

A scenic view of the Magdeburg skyline across the River Elbe. In the background, the Gothic Magdeburg Cathedral with its two towers stands prominently. To the right, the Marienkirche (St. Mary's Church) with its red-tiled roof and two spires is visible. The foreground shows the river with several boats, including a large white passenger ship and some smaller industrial vessels. A bridge spans the river in the middle ground.

# 05 - The R language: Vectors, Classes, Functions, Iteration

Data Science with R · Summer 2021

Uli Niemann · Knowledge Management & Discovery Lab

<https://brain.cs.uni-magdeburg.de/kmd/DataSciR/>

# Outline

## Vectors:

- [Basic vector operations](#)
- [Vector types](#)
- [Vector properties](#)
- [Vector coercion](#)
- [Creating vectors](#)
- [Special values](#)
- [Vector recycling](#)
- [Lists](#)

## Classes:

- [Data frames](#)
- [Factors](#)
- [Dates](#)

## Functions

## Iteration

# Vectors

# Basic vector operations

Suppose we have the following vector:

```
(v <- c(5, 2, 9, 1, 6, 2, 4, 5, 5, 1))
```

```
## [1] 5 2 9 1 6 2 4 5 5 1
```

Sort the elements of a vector with `sort()`:

```
sort(v) # order() returns the order of the values as vector indices
```

```
## [1] 1 1 2 2 4 5 5 5 6 9
```

Get the unique values of a vector by the order in which they appear with `unique()`:

```
unique(v)
```

```
## [1] 5 2 9 1 6 4
```

Create a contingency table for a vector with `table()`:

```
table(v)
```

```
## v
## 1 2 4 5 6 9
## 2 2 1 3 1 1
```

# Vectors types

- In `R`, there are two vector types: **atomic vectors** and **lists**.
- An atomic vector is a sequence of elements of the **same data type**.
- Lists are **recursive vectors**, i.e., lists can contain other lists.

The most important atomic data types are:

- `logical`: `FALSE` or `TRUE`
- `integer`: whole number, e.g. `5L`
- `double`: floating-point number, e.g. `3.4`
- `character`: character string, e.g. `"DataSciR"`

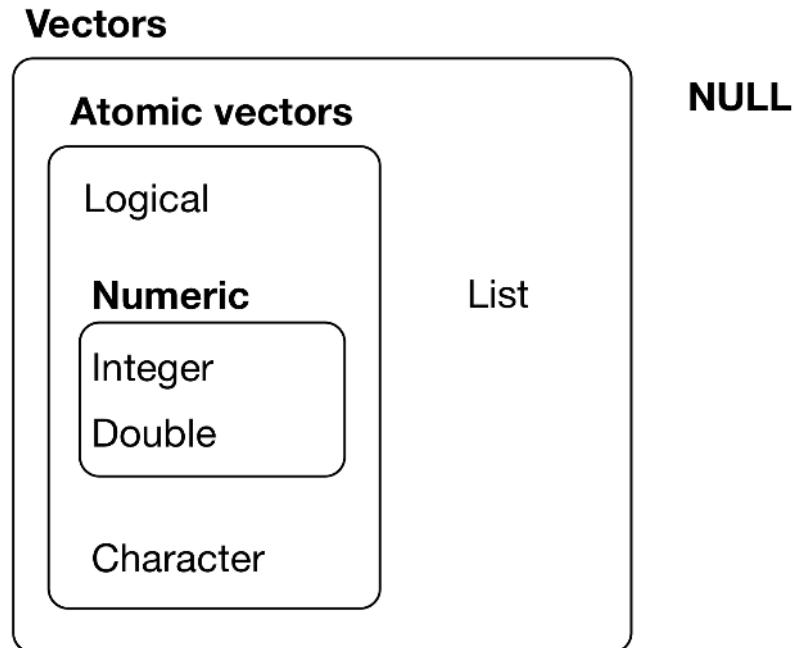


Figure source: Hadley Wickham and Garrett Grolemund. *R for Data Science*. O'Reilly, 2017.

# Vector properties

Vectors have two major properties: **type** and **length**.

```
x <- 1:5
typeof(x)

## [1] "integer"

length(x)

## [1] 5
```

# Naming vector elements

Naming and renaming of vector elements:

```
names(x) <- letters[1:5] # same as x <- c(a = 1, b = 2, c = 3, d = 4, e = 5)  
x
```

```
## a b c d e  
## 1 2 3 4 5
```

```
typeof(x)
```

```
## [1] "integer"
```

# Accessing vector elements

```
x[2] # the second element
```

```
## b  
## 2
```

```
x["b"] # the element named "b"
```

```
## b  
## 2
```

```
x[-2] # all elements but the second
```

```
## a c d e  
## 1 3 4 5
```

```
x[-(3:5)] # all elements but the third to fifth
```

```
## a b  
## 1 2
```

# Vector coercion

An atomic vector must be homogeneous with respect to the type of its elements. If you create a vector with elements of mixed types, R tries to convert the elements into the **most flexible** type of its elements. This process is called **vector coercion**. As a result, creating a vector with elements of mixed types does not yield an error.

**Order of types**, from *least flexible* to *most flexible*:

1. `logical(FALSE, TRUE)`
2. `integer(1L, 2L)`
3. `double(0.51, 3.19)`
4. `character("abc", "xz")`

```
x <- c(1, 4, 1, 3, 2)
typeof(x) # why not integer?
```

```
## [1] "double"
```

```
x <- c(1, 4, 1, 3, 2, "4")
x
```

```
## [1] "1" "4" "1" "3" "2" "4"
```

```
typeof(x)
```

```
## [1] "character"
```

# Implicit coercion

Functions that require a specific vector type use implicit coercion:

```
# Create a random boolean vector with 10 elements.  
lg1_vec <- sample(c(FALSE, TRUE), size = 10, replace = TRUE)  
lg1_vec
```

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

```
sum(lg1_vec) # sum() requires numeric type ---> FALSE -> 0, TRUE -> 1
```

```
## [1] 4
```

```
mean(lg1_vec) # mean() requires numeric type ---> FALSE -> 0, TRUE -> 1
```

```
## [1] 0.4
```

```
paste("this", "is", "a", "test") # concatenate strings
```

```
paste(lg1_vec, collapse = " ")
```

```
## [1] "this is a test"
```

```
paste(c("this", "is", "a", "test"), collapse = " ")
```

```
## [1] "this is a test"
```

⌚ What is the result?

```
## [1] "TRUE FALSE FALSE TRUE FALSE TRUE FALSE F
```

# Explicit coercion

```
x <- c(1, 4, 1, 3, 2, "4.0")  
x
```

```
## [1] "1"   "4"   "1"   "3"   "2"   "4.0"
```

Convert a `character` vector into a `double` vector (explicit coercion):

```
as.numeric(x)
```

```
## [1] 1 4 1 3 2 4
```

Explicit coercion can also be realized with `as.logical()`, `as.integer()`, `as.double()` and `as.character()`.

