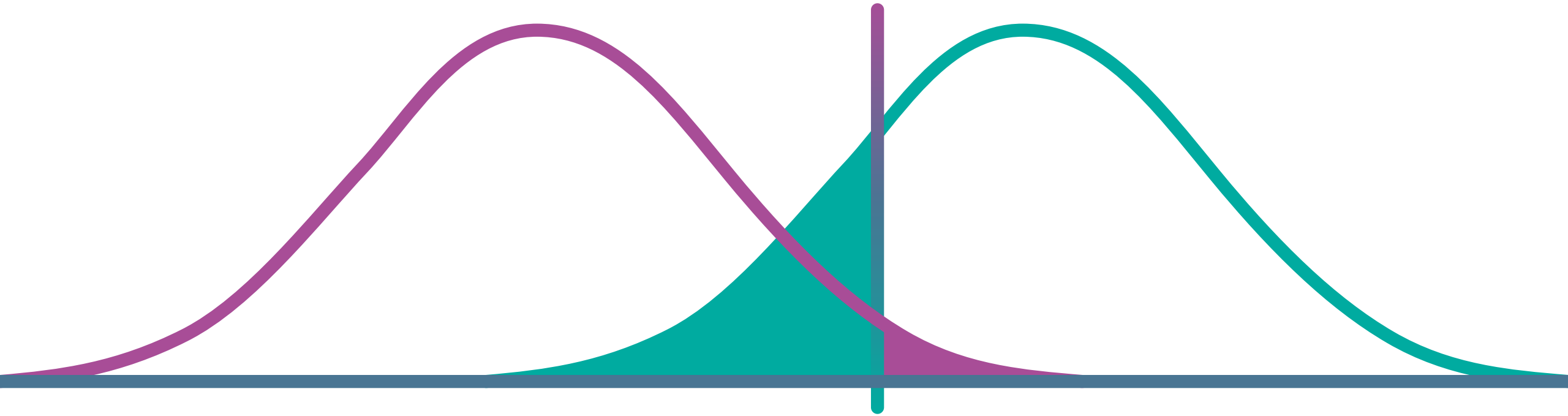


geom_violin()

```
df %>%
  ggplot(aes(factor_x, num_y))+
  geom_violin()
```



A mix of `geom_boxplot()` and `geom_density()`



Formal Data Analysis

Analysis – basics

data **model specification** **analysis method**

Analysis – components

BONUS ☺

Data

Data are supplied via:

- **Complete data frame** (variables are selected via model specification)
- **Pruned data frame** with analysis variables only
- **Separate vector(s)**

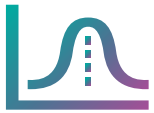
Model Specification

Models link data elements with their roles in the analyses (such as predictor or dependent variable)

Models are specified via:

- **R formula syntax**:
  `y ~ x1 + x2 + x3`
- Specific function **arguments**
- **Implied** by data structure

Analysis Method



The **method** includes both the general analysis type as well as analysis configuration.

Method are specified via:

- The choice of **R function**
  e.g., `cor.test`, `t.test`
- Via specific argument
- Implied by the supplied data

Analysis – **Fit objects** (i.e., analyses results)

- Performing an analyses in R prints a short result summary.
- **This is misleading!**

- Most analyses in R generate a comprehensive **fit object**, with detailled information.
- Hence, we **assign the result** of the analysis function **to a variable**.
- Then we can **pluck** what **information** we need from the saved fit object.

Correlation: `cor.test()`

`cor.test()` works like `cor()` but also performs Null-Hypothesis Significance Testing (NHST).

`cor.test(df$var_a, df$var_b)` choice of method = c("**pearson**", "kendall", "spearman")

Result if function is just called:

```
        Pearson's product-moment correlation

data:  iris$Petal.Length and iris$Petal.Width
t = 43.387, df = 148, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9490525 0.9729853
sample estimates:
      cor
0.9628654
```

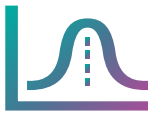
Correlation: `cor.test()`

Data

- Data is supplied as **two separate vectors**.
- Note that we have to specify the data frame: `some_df$some_var`

Model Specification

- The two supplied vectors **implicitly specify** their respective roles in the analyses.



Analysis Method

- The method family is speficied via the function `cor.test()`
- However, if we want Spearman or Kendall instead of the Person default, we have to specify that via the argument `method`.

`cor.test()` – Fit object

- Save result to variable:

```
cor_fit <- cor.test(df$var_a, df$var_b)
```

- In **base R**, you can always try those three functions on fit objects:
  - **`print()` – works with cor.test()**
  - `summary()` – does not work
  - `plot()` – does not work

- The **`broom::` package** provides a tidy interface for many analyses.
  - For `cor.test()`, we can use `broom::glance()` or `broom::tidy()` for the same result

`cor.test()` example with the iris dataset

```
cor_result <- cor.test(iris$Petal.Length, iris$Petal.Width)
cor_result  %>% broom::glance()
```

A tibble: 1 x 8

estimate <dbl>	statistic <dbl>	p.value <dbl>	parameter <int>	conf.low <dbl>	conf.high <dbl>	method <chr>	alternative <chr>
0.9628654	43.38724	4.675004e-86	148	0.9490525	0.9729853	Pearson's product-moment correlation	two.sided

1-1 of 1 rows

If you prefer a horizontal table nicely formatted in R Markdown:

```
library(kableExtra)
cor_result  %>% glance() %>% t() %>% kable() %>% kable_styling()
```

`t_test()` – Independent sample t-test

- **Base R** has a clunky t.test() function, that we will ignore for now ☺
- Instead, we use `t_test()` from the **infer:: package** (part of tidymodels::)

```
library(infer)
df %>% t_test(dependent_numeric ~ independent_factor)
```

Note the formula syntax with ~

- The output is already formated as if we had used tidy() or glance()

statistic	-1.11931	t-statistic
t_df	365.6404	degrees of freedom
p_value	0.2637428	p value
alternative	two.sided	two-sided or one-sided t-test
estimate	-1.538432	mean difference (group 1 minus 2; by factor level order)
lower_ci	-4.241245	Lower confidence interval boundary (95% default)
upper_ci	1.164381	Upper confidence interval boundary (95% default)

`infer::chisq_test()` – χ^2-test for independence

- Easy test for independence of two nominal variables:

```
library(infer)
df %>% chisq_test(nominal_a ~ nominal_b)
```

"statistic"	"chisq_df"	"p_value"
30.682523	5	1.1e-05

`lm()` – Linear OLS Regression

- lm() basic call:

```
lm(y ~ x, data = df)
```

formula: `y ~ x` — data: `data = df`

lm() call with a pipe: Use the dot stand-in for the piped data!

(Helpful, to change data filters on the fly in exploratory analyses.)

```
df %>% lm(y ~ x, data = .)
```

And usually, we want to **assign** the regression result to a variable:

```
lm_result <- lm(y ~ x, data = df)
```

`lm()` – Formulas

We specify the model for our regression with R formula syntax

Univariate Regression

```
dependent_var ~ predictor
```

Multivariate Regression (additive model)

```
dependent_var ~ pred_1 + pred_2 + pred_3
```

`lm()` – Interactions

	All possible interactions	**Only specific interactions**
	`a * b * c`	`a + b + c + a:b`
	Terms generated by lm():	**Terms generated by lm():**
additive terms	a b c	a b c
Two-way interaction terms (first-order interaction)	a:b a:c b:c	a:b
Three-way interaction term (higher-order interactions)	a:b:c	

BONUS ☺

`lm()` – Factor predictors

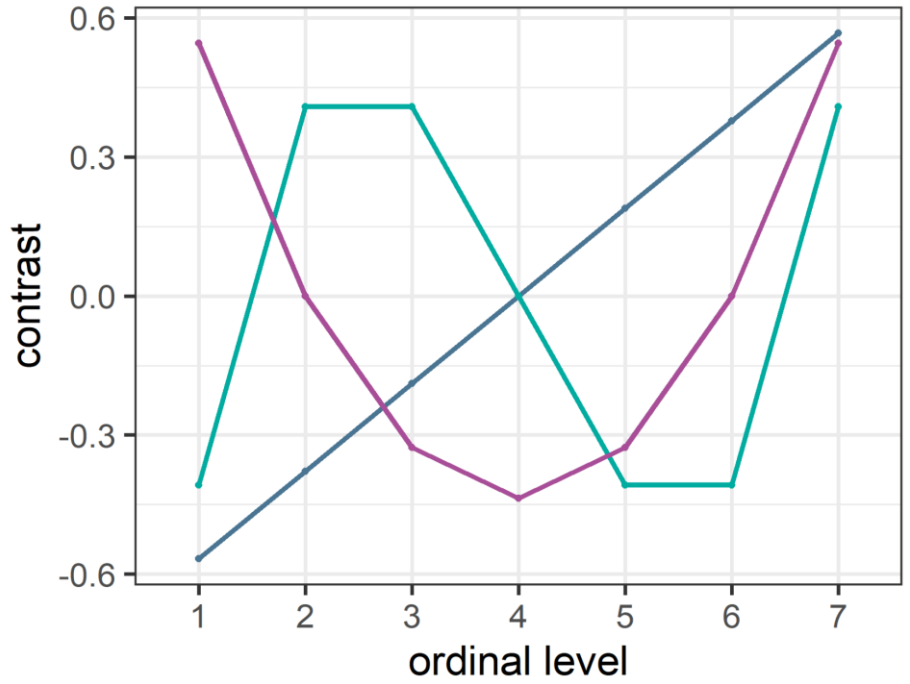
- lm() automatically recognizes factor predictors as categorical
- **Factor predictors** are entered into the model via **treatment coding** (i.e., indicator coding; colloquially: dummy coding)
  - The first level of the factor becomes the **reference category** and is **omitted** from the model
  - All other levels are entered as treatment dummies, meaning their effects are interpretable in contrast to the reference category.