# Creating vectors

**Vectors** can be created with `c()`, `:`, `seq()` or `rep()`.

```
# "C"ombine elements to a vector  
c(1,2,3)
```

```
## [1] 1 2 3
```

```
# integer sequence  
1:3
```

```
## [1] 1 2 3
```

```
# sequence with an increment of 0.5  
seq(1, 3, by = 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0
```

```
# repeat the whole vector  
rep(1:3, times = 2)
```

```
## [1] 1 2 3 1 2 3
```

```
# repeat each element of the vector  
rep(1:3, each = 2)
```

```
## [1] 1 1 2 2 3 3
```

There is no separate data structure for **scalars** in `R`. Scalars are simply vectors of length 1.

# Special values

**Missing values** or **unknown values** are represented as `NA` (*not applicable*).

These special values can be identified with `is.na()`.

```
is.na(c(1, NA, 5))
```

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

Further special values besides `NA` include:

- `NaN` (**not a number**): e.g. `sqrt(-2)` → `is.nan()`
- `Inf`: e.g. `1/0` → `is.infinite()`
- `NULL`: absence of a whole vector → `is.null()`

# Vector recycling

When combining two vectors, R tries to match their lengths.

**Vector recycling** involves replicating elements of the shorter of two vectors so that the two vectors' lengths are equal.

```
1:6 + 1:3
```

```
## [1] 2 4 6 5 7 9
```

```
1:5 + 1:3
```

⌚ What is the result?

```
1:5 + 1:3
```

```
## Warning in 1:5 + 1:3: longer object length is  
## not a multiple of shorter object length
```

```
## [1] 2 4 6 5 7
```

Equivalent with:

```
1:5 + c(1:3, 1, 2)
```

```
## [1] 2 4 6 5 7
```

# Lists

A **list** is a **recursive vector**, because it can contain other vectors.

Create a list with `list()`:

```
x <- list(1:5)
x
## [[1]]
## [1] 1 2 3 4 5

y <- list(1, 2, 3, 4, 5)
y
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5
```

Inspect the **structure** of a list with `str()`:

```
str(y)
## # List of 5
## $ : num 1
## $ : num 2
## $ : num 3
## $ : num 4
## $ : num 5
```

# Lists

A list is a vector that can contain **elements of different types**:

```
x <- list(TRUE, 1L, 1.23, "u")  
str(x)
```

```
## List of 4  
## $ : logi TRUE  
## $ : int 1  
## $ : num 1.23  
## $ : chr "u"
```

Lists can contain other lists:

```
x <- list(list(TRUE, 1L), list(1.23, "u"))  
str(x)
```

```
## List of 2  
## $ :List of 2  
##   ..$ : logi TRUE  
##   ..$ : int 1  
## $ :List of 2  
##   ..$ : num 1.23  
##   ..$ : chr "u"
```

# List subsetting

There are 3 ways to access list elements:

- `[]` extracts a  **sublist**. The results is **always** a list,
- `[[]]` extracts a **single element** and removes one level of hierarchy,
- `$` extracts a **named element**.

```
x <- list(  
  list(TRUE, 1L),  
  list(1.23, "u"))  
str(x)  
  
## List of 2  
## $ :List of 2  
##   ..$ : logi TRUE  
##   ..$ : int 1  
## $ :List of 2  
##   ..$ : num 1.23  
##   ..$ : chr "u"
```

```
str(x[1])  
  
## List of 1  
## $ :List of 2  
##   ..$ : logi TRUE  
##   ..$ : int 1
```

```
str(x[[1]])  
  
## List of 2  
## $ : logi TRUE  
## $ : int 1
```

```
str(x[[1]][[1]])  
  
## logi TRUE
```

# List subsetting

Compare list subsetting to this unusual pepper shaker:



Figure adapted from: Hadley Wickham and Garrett Grolemund. [\*"R for Data Science"\*](#). O'Reilly, 2017.

# Classes

# Classes

- Classes are used to create more complex data structures
- The class attribute of a vector determines its behavior
- The class attribute mostly refers to the [S3 class](#)
  - Examples: data frames, factors, and dates

# Data frames

Data frames are built on top of regular lists. Thus, element indexing works similar. An important difference between data frames and lists is that a data frame requires its vectors (variables) to have the same length while a list does not.

Example: the `mtcars` dataset

*The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models). — `?mtcars`*

```
mtcars
```

```
##          mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6 160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4 108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0  0    3    2
## Valiant       18.1   6 225 105 2.76 3.460 20.22  1  0    3    1
```

`class(mtcars) → data.frame`    `typeof(mtcars) → list`

# Extracting data frame elements

You can extract a single element (such as a variable of a data frame) with `[[[]]]`, `$`, and `dplyr::pull()`:

```
mtcars[[1]] # [[[]]] using the column index
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7
## [18] 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4
```

```
mtcars[["mpg"]] # [[[]]] using a character string
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7
## [18] 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4
```

```
mtcars$mpg # $ (without quotation marks!)
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7
## [18] 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4
```

```
mtcars %>% pull(mpg) # pull is a dplyr function
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7
## [18] 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4
```

To access one or more elements, you can use `[]`:

```
mtcars[c("mpg", "disp")]
```

```
##                                     mpg   disp
## Mazda RX4           21.0 160.0
## Mazda RX4 Wag      21.0 160.0
## Datsun 710          22.8 108.0
## Hornet 4 Drive     21.4 258.0
## Hornet Sportabout   18.7 360.0
## Valiant             18.1 225.0
## Duster 360          14.3 360.0
## Merc 240D           24.4 146.7
## Merc 230             22.8 140.8
## Merc 280             19.2 167.6
## Merc 280C            17.8 167.6
## Merc 450SE           16.4 275.8
## Merc 450SL           17.3 275.8
## Merc 450SLC          15.2 275.8
## Cadillac Fleetwood  10.4 472.0
## Lincoln Continental 10.4 460.0
## Chrysler Imperial    14.7 440.0
## Fiat 128              32.4 78.7
## Honda Civic            30.4 75.7
## Toyota Corolla         33.9 71.1
## Toyota Corona          21.5 120.1
## Dodge Challenger       15.5 318.0
## AMC Javelin            15.2 304.0
## Camaro Z28              13.3 350.0
## Pontiac Firebird        19.2 400.0
## Fiat X1-9                27.3 79.0
## Porsche 914-2           26.0 120.3
```

You can filter rows and subset variables at the same time:

```
mtcars[1:3, c("mpg", "disp")]
```

```
##          mpg disp
## Mazda RX4    21.0 160
## Mazda RX4 Wag 21.0 160
## Datsun 710   22.8 108
```

Alternatively, if the rows are named:

```
mtcars[c("Mazda RX4", "Mazda RX4 Wag",
        "Datsun 710"),
       c("mpg", "disp")]
```

```
##          mpg disp
## Mazda RX4    21.0 160
## Mazda RX4 Wag 21.0 160
## Datsun 710   22.8 108
```

# Factors

- A **factor** is a vector class built on top of an **integer** vector.
- A factor vector contains predefined categorical values, the so-called **levels**.

```
(gender <- factor(c("m", "f", "f", "m", "m"), levels = c("f", "m", "d")))
```

```
## [1] m f f m m  
## Levels: f m d
```

```
class(gender)
```

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

```
typeof(gender)
```

```
## [1] "integer"
```

```
levels(gender)
```

```
## [1] "f" "m" "d"
```

# Factors

A values that is not an element of the set of levels must not be used:

```
gender[2] <- "male" # only "f", "m", and "d" allowed

## Warning in `<- .factor`(`*tmp*`, 2, value = "male"): invalid factor level, NA generated

gender

## [1] m      <NA> f      m      m
## Levels: f m d
```

# Factors

Factors are also useful if all possible values are known in advance but some of them are not observed initially.

```
table(gender)
```

```
## gender  
## f m d  
## 1 3 0
```

Compare with:

```
x <- as.character(gender)  
table(x)
```

```
## x  
## f m  
## 1 3
```

# The `forcats` package

The core Tidyverse package `forcats` provides functions to modify categorical variables.

Function	Description
<code>fct_reorder()</code>	Change the levels order by the values of another variable
<code>fct_inorder()</code>	Change the levels order by the order in which they first appear
<code>fct_rev()</code>	Reverse the levels order
<code>fct_relevel()</code>	Change the order of one or more levels manually
<code>fct_reorder2()</code>	Change the levels order by the values of a second variable. Which value of the second variable is considered is dependent on the highest value of a third variable.
<code>fct_collapse()</code>	Merge two or more levels
<code>fct_recode()</code>	Rename levels
<code>fct_lump()</code>	Merge levels based on their occurrence
...	



# The data: descriptors of socio-economic status in the US

## Wealth and income in the USA over time

## Employed Status

```
income <- read_rds(here::here("data/income.rds"))
glimpse(income)
```

```
## Rows: 1,854
## Columns: 5
## $ year          <dbl> 2019, 2019, 2019, 2019, 2019, 2018, 2018, 2018, 2018~
## $ race          <chr> "All Races", "All Races", "All Races", "All Races", "All Races", ~
## $ number        <dbl> 128451000, 128451000, 128451000, 128451000, 128451000, 128451000~
## $ income_quintile <chr> "Lowest", "Second", "Third", "Fourth", "Highest", "Top 5%", "Low~
## $ income_share   <dbl> 3.1, 8.3, 14.1, 22.7, 51.9, 23.0, 3.1, 8.3, 14.1, 22.6, 52.0, 23~
```

"The US Census provides Historical Income Tables, of which we have joined several to compare wealth and income over time by race."

Source: [TidyTuesday](#)