Factor predictor	**Dummy for second f.-level**	**Dummy for third f.level**
`Employment` (var name)	`Employmentself-employed`	`Employmentunemployed`
`employee`	`0`	`0`
`self-employed`	`1`	`0`
`unemployed`	`0`	`1`

BONUS ☺

lm() – Ordered Factor predictors

- Ordered Factors behave differently.
  (i.e., factor(ordered = TRUE)
- lm() estimates **polynomial contrasts** for **ordinal variables**
  (i.e., **linear**, **quadratic**, **cubic** …)
- The plot shows the contrast coding for an ordinal variable with seven levels



Tipp: If this default behavior is undesired, you can:
- Transform the variable to numeric for the linear effect
- Transform the variable to a normal factor for dummy coding
- Manually choose a contrast algorithm via lm(contrasts = …)

`lm()` results: `glance()` model overview

```
lm_result <- lm(y ~ x, data = df)
glance(lm_result)
```

`r.squared`	R^2 - coefficient of determination
`adj.r.squared`	R^2 adjusted for degrees of freedom
`sigma`	Estimated standard error of residuals
`statistic`	F statistic
`p.value`	p-value
`df`	Degrees of freedom
`logLik`	Log-Likelihood of the model
`AIC`	Akaike's Information Criterion
`BIC`	Bayesian Information Criterion
`Deviance`	Deviance of the model
`df.residual`	Residual degrees of freedom
`nobs`	Number of observations used

Tipp: For a horizontal, well formatted output in R Markdown, use:

```
library(kableExtra)
lm_result  %>% glance() %>% t() %>% kable() %>% kable_styling()
```

`lm()` results: `tidy()` model coefficients

```
lm_result <- lm(y ~ a:b, data = df)
tidy(lm_result)
```

term	estimate	std.error	statistic	p.value
(Intercept)				
a				
b				
a:b				

estimated values (unstandardized B)

standard errors

t-statistics

two-sided p-value

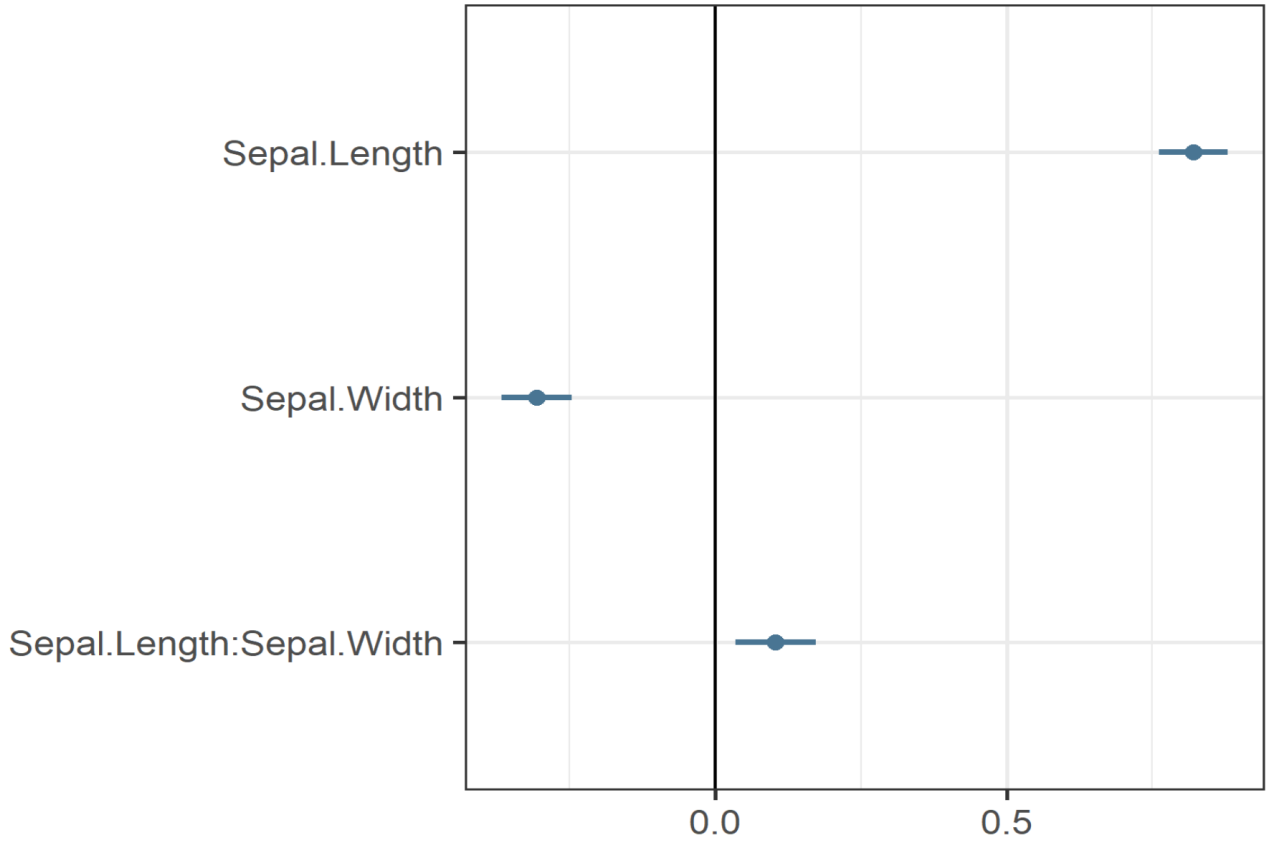
Tipp: For standardized estimates, mutate numeric predictors to z-scores with scale() before running lm().

```
df %>% mutate(across(where(is.numeric), scale)) %>%
  lm(y ~ a + b + c, .)
```

automatically scale()s all numeric variables in df

`dotwhisker::dwplot()` – Plotting results

```
lm_result %>% dotwhisker::dwplot(
  vline = geom_vline(aes(xintercept = 0)))
```



Dot-and-whisker plots visualize the model coefficients of a regression.

They show the **coefficient values** (dot) and **confidence intervalls** around them (whiskers).

The **vline** … argument adds a vertical line; here at zero.

`lm()` results: augment() for fitted values

```
augment(lm_result)
```

augment() applied to a lm() result creates a dataframe with:

- All empirical data we put into the model (dv and predictors)
- The predicted values for each respondent (`.fitted`)
- And many other helpful analysis derived values:

`.lower`
Lower bound on interval for fitted values.

`.se.fit`
Standard errors of fitted values.

`.resid`
The difference between observed and fitted values.

`.std.resid`
Standardised residuals.

Tipp: You can also easily add the agument() variables to your original data on which the lm() was based.

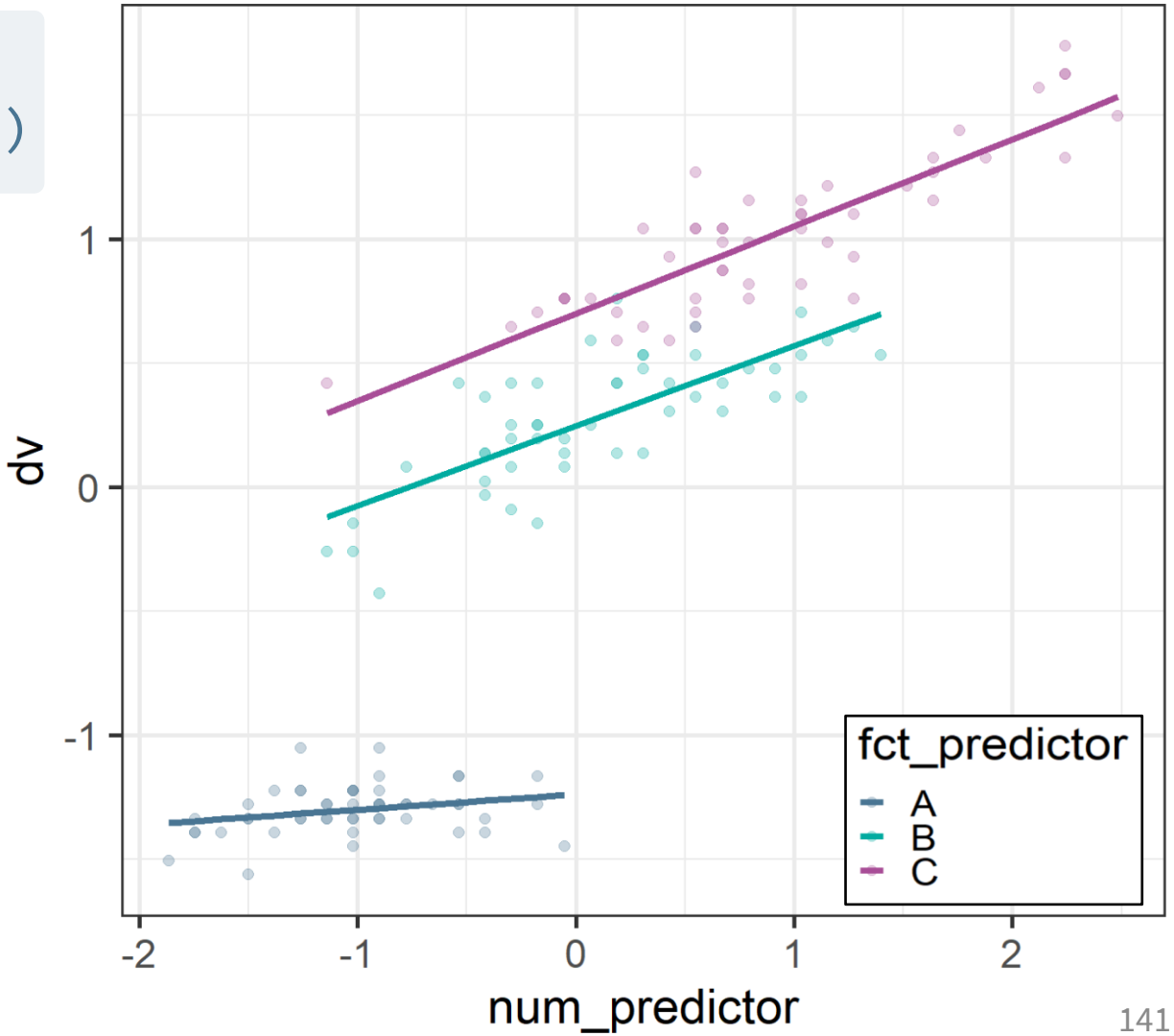
```
augment(lm_result, original_df)
```

`lm()` results: Plotting with augment()

```
lm_result <- df %>%
  lm(dv ~ num_predictor + fct_predictor, .)
```

```
lm_result %>% augment() %>%
  ggplot(aes(color = fct_predictor))+
  geom_point(aes(num_predictor, dv))+
  geom_line(aes(num_predictor, .fitted))
```

- Note the result split after the fct_predictor via **`aes(color)`**
- **`geom_point()`** for raw data
- **`geom_line()`** for the .fitted data



`lm()` Model comparison via **`anova()`**

BONUS ☺

Fitting three models
(Note: Terms are added but non removed)

```
iris_lm_1 <- lm(Petal.Length ~ Sepal.Length, iris)
iris_lm_2 <- lm(Petal.Length ~ Sepal.Length + Species, iris)
iris_lm_3 <- lm(Petal.Length ~ Sepal.Length * Species, iris)
```

Calculate explanatory power

```
glance(iris_lm_1)$adj.r.squared  0.7583327
glance(iris_lm_2)$adj.r.squared  0.9743786
glance(iris_lm_3)$adj.r.squared  0.9781215
```

Compare models with F-Test

```
anova(iris_lm_1, iris_lm_2, iris_lm_3) %>% tidy()
```

					F-statistic	NHST
	res.df	**rss**	**df**	**sumsq**	**statistic**	**p.value**
iris_lm_1	148	111.459	NA	NA	NA	NA
iris_lm_2	146	11.657	2	99.802	731.905	0
iris_lm_3	144	9.818	2	1.839	13.488	0

Presenting Results: **Text and Tables**

RMD output formats

- **R Markdown** kann be „knitted“ into different **output formats**
  - **HTML** – for easily navigatable and interactive documents
  - **PDF** – Knitting formatted documents via Latex
  - **Microsoft Word**

- However, R Markdown can also knit:
  - **Presentation** slides
  - Whole web-based **books** via Bookdown

RMD Basics – RMD File Components

RMD Document settings are defined in the **YAML-Header**

R code in **code chunks** is run and results added to the document

Markdown outside of chunks is transformed into formatted document text

````
---
title: "R Course Sketches"
output: html_document
---

```{r}
df %>% head()
```

# Markdown heading

Some markdown text.
````

- Output type (HTML etc.)
- Title, Author etc.
- Table of Contents etc.

- First chunk (setup, include = FALSE) can used for settings and package loading (Please load all packages in one place and not sprinkled through the RMD)

- See R Markdown Cheatsheet for a quick overview [link ↗]

RMD HTML: Dynamic Table of Contents

Display a **table of contents** that stays as you scroll through the document.

Clicking **navigates** to section.

Defined in **YAML Header:**

```
---
title: " "
output:
  html_document:
    toc: true
    toc_depth: 2
    toc_float:
      collapsed: false
---
```

◀ display a table of contents?

◀ Heading levels to display (e.g., 1 and 2)

◀ TOC now floats left of the document while scrolling

◀ false: All headings are show;
true: lower headings revealed on click or scroll

RMD Basics – Chunk output

Naming chunks adds them to the document outline in R Studio (Name must be unique!)

Chunk options govern chunk behavior (e.g., what is displayed in final document)

```
```{r chunk_name, include = FALSE, echo = FALSE}
# some R code
```
```

Tipp: Set **options for all chunks** in the setup chunk.
You can always override this setting for individual chunks!

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = FALSE)
```
```

Tipp: In Rstudio `[CTRL]+[ALT]+[i]` adds chunks; And within chunks, it splits them in two chunks!

`include = FALSE`
Code runs, but results are not displayed

`echo = FALSE`
R Syntax is ommited from document

`message = FALSE, warning = FALSE`
Messages and warnings of functions are not printed

`fig.width = 8, fig.height = 10`
Define plot size in knitted document

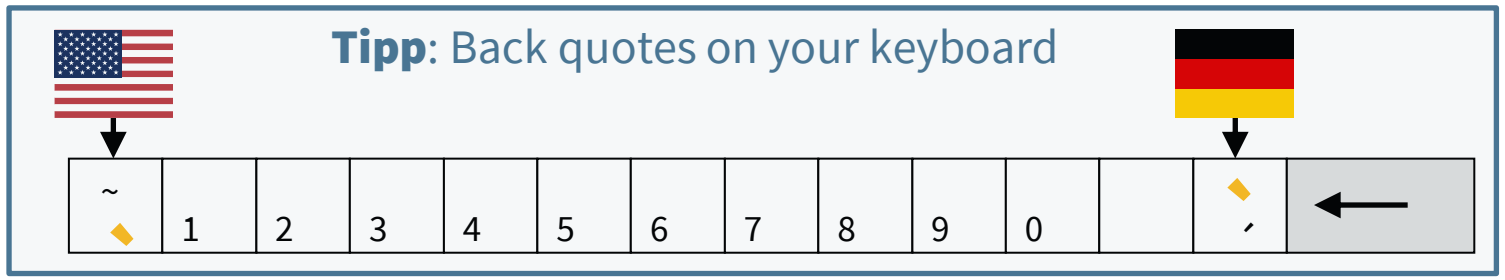
RMD Basics – Inline R Code

- R Markdown can also add R Results „inline“, meaning inserting it directly into Markdown text.
- e.g.: Text in Markdown, but outside of a chunk:

```
The model reached an adjusted R² of `r glance(iris_lm_1)$adj.r.squared`.
```

Inline code within back quotes ` and begins with „r“. R code runs and result is added to the text.

- For longer expressions: Prepare values in previous Code chunks and call them inline with a variable.



kableExtra:: – Great tables for HTML and PDF

- Create a table in PDF or HTML instead of printing dfs as Text:

```
```{r echo = FALSE}
df %>% kableExtra::kable()
```
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa

- In HTML: Beautify your tables

```
```{r echo = FALSE}
df %>% kable() %>% kable_styling()
```
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa

Tipp: `kable_styling()` has many helpful arguments to customize your table further!
I often use **`kable_styling(full_width = FALSE, position = "left")`** for smaller tables

DT:: – Interactive HTML tables

- `DT::` ports the powerfull DataTable library from JavaScript to R
- `DT::datatable()` creates interactive HTML tables that can be sorted, searched, and conditionally formatted (and much more)

````
```{r echo = FALSE}
df %>% DT::datatable()
```
````

Tipp: datatable() is more powerful than kable(), but settings can be complicated.

datatable() is wonderful for long tables, however!

See documentation for details: https://rstudio.github.io/DT/

Show 10 entries Search:

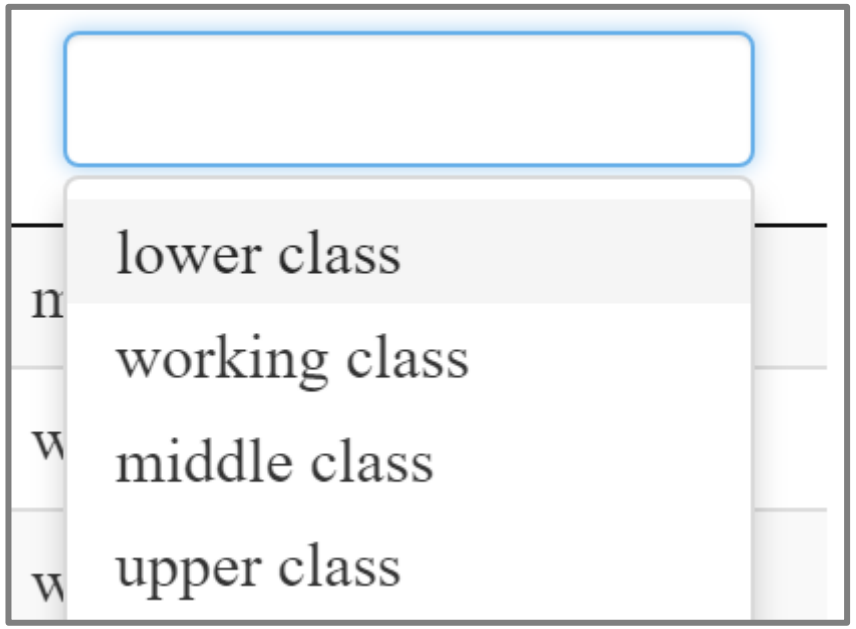
	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa

Showing 1 to 10 of 150 entries Previous 1 2 3 4 5 … 15 Next

DT:: – Example: Smart Filters

- You can make your datatable() filterable.
- `DT::datatable()` features smart filtering, depending on the datatype!
- Text search for character, sliders for numeric, and dropdown for factor

```
datatable(filter= "top")
```



R Output to dynamic Markdown

BONUS ☺

- Sometimes, we want reports to have dynamic elements that are added based on a data set with an arbitrary number of elements.
- This requires telling the Markdown engine to knit R output as if it were Markdown!
- Note that we need two things:
  1. The chunk option results = `results='asis'`
  2. Printing strings with `cat()` or with glue() instead of with print

````
```{r results='asis'}
cat("# RMD Heading from inside a code chunk")
```
````

R Output to dynamic Markdown

BONUS ☺

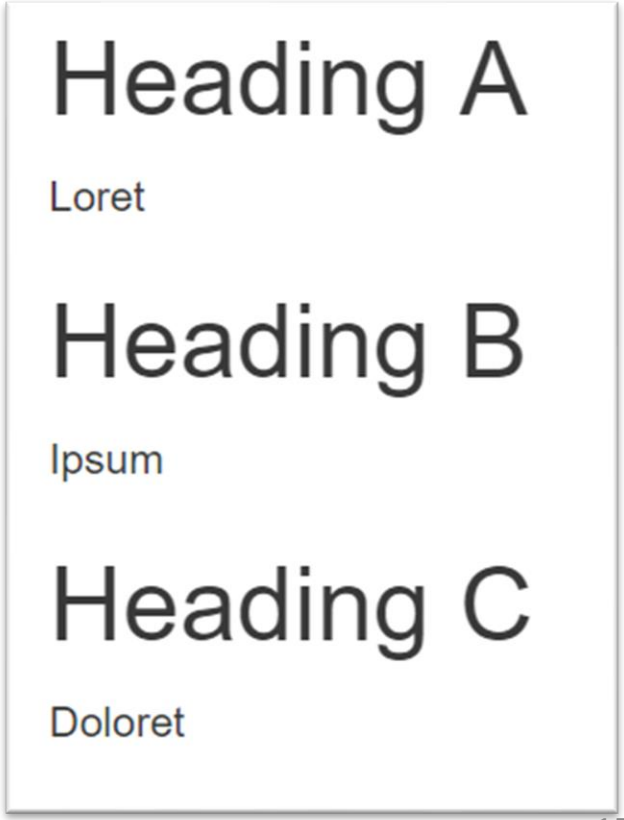
Content in data frame → Dynamic Markdown generation → Knitted Document

headings	content
# Heading A	Loret
# Heading B	Ipsum
# Heading C	Doloret

Loop through the data frame

```
```{r results='asis'}
walk2(text_df$headings, text_df$content,
      ~{
        cat(.x)
        cat("\n")
        cat(.y)
        cat("\n\n")}
      )
```
```