# The data: descriptors of socio-economic status in the US

# Wealth and income in the USA over time

## Employed Status

## Employed persons by industry, sex, race, and occupation in the US between 2015 and 2020

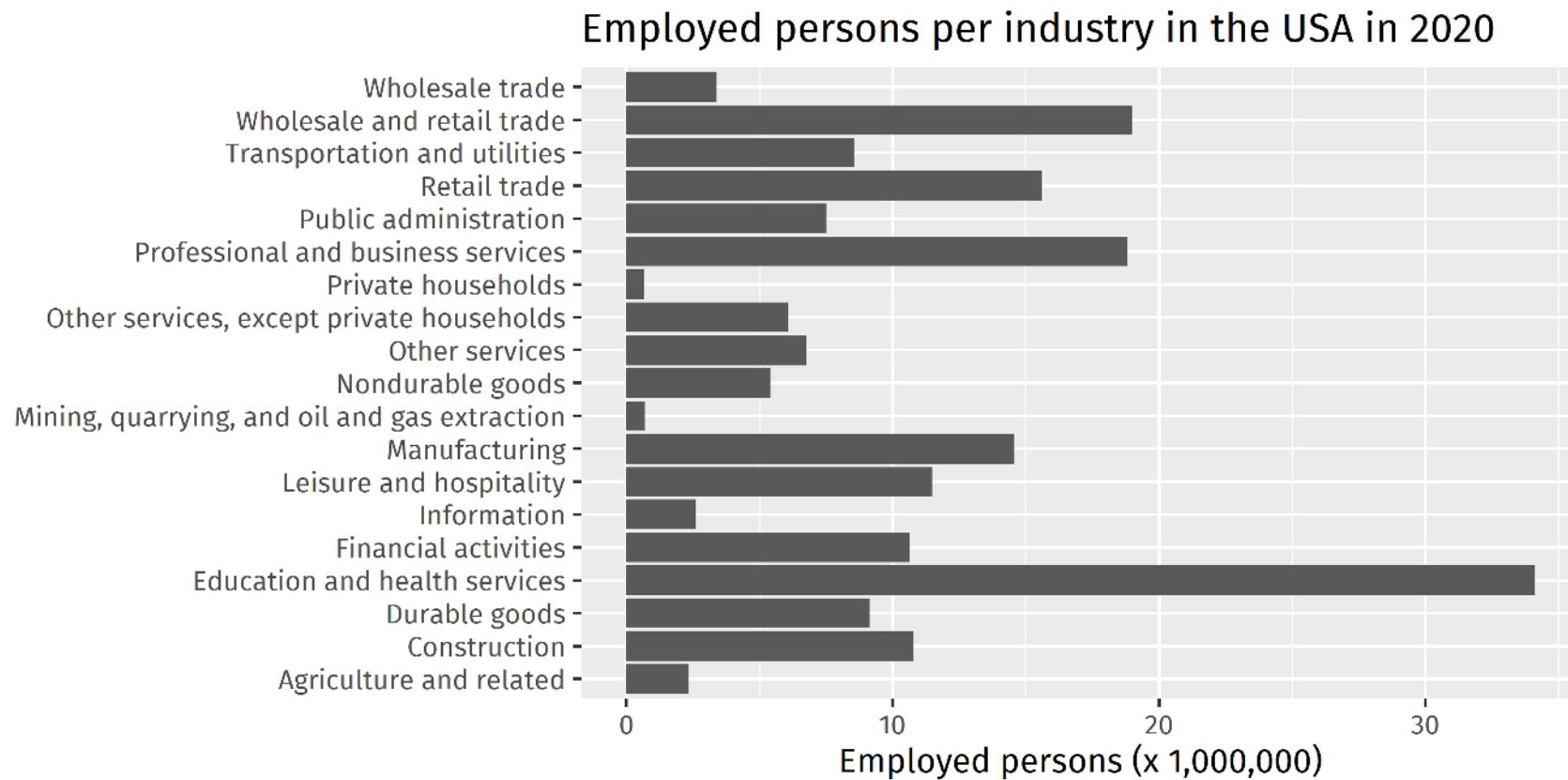
Source: TidyTuesday

## fct\_reorder()

Default: alphabetic order

Industries ordered by number of employed persons

Code

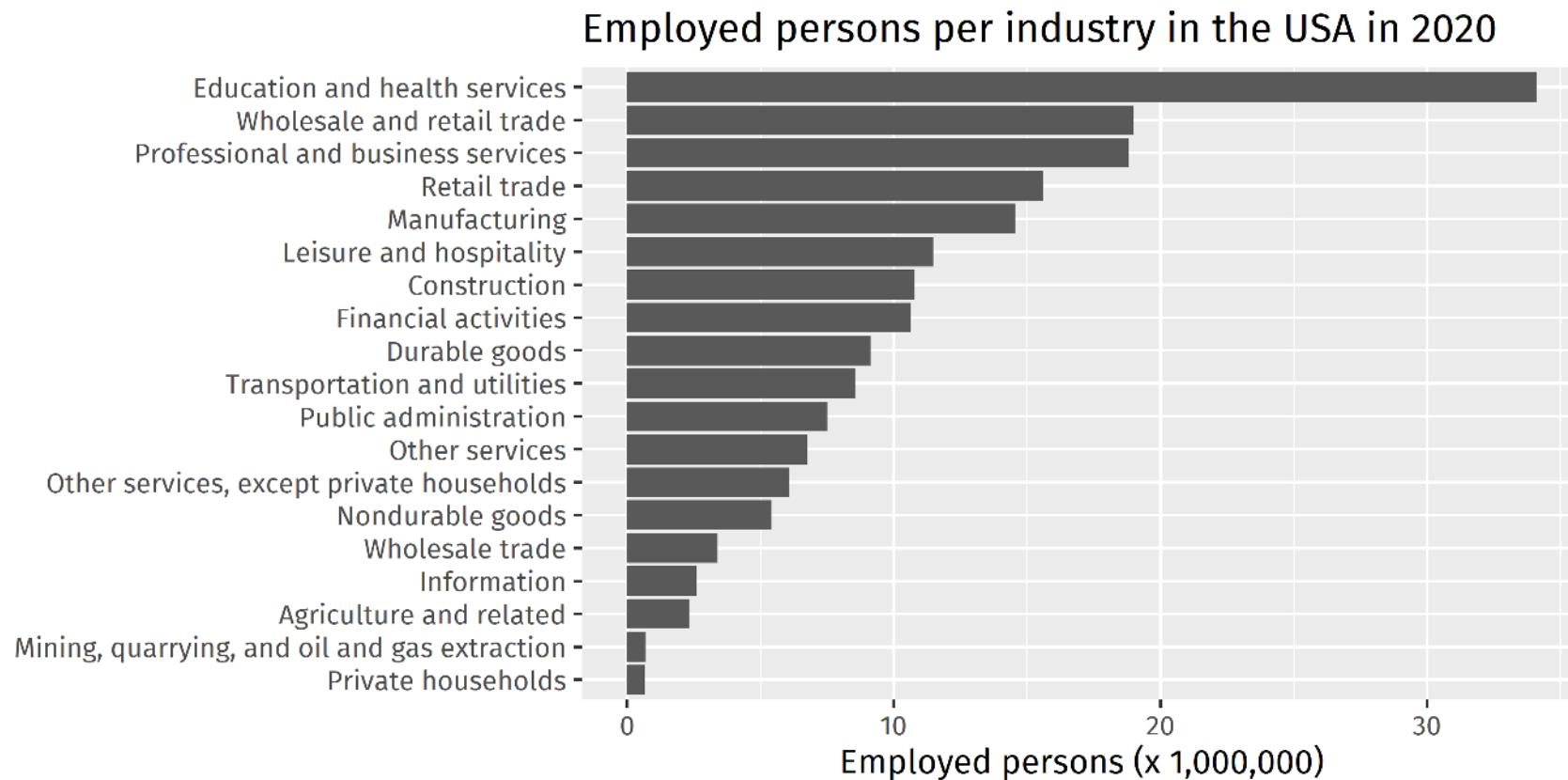


## fct\_reorder()

Default: alphabetic order

Industries ordered by number of employed persons

Code



## fct\_reorder()

Default: alphabetic order

Industries ordered by number of employed persons

Code

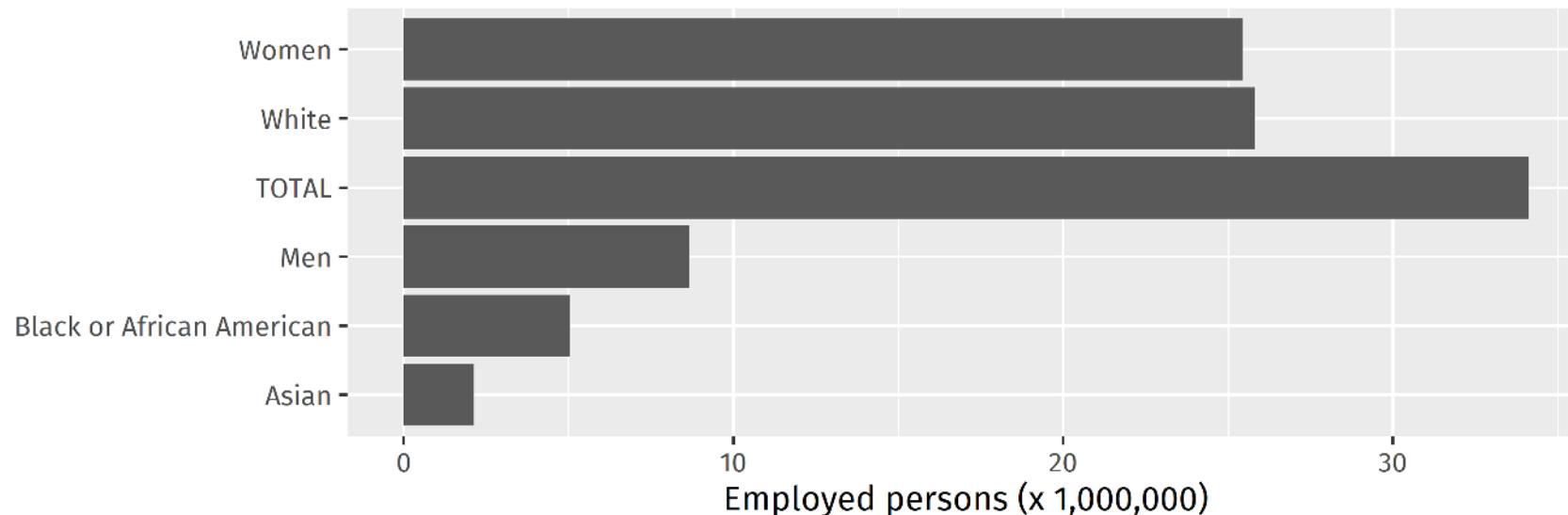
Syntax: `fct_reorder(<factor_column>, <numeric_column_to_order_by>)`

```
employed %>%
  drop_na() %>%
  filter(year == 2020, race_gender == "TOTAL") %>%
  distinct(industry, industry_total) %>%
  mutate(industry = fct_reorder(industry, industry_total)) %>%
  ggplot(aes(x = industry_total / 1000000, y = industry)) +
  geom_col() +
  labs(
    x = "Employed persons (x 1,000,000)", y = NULL,
    title = "Employed persons per industry in the USA in 2020"
  )
```