Add new lines to separate strings



`glue()` — Lightweight templating

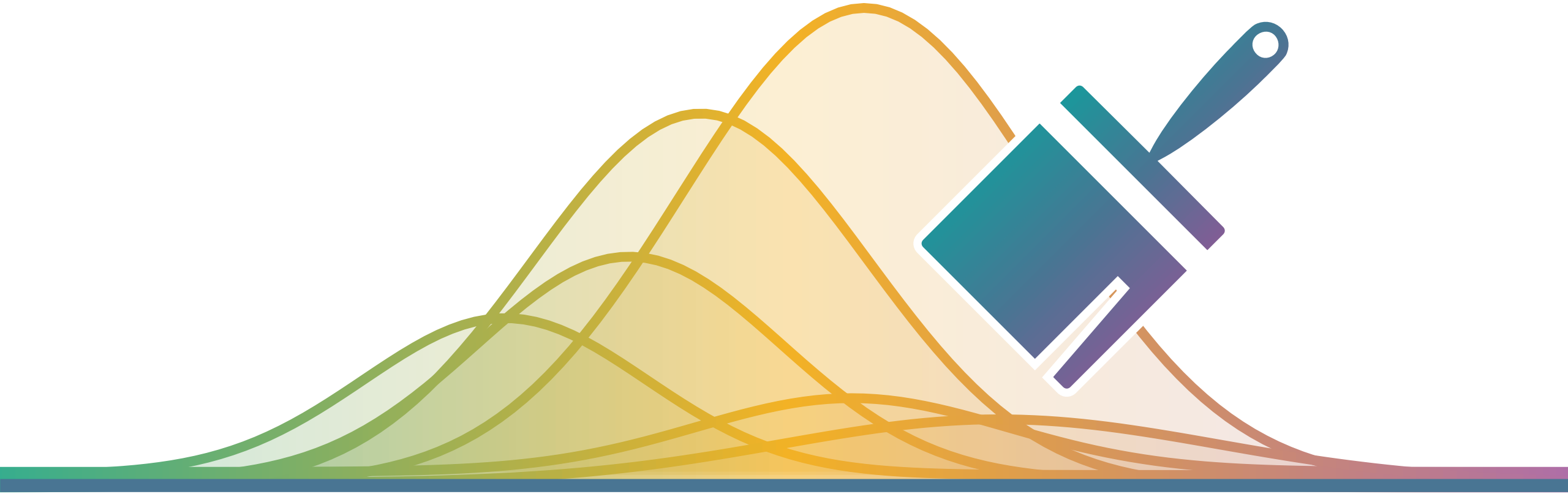
BONUS ☺

- For such dynamic markdown, we often need to combine static and dynamic string elements (i.e., templating).
- Base R does this with `paste0()`. However, `glue()` is more convenient.
- `glue()` takes a string and replaces parts in {} curly brackets with evaluated R code. This can be string variables or function calls.

```
string_var <- "dynamic elements"
glue("Some static text with {string_var}.")
```

- glue_data() can loop through data frames!
  So the code from the last slide becomes:

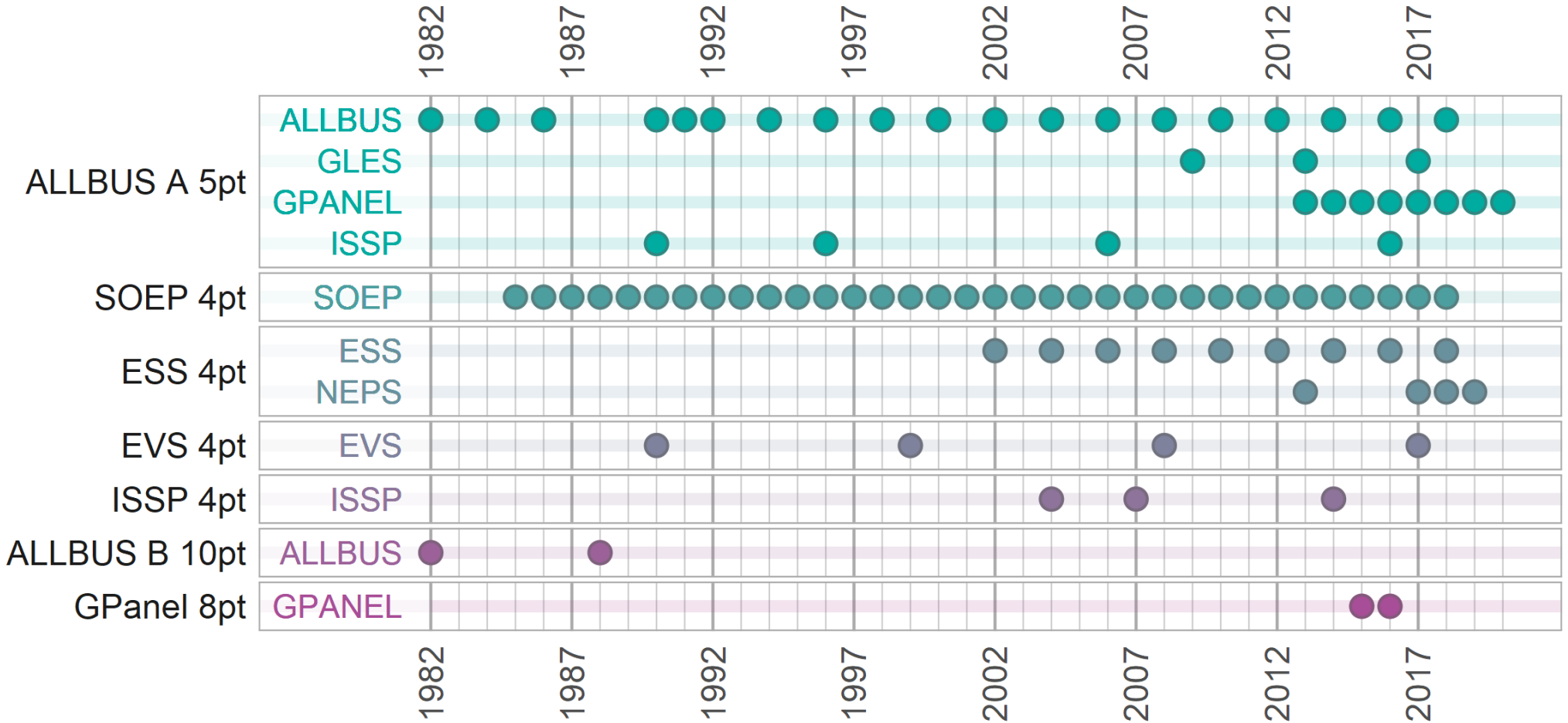
```
text_df %>%
glue_data("{headings}
          {content}\n\n")
```



Presenting Results: **Plots**

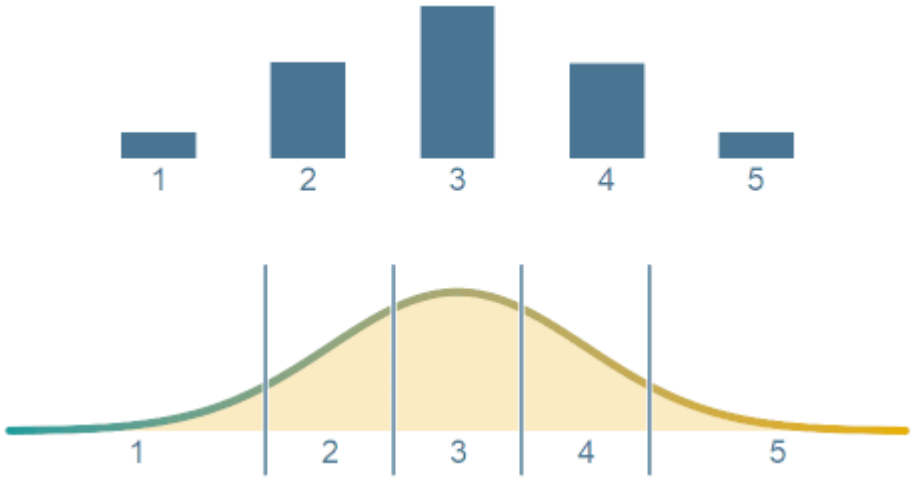
`ggplot()` default

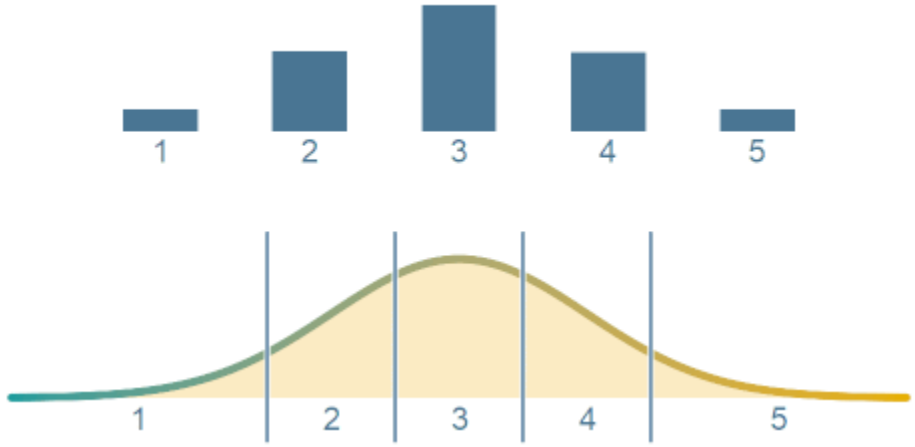
However, you can customize everything!