## fct\_inorder()

[Start](#)[fct\\_inorder\(\): order by first appearance](#)[fct\\_infreq\(\) + fct\\_rev\(\)](#)[Code](#)

Employed persons per race/gender in education and health services in the USA in 2020



```
employed$race_gender %>% unique() # order in the data
```

```
## [1] "TOTAL"          "Men"           "Women"         "  
## [4] "White"          "Black or African American" "Asian"
```

## fct\_inorder()

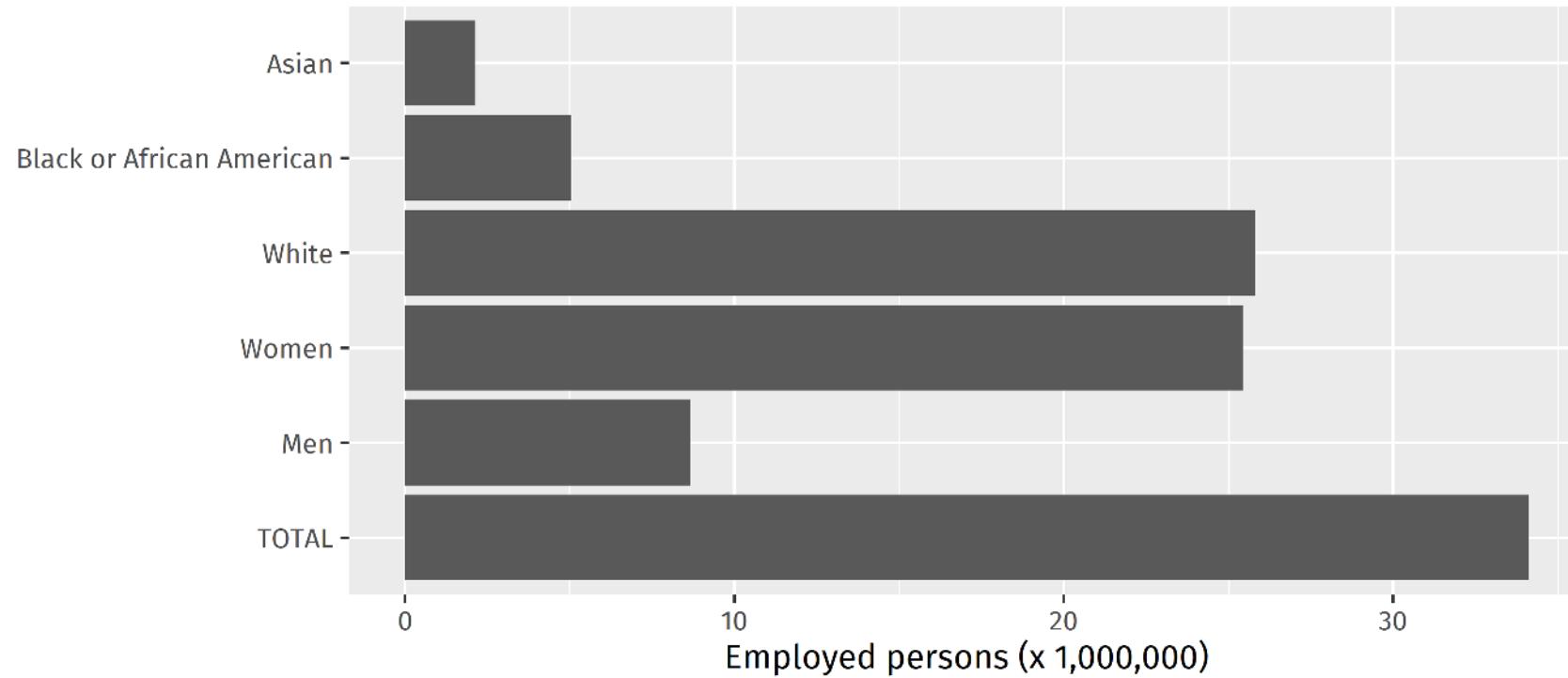
Start

fct\_inorder(): order by first appearance

fct\_infreq() + fct\_rev()

Code

Employed persons per race/gender in education and health services in the USA in 2020