Even create plots, which no longer look like plots

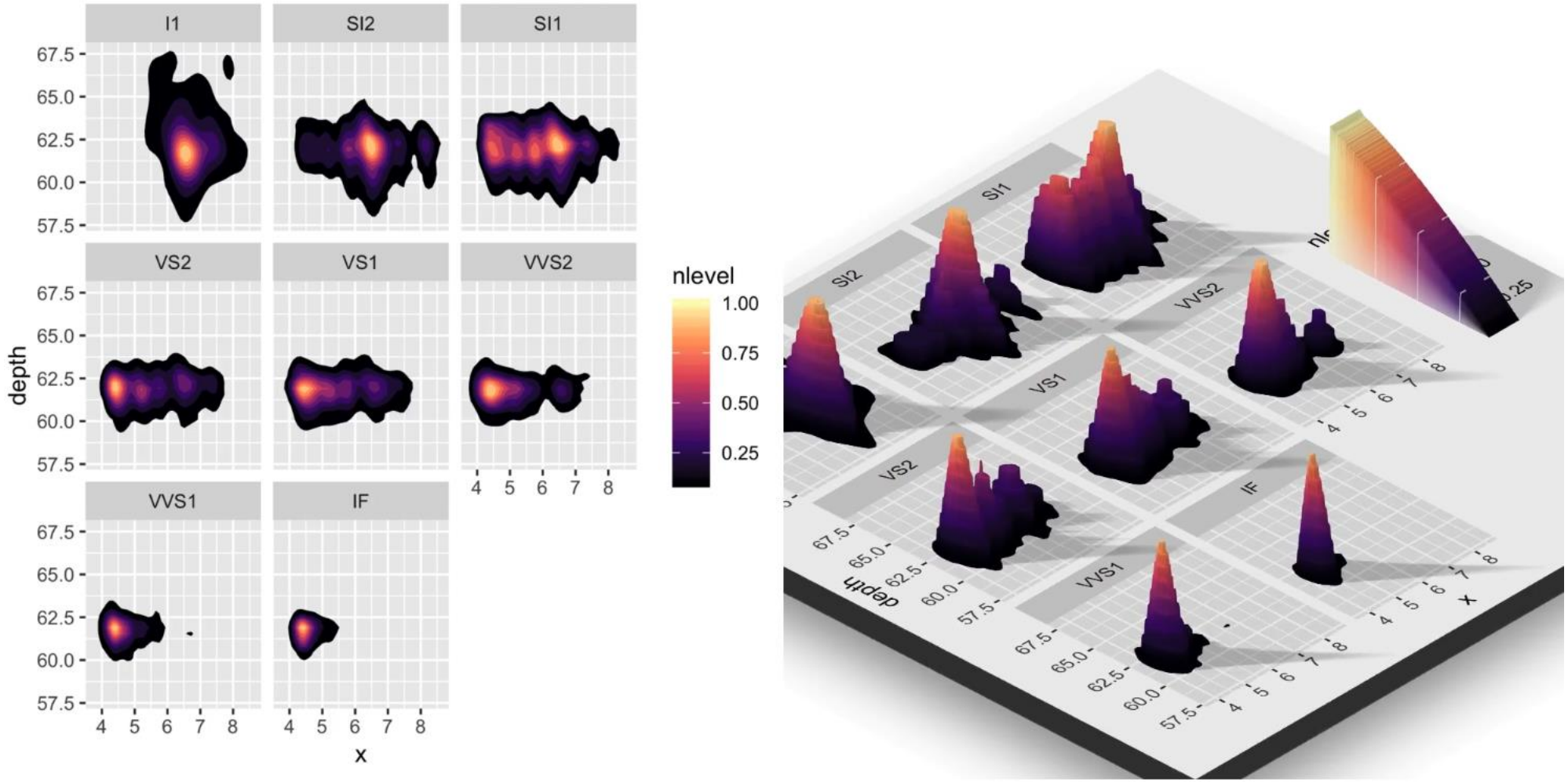
And **animations**! e.g., with `gganimate::`





All animations here were done in R:
https://blog.gesis.org/not-by-any-stretch-of-the-imagination-a-cautionary-tale-about-linear-stretching/

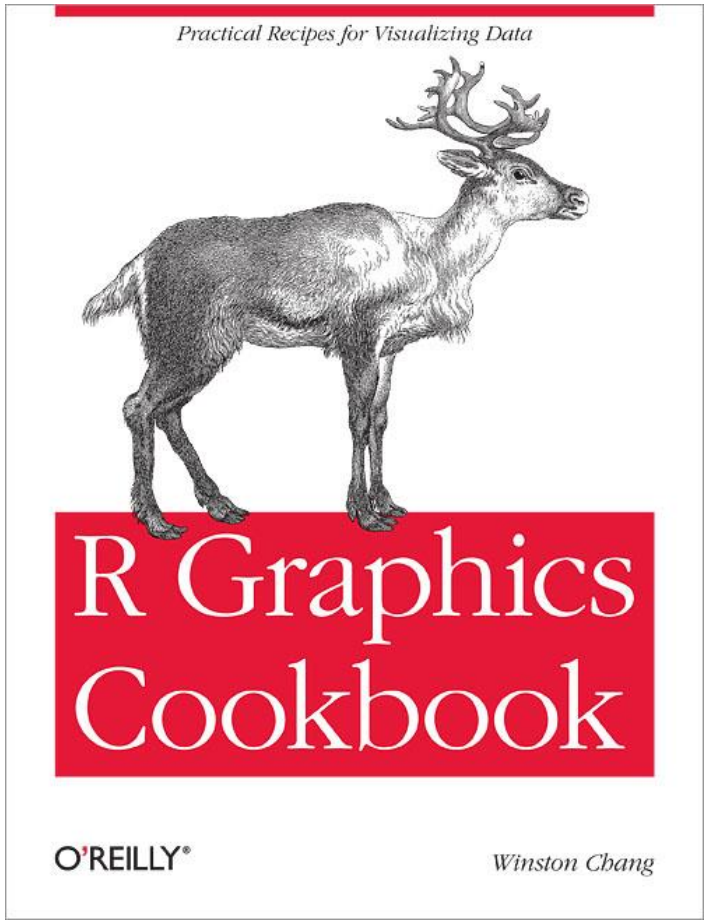
And (animated) **3D Plots** with rayshader::



Tyler Morgen-Wall: https://www.tylermw.com/3d-ggplots-with-rayshader/

Free (online) **Books** for `ggplot::`

- **R for Data Science**
  A quick introduction
  https://r4ds.had.co.nz/data-visualisation.html
- **R Graphics Cookbook**
  Detailed recipes for many common graphing tasks
  https://r-graphics.org
- **ggplot2: Elegant Graphics for Data Analysis**
  An in-depth look at ggplot2
  https://ggplot2-book.org
- **ggplot2 reference**
  The extensive ggplot2 reference listing every single detail
  https://ggplot2.tidyverse.org/reference



`ggplot2::` Customization

 Facetting and Patchworking

 Styling and Themes

 Saving Plots as image files

Facets – Splitting plots by variable

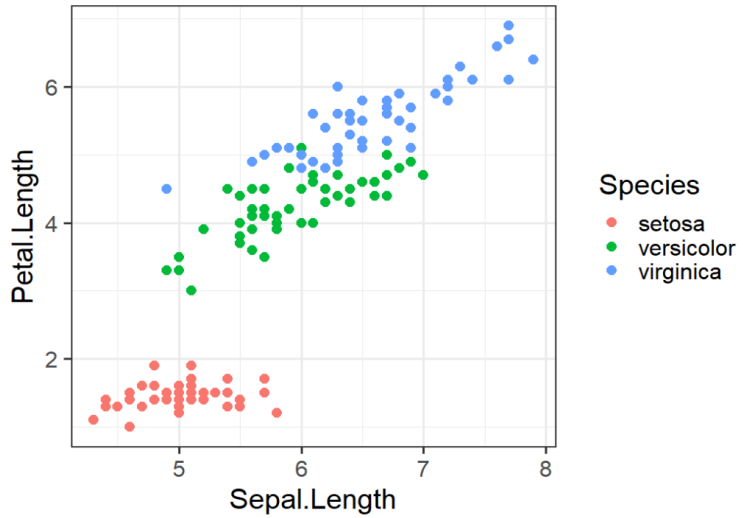
- Visualizing data and results for print usually means we have to combine many dimensions in one visualization
- Information **dimensions** can be assigned to:
  - **Position** (x and y)
  - **Style** (color, fill, shape, linetype)
- However, sometimes we need even more dimensions or the display is too **overloaded**
- **Facets** mean we split the visualization into different sub-plots; e.g., the same plot once for each group in a factor

„Three-dimensional“ plot

Third dimension via:

color

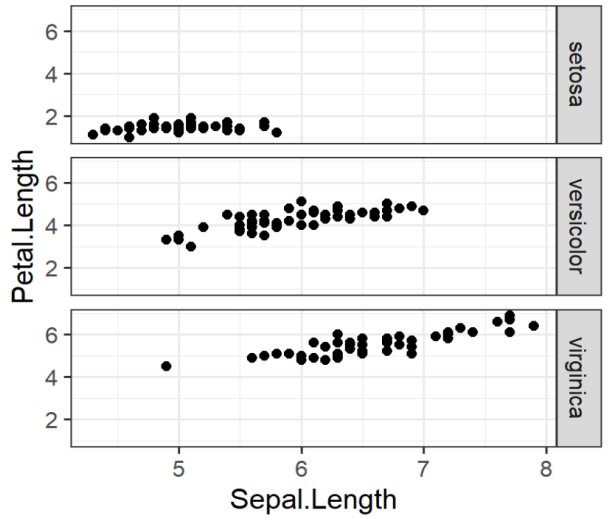
```
iris %>%
  ggplot(aes(Sepal.Length,
             Petal.Length,
             color = Species))+
  geom_point(size = 3)
```



Third dimension via:

facet

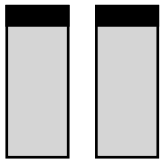
```
iris %>%
  ggplot(aes(Sepal.Length,
             Petal.Length))+
  facet_grid(rows = vars(Species))+
  geom_point(size = 3)
```



`facet_grid()` – Anatomy

- You can define **`rows`** or **`cols`**:

```
facet_grid(rows = vars(Species))
```



```
facet_grid(cols = vars(Species))
```

- Wrap variables governing the facets in **`vars()`**

```
facet_grid(rows = vars(Species))
```