## fct\_inorder()

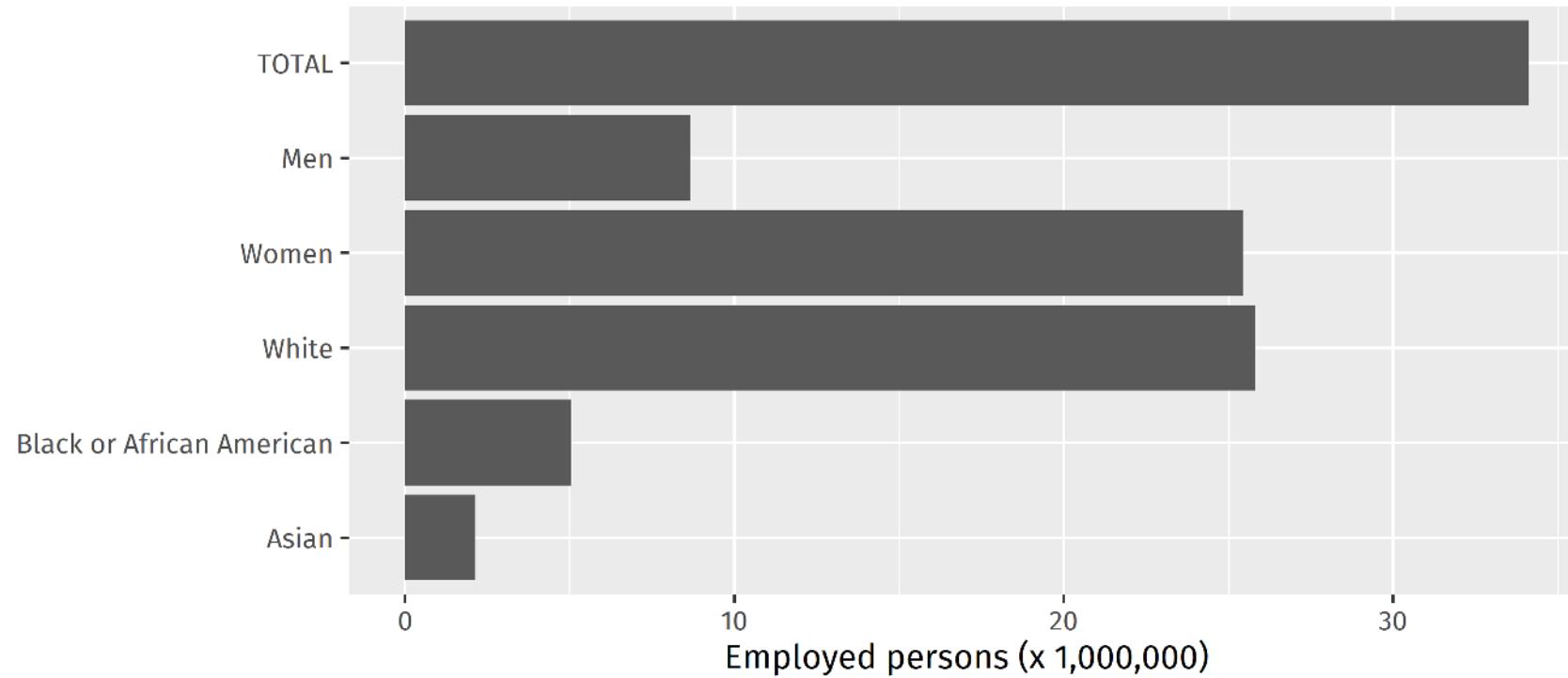
Start

fct\_inorder(): order by first appearance

fct\_infreq() + fct\_rev()

Code

Employed persons per race/gender in education and health services in the USA in 2020



## fct\_inorder()

Start

fct\_inorder(): order by first appearance

fct\_infreq() + fct\_rev()

Code

Syntax: `fct_inorder(<factor_column>)`

```
employed %>%  
  mutate(race_gender = fct_inorder(race_gender)) %>%  
  filter(year == 2020, industry == "Education and health services") %>%  
  drop_na() %>%  
  distinct(industry, race_gender, industry_total) %>%  
  ggplot(aes(x = industry_total / 1000000, y = fct_rev(race_gender))) +  
  geom_col() +  
  labs(  
    x = "Employed persons (x 1,000,000)", y = NULL,  
    title = "Employed persons per race/gender in education and health\\nservices in the USA in 2020"  
  )
```

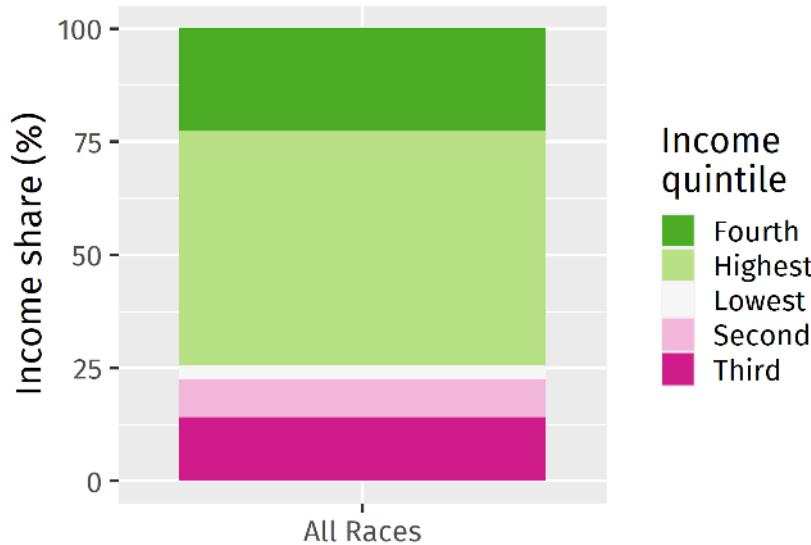
## fct\_relevel()

Category names have an inherent order

fct\_relevel()

Code

Income share by income quantile  
in the USA in 2019



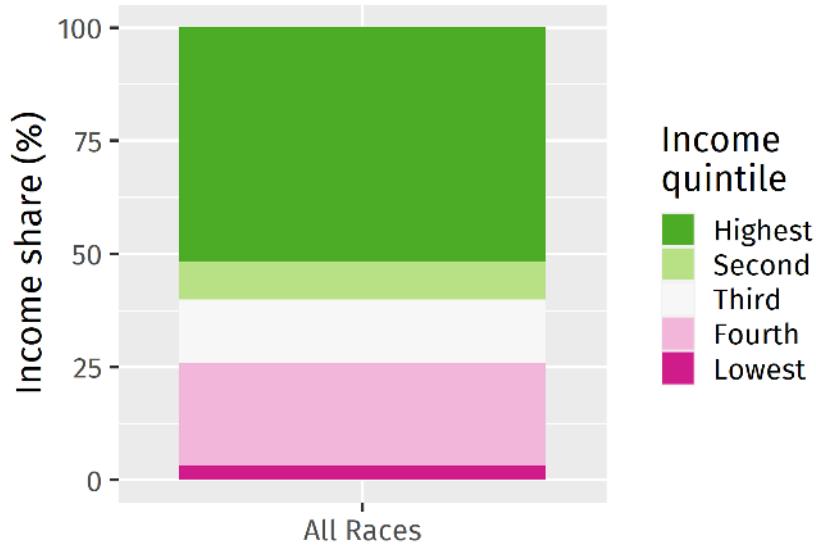
## fct\_relevel()

Category names have an inherent order

fct\_relevel()

Code

Income share by income quantile  
in the USA in 2019



## fct\_relevel()

Category names have an inherent order

fct\_relevel()

Code

Syntax: `fct_relevel(<factor_column>, <level_to_bring_to_first_pos>, <level_to_bring_to_second_pos>, ...)`

```
income %>%
  filter(year == 2019, race == "All Races", income_quintile != "Top 5%") %>%
  mutate(income_quintile = fct_relevel(income_quintile,
                                         "Highest", "Second", "Third", "Fourth", "Lowest"
  )) %>%
  ggplot(aes(x = race, y = income_share, fill = income_quintile)) +
  geom_col() +
  scale_fill_brewer(palette = "PiYG", direction = -1) +
  labs(x = NULL, y = "Income share (%)", fill = "Income\\nquintile",
       title = "Income share by income quantile\\nin the USA in 2019")
```