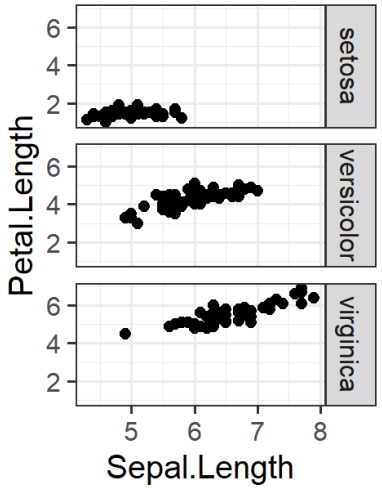
- By default, the x- and y- Axis of each facet have the same scale.Free them via:

```
facet_grid(rows = vars(Species), scales = "free")
```

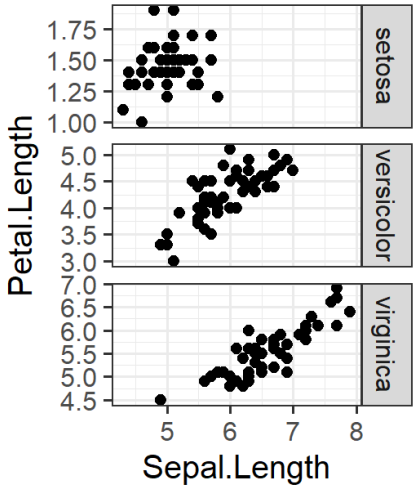
...or "free_x" or "free_y" for just one axis

scales:

fixed



free



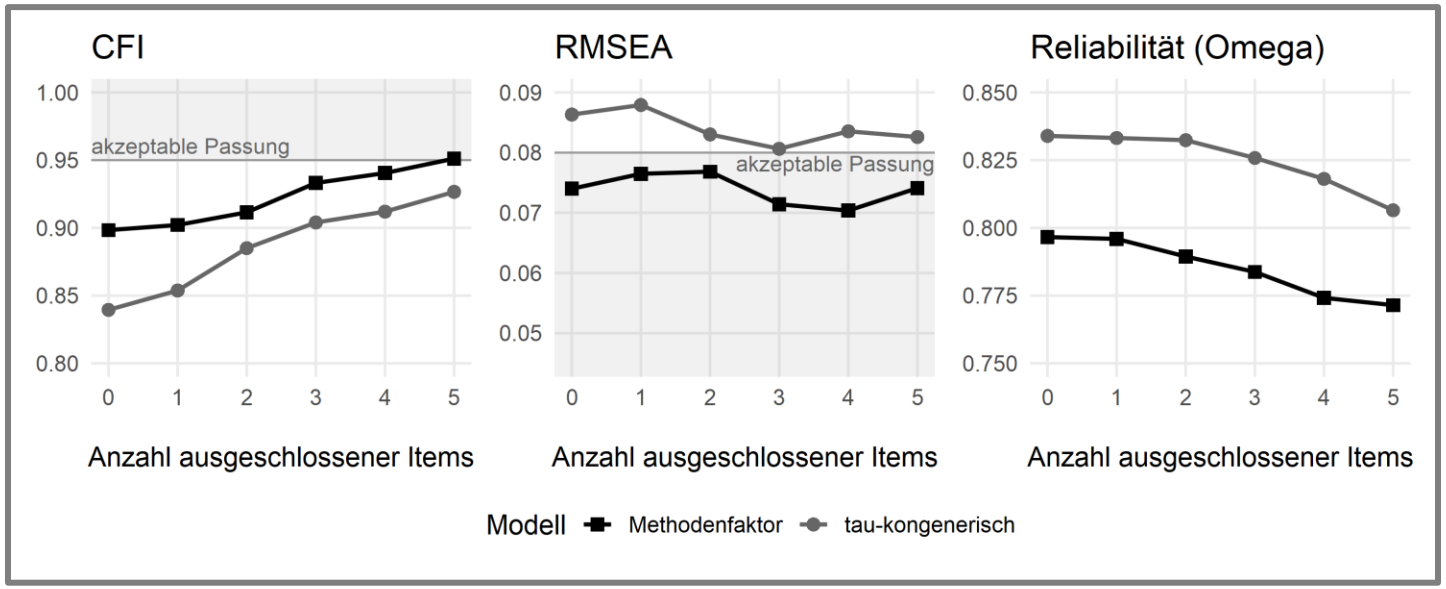
`patchwork::` – Combining plots into one

- Sometimes, we want to arrange different plots into a single visualization for print
- **`patchwork::`** makes this very easy.
  1. Save plots to variables: `plot_a <- ggplot(…)`
  2. Combine plots with „mathematical operators“: `cfa_p + rmsea_p + rel_p`

`plot_a + plot_b`

`plot_a / plot_b`

`(a + b) / c`



ggplot2:: - style basics

- There are three **style components**:
  - **Themes** govern how the non-data elements of the plot look (axis, legend etc.)
  - **Data styling** governs how geoms look
  - **Axis styling** governs the units and position of the axises

- **Themes** can be changes with…
  - **Complete themes** (i.e., presets)
  - Or manually **customized** via `theme()`

- **Data styling** can be…
  - static outside of `aes()`
  - data dependent within `aes()`

ggplot2::
Complete themes

Just add the respective theme function to a ggplot with +

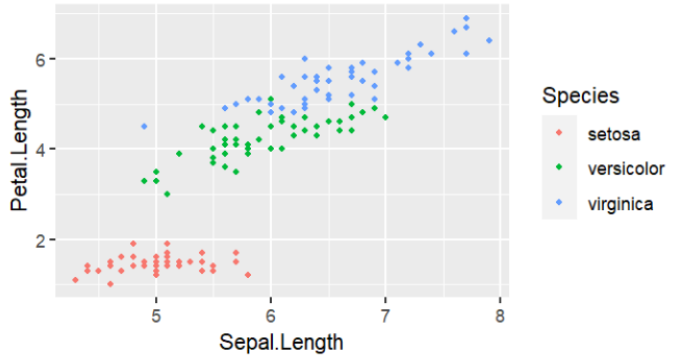
```
some_plot + theme_bw()
```

There are many presets. Right are four examples.

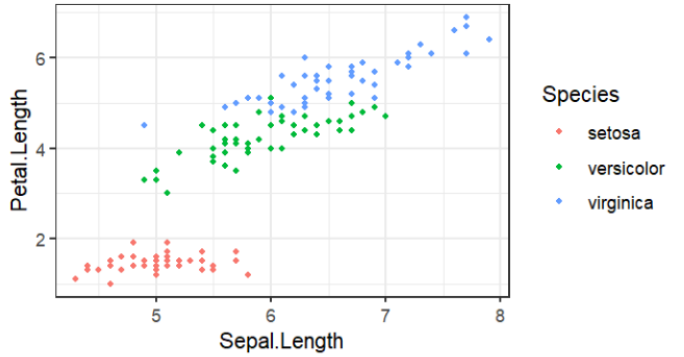
You can increase the fontsize and plot line thickness via:

```
theme_bw(base_size = 20)
```