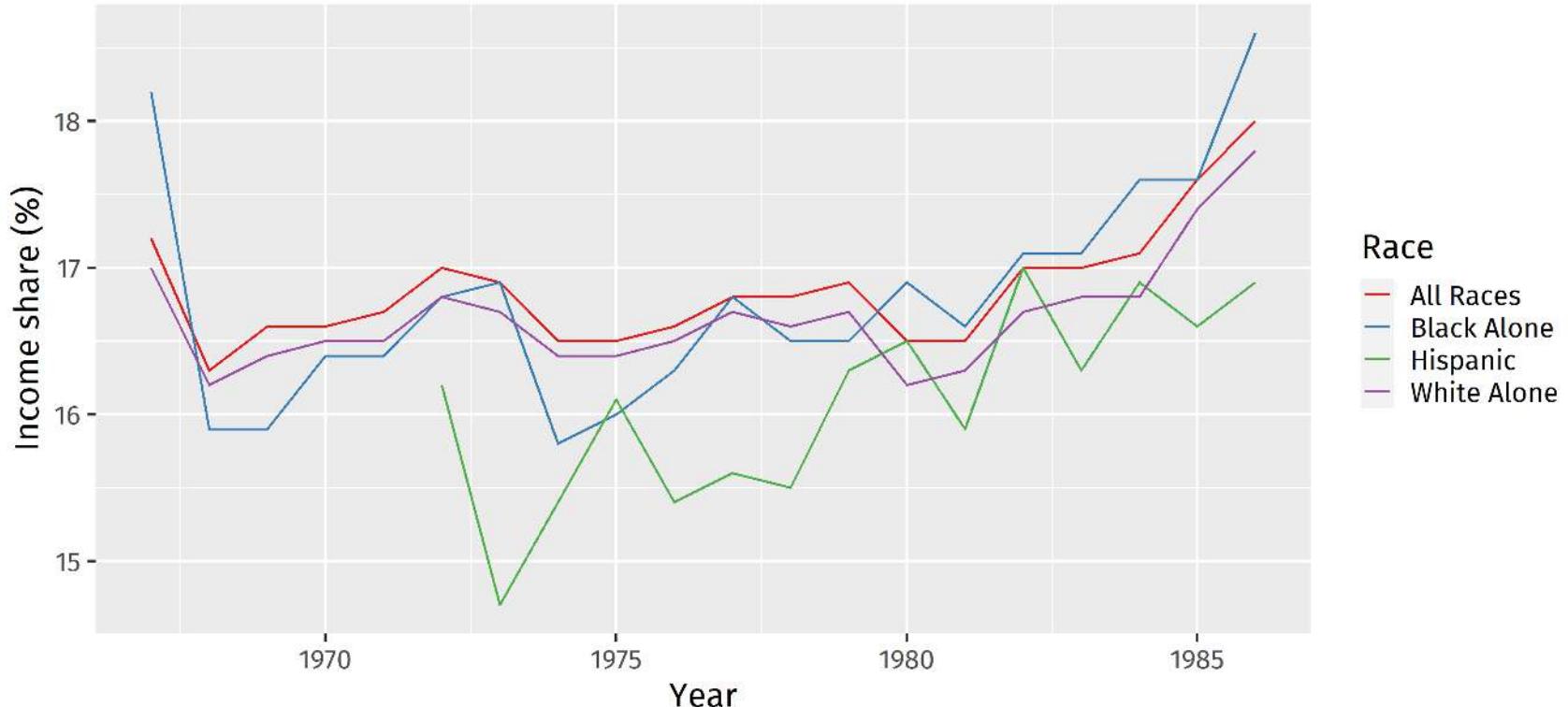
## fct\_reorder2()

Hard to read

fct\_reorder2()

Code

Income share of the top 5% incomes by race in the USA over time



Goal: Order `race` by `income_share` in 1986 (last year)

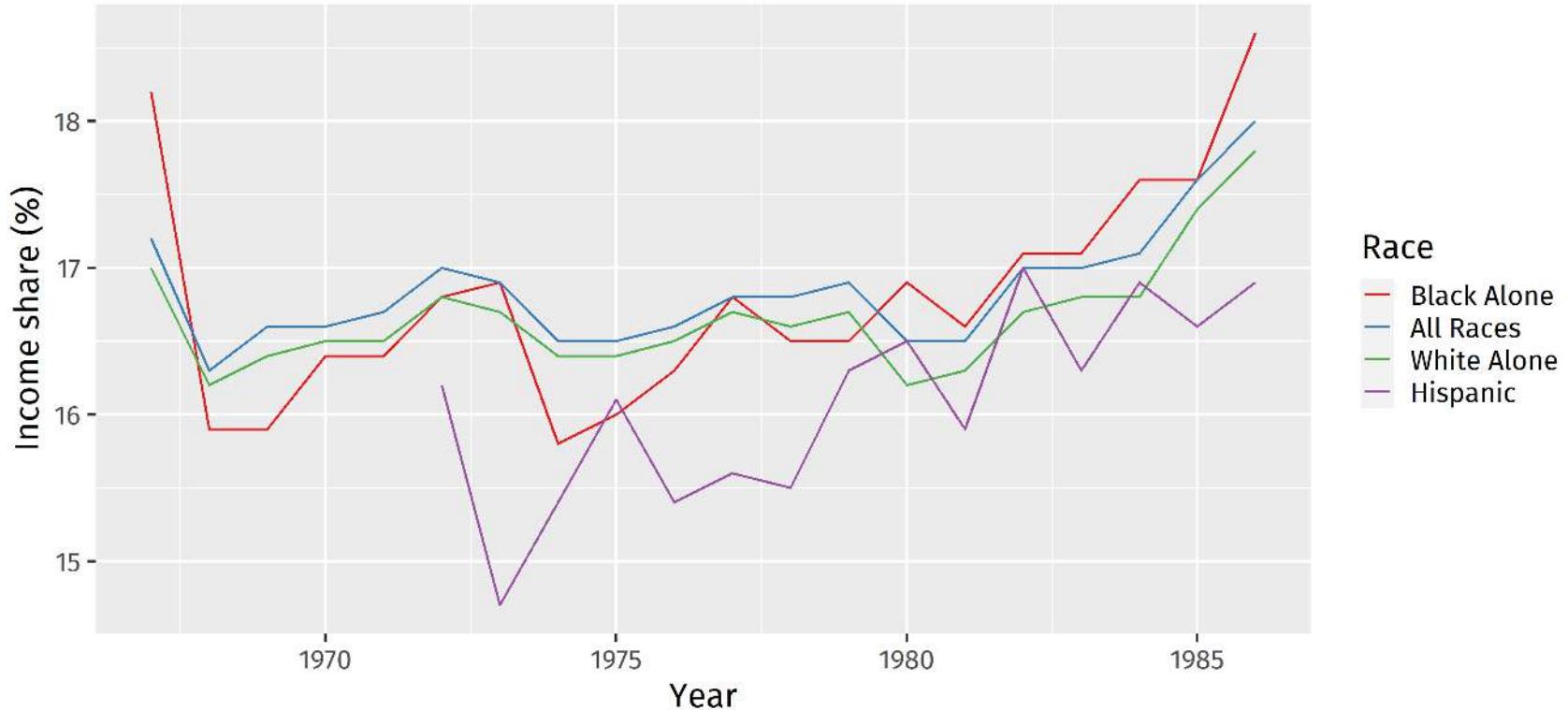
## fct\_reorder2()