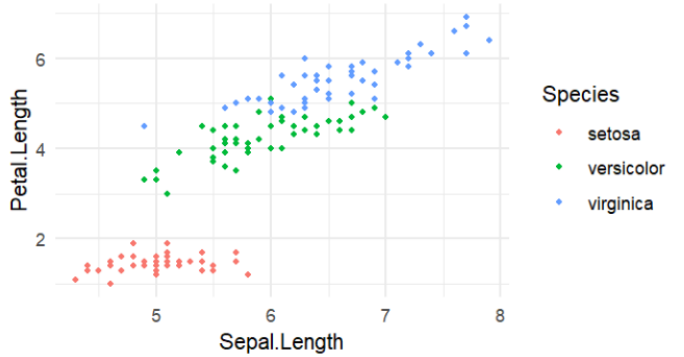
`theme_grey()`



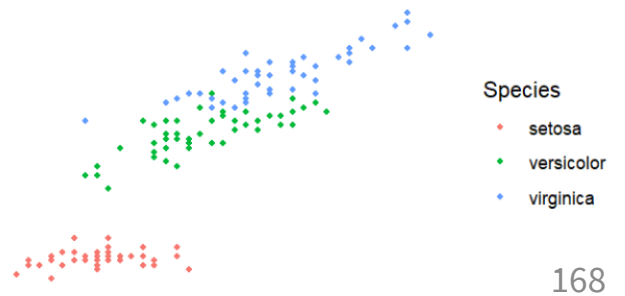
`theme_bw()`



`theme_minimal()`



`theme_void()`

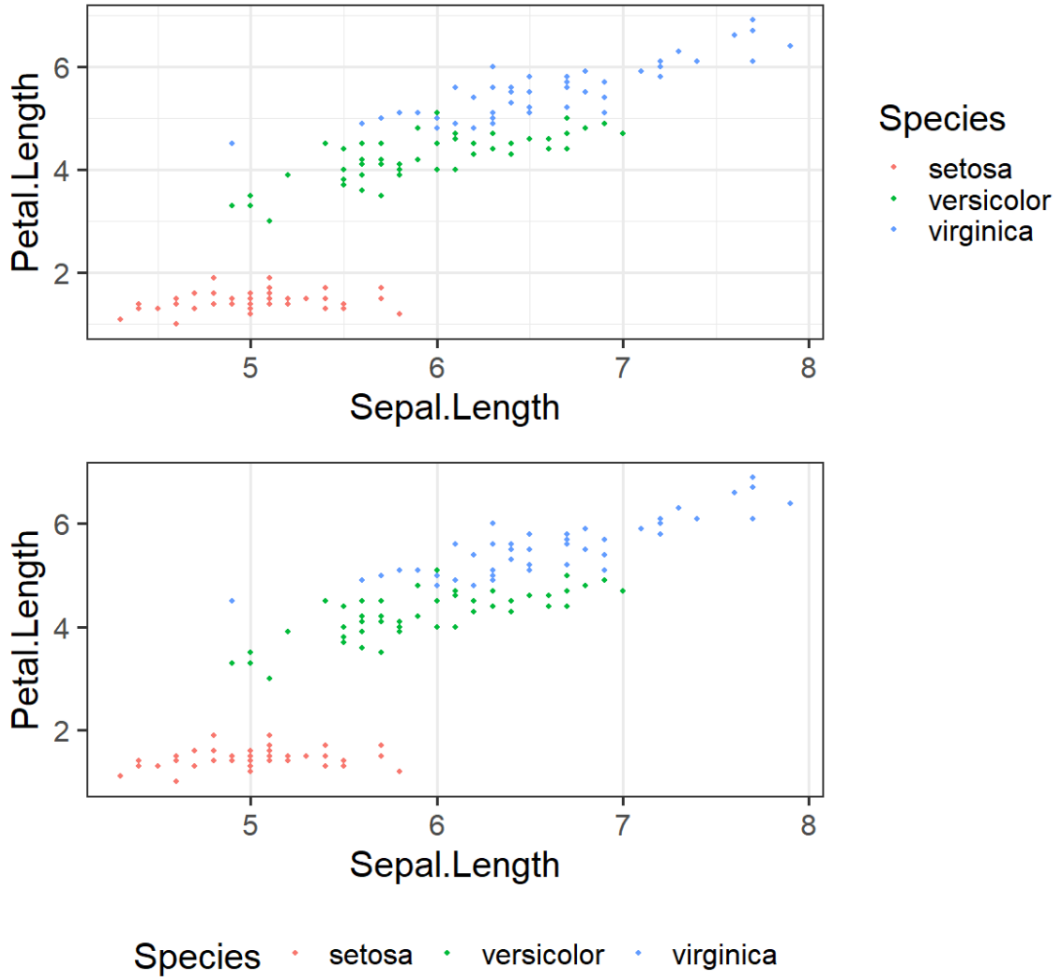


ggplot2:: custom theming

- We would usually start with a complete theme and then tweak it.

```
some_plot +
theme_bw(base_size = 20)+
theme(
  panel.grid.minor = element_blank(),
  panel.grid.major.y = element_blank(),
  legend.position = "bottom"
)
```

Tipp:
R graphics cookbook lists common arguments [link↗]



ggplot2:: data styling

- **Data styling** means changing the appearance of the graph elements drawn by geoms. (e.g., points, data lines, areas, bars)

- **Static styling** means applying decorative styles (e.g., making **all** points blue to fit your slide color theme)

```
geom_point(aes(x_var, y_var), color = "blue")
```
style **outside** `aes()`

- **Data dependent styling** means applying styles to signify data dimensionality (e.g., making points vary in color **depending** on a factor variable)

```
geom_point(aes(x_var, y_var, color = factor_var))
```
style **inside** `aes()`

`ggplot2::` data styling

Style arguments for geoms vary depending on their graphical elements.

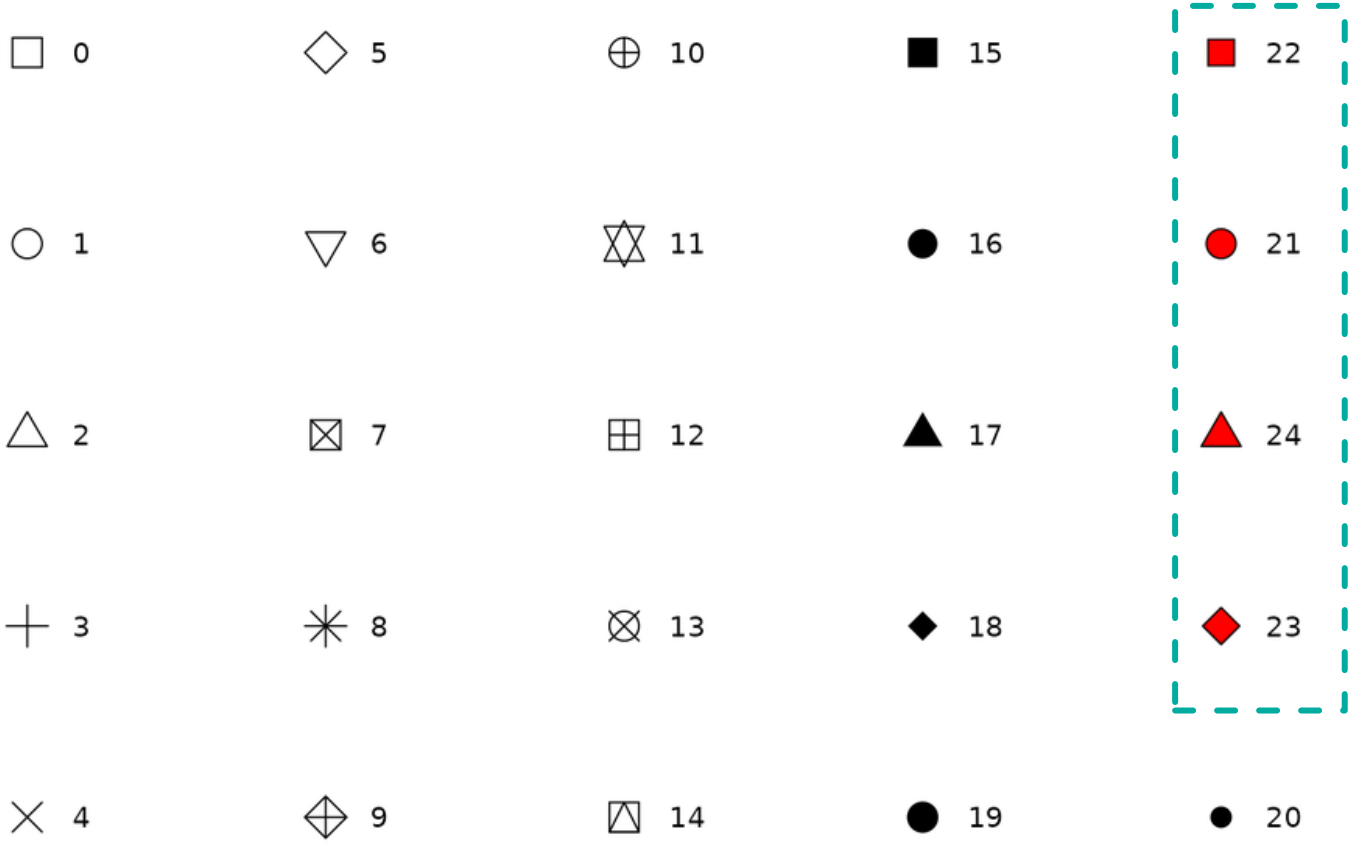
Here are basic style characteristics:

	points	lines	areas (e.g., bars)
`color`	depending on shape, changes whole point or only contour	changes line color	changes contour color
`fill`	(for some point shapes, changes inner color)	—	changes area color
`shape`	changes the point shape	—	—
`linetype`	—	changes line appearance (e.g., dashed lines)	changes contour appearance

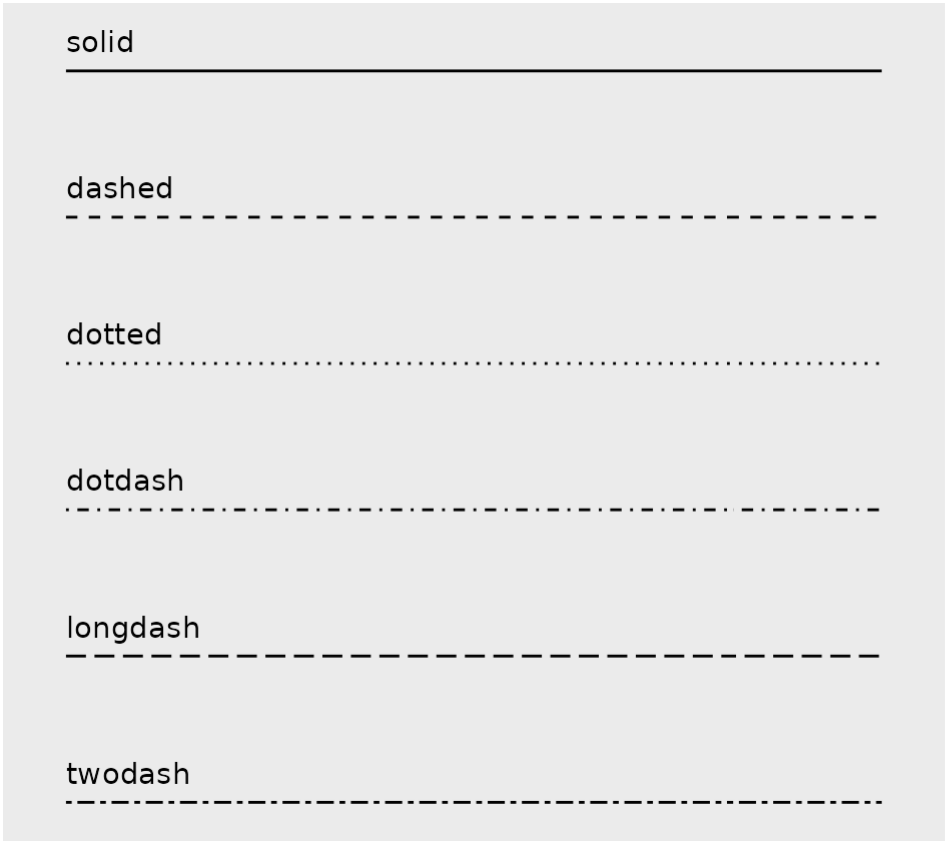
Point shapes and linetypes

shape

point shapes influenced by fill



linetype



Custom data dependent styles

Tipp: There is a difference between **discrete** (factor) and **continuous** (numeric)colors!

Data dependent styles choose default colors (or shapes, or linetypes) for different data values

We can **change those defaults** in several ways

1. Manual styling (discrete styles for factors):

Hex-colors for:

```
scale_color_manual(values = c("#497593", "#00ABA0", "#A84D97")
```

```
scale_fill_manual(), scale_shape_manual(), scale_linetype_manual()
```

2. Using color themes (discrete or continuous):

Greyscale	Brewer themes	viridis::
`scale_color_grey()` `scale_fill_grey()`	`scale_color_brewer(` `  palette = "Spectral")`	`library(viridis)` `scale_color_viridis()`

for discrete scales add argument:
`discrete = TRUE`

ggplot2:: **Axis** customization

- **Axes** can be customized in numerous ways.
- Depending on the plotted data, axes are usually either **continuous** or **discrete**.
- Axis customization is especially important for **continuous values**!

```
scale_x_continuous(
  "x-Axis name",
  breaks = c(1:10),
  minor_breaks = seq(1.1, 9.9, 0.1),
  limits = c(4, 9))
```

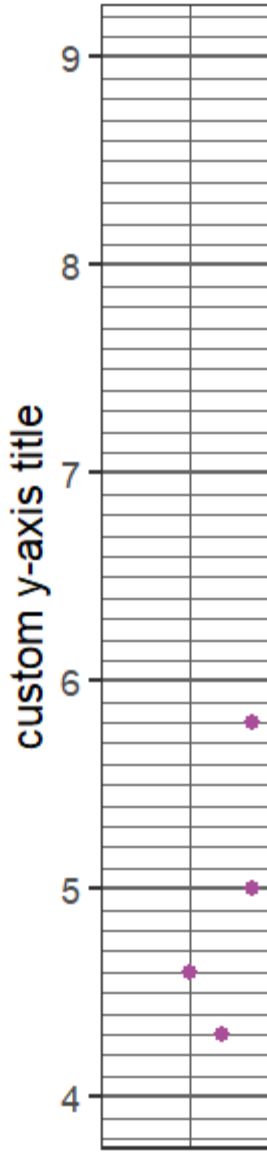
defines the continuous x-axis

Replaced the default variable name with a custom title

Defines major grid lines and axis numbers to display

Defines minor grid lines (thinner, without numbers)

Defines minimum and maximum values to plot



Tipp: Use **`seq()`** to customize breaks

`seq(1, 10, by = 0.5)`	Define start, end, and step width
`seq(3, 44, length.out = 10)`	Define start, end, and the desired number of steps

`ggsave()` – **Saving** plots to image file!

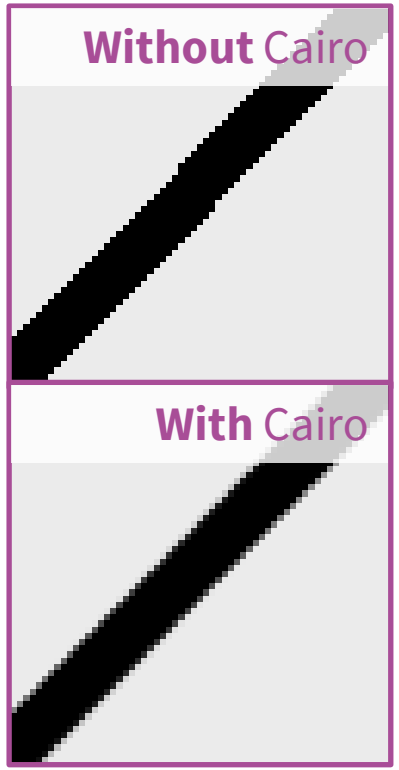
Fed up with journals asking for specific DPI?
`ggsave()` has your back ☺

Tipp: Install Cairo:: for smoother graphs!
`install.packages("Cairo")`

```
library(Cairo)
ggsave(filename = "plot.png",
       plot = some_plot,
       device = "png",
       type = "cairo",
       path = "images/",
       width = 10,
       height = 15,
       units = "cm",
       dpi = 300)
```

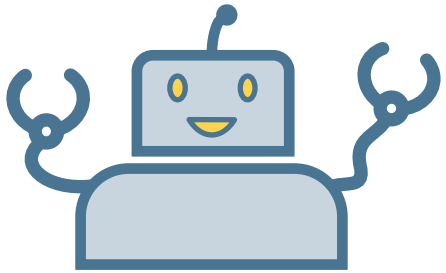
- For anti-aliasing (smooth lines)
- filename to save
- plot to save; otherwise: last displayed plot
- image type (redundant w. file name extension
- activate Cairo anti-aliasing (for Windows)
- save to a path (is added to filename)
- image width
- image height
- in unit: "in", "cm", "mm", "px“
- determines pixel size for inch, cm, and mm

Zoomed examples



Explore the R Universe!

Automation in R with **functionals**



- You can **automate** many **repetetive tasks** in R with functionals.
- Functionals take functions as arguments and apply them to all elements of a vector or dataframe (or list). The output is a vector, or dataframe (or list)

- **`Dplyr:: verbs`** (e.g., `mutate`) provide a functionals interface via **`across()`**

- **Tidyverse's `Purrr::`** meanwhile provides general purpose functionals.

`across()` – `mutate()` several columns at once

Selector — function — optional arguments for function

```
df %>%
  mutate(
    across(where(is.numeric),round, digits = 3)
  )
```

Tipp:
Write **custom functions** inside across()
Start with **~**
Then write code, referring to the current column with **.x**

```
~ scale(.x) %>% round(3)
```

Selectors	
`everything()`	selects all columns
`where()`	select by type with is.numeric, is.character, is.factor, is.labelled
`starts_with(), ends_with(), contains()`	select by varabiel name elements e.g., starts_with("extraversion_")
`1:3`	select via column indizes
`var_a:var_f`	select from var_a to var_f
`c(var_a, var_f, var_z)`	select those variables (do not omit c() !)

`across()`–
`summarise()` with multiple columns and summary functions

e.g., summarise all numeric variables in df with the aggegation functions mean, sd and max.

Selector

list of aggregation functions

argument passed to all aggegation functions!

```
summarise(
    across(where(is.numeric), list(mean = mean, sd = sd, max = max), na.rm = TRUE)
  )
```

Tipp: `across(.names)`
The summarised variables are **named** in the format `varname_function` (e.g., `age_mean`)
Change this with the across argument **`.names`**
Construct names with the components **`{.col}`** and **`{.fn}`** for column and function names.
`names = "{.fn}_{.col}"` results in **`mean_age`**, for example

Bonus tip: If you supply a custom name in **mutate()**, it will create a new variable and retain the original.
(e.g., names = **`"{.col}_z„`** results in **`num_var_z`**, while the original **`num_var`** is retained.)

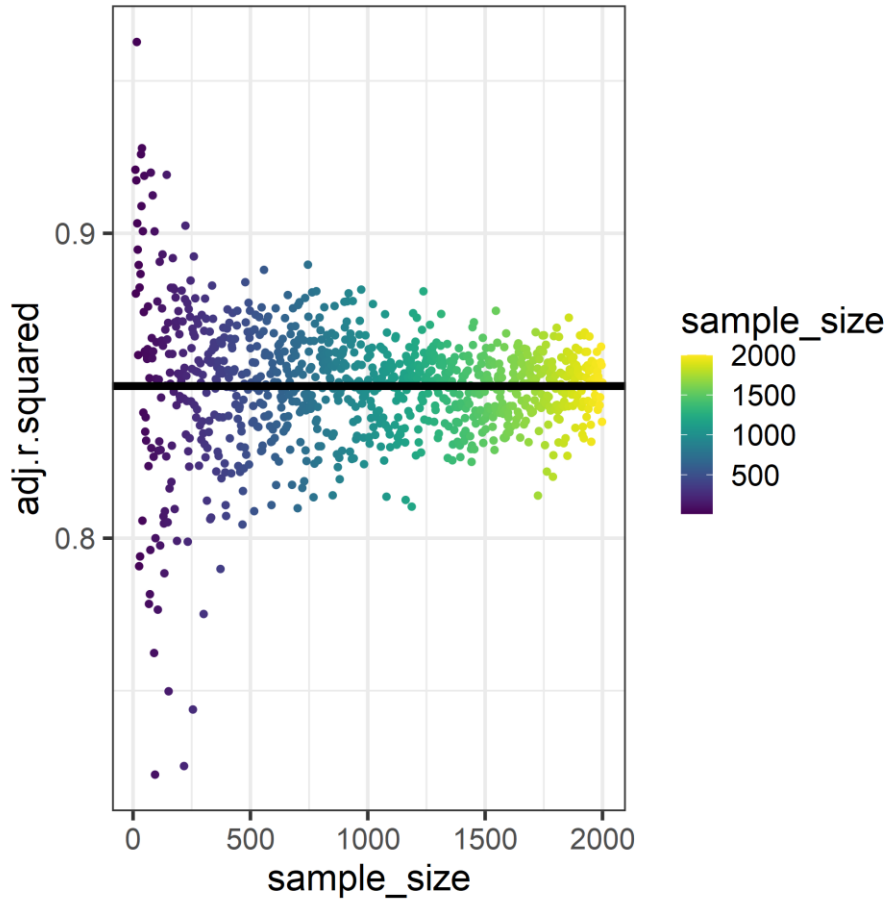
`purrr::` - Automate everything!

- The family of map() functions from purrr:: are like across, but for all kinds of objects in R!
- Once you feel more comfortable in R, purrr:: should be a next step!

- For an introduction:
  R for Data Science
- For a closer look:
  Advanced R
- Easy parallel processing for purr::
  `furrr::`

Calculate 1000 linear models with increasing sample size (n = 10 — n = 2000)

```
many_lms <- tibble(
  iteration = c(1:1000),
  sample_size = seq(10, 2000, length.out = 1000)
) %>%
  mutate(
    df_samples = map(sample_size, ~sample_n(diamonds, .x)),
    linear_models = map(df_samples, ~ lm(price ~ carat, data = .x)),
    model_summaries = map(linear_models, ~glance(.x))
  ) %>%
  unnest(cols = model_summaries)
```



Shiny – R powered Web Apps

Shiny creates interactive web applications powered by **R**.
Shiny provides both **back end** (server side) and **front end** (user interface).

There are many exciting **use cases**:

- Data **Dashboards**
- Interactive **data visulizations**
- Interactive **research publications**
- Dataset **download wizards**

See the shiny showcase for some examples.

RStudio

Version control and project organization

- **RStudio Projects** allow for tidy, self-contained data and code organization. [link↗]

- If a more formal environment is needed, you can create your own **Package**.
  (See Wickham‘s Book R Packages)

- **Version control** (e.g., via Git) makes collaboration and sharing easy and safe. [link↗]

Git Version Control

- **Git** (e.g., via GitLab) provides **file sharing capabilities**
- It remembers past versions of your code and makes reverting to earlier versions easy.
- Projects can also be „**branched**“, meaning that **independent project copies** can be split from the original project.
  - That way, **new features** can be developed and tested safely.
  - **Different versions** of a tool can be used and maintained in parallel.
  - **Paper revisions** become a breeze! (Well, not really. ☹ But at least the project organization is less burdonsome. ☺)
  - Branches can later be **merged**, for example to add the tested features.