Hard to read

fct\_reorder2()

Code

Income share of the top 5% incomes by race in the USA over time



Race

- Black Alone
- All Races
- White Alone
- Hispanic

## fct\_reorder2()

Hard to read

fct\_reorder2()

Code

Syntax: fct\_reorder2(<factor\_column>, <numeric\_column>, <numeric\_column>)

```
income %>%
  filter(income_quintile == "Top 5%", year <= 1986) %>%
  filter(race %in% c("All Races", "Black Alone", "Hispanic", "White Alone")) %>%
  mutate(race = fct_reorder2(race, year, income_share)) %>%
  ggplot(aes(x = year, y = income_share, color = race)) +
  geom_line(size = 0.7) +
  scale_color_brewer(palette = "Set1") +
  labs(
    x = "Year", y = "Income share (%)", color = "Race",
    title = "Income share of the top 5% incomes by race in the USA over time"
  )
```

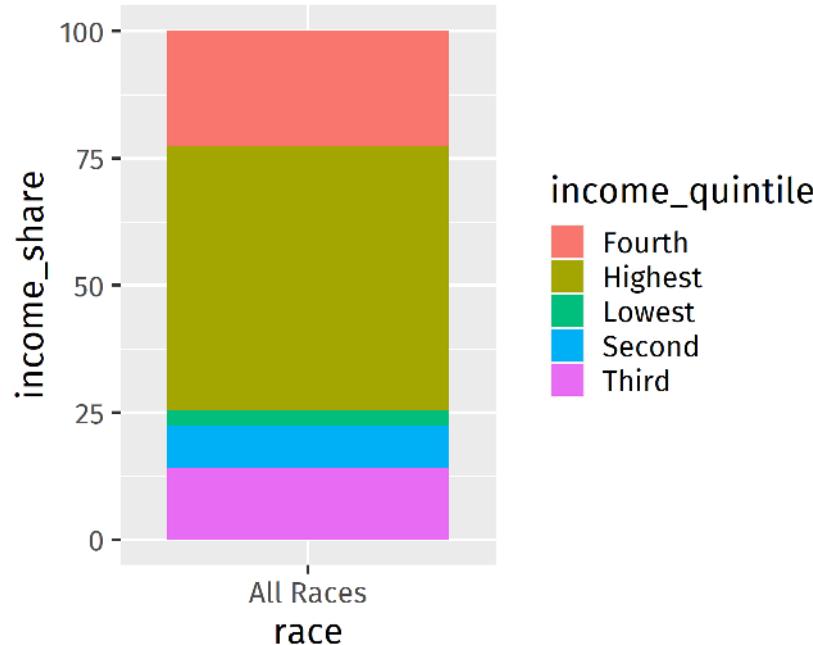
## fct\_collapse()

We want to display "Highest" vs. others combined

fct\_collapse()

fct\_collapse() + fct\_rev()

Code



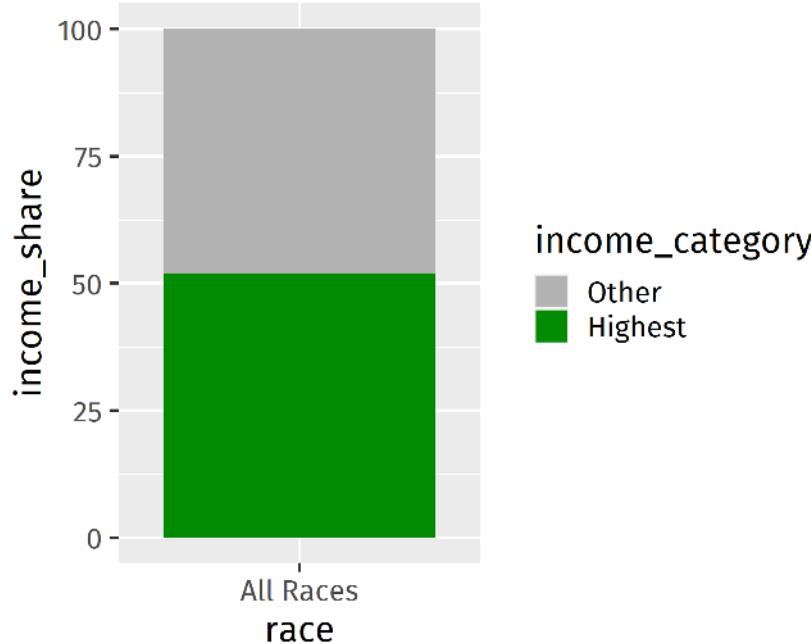
## fct\_collapse()

We want to display "Highest" vs. others combined

fct\_collapse()

fct\_collapse() + fct\_rev()

Code



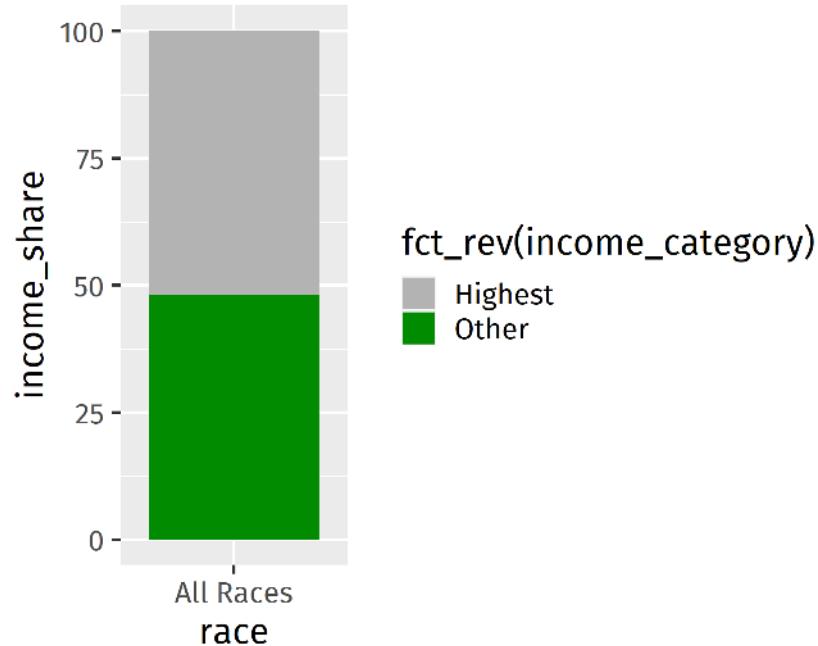
## fct\_collapse()

We want to display "Highest" vs. others combined

fct\_collapse()

fct\_collapse() + fct\_rev()

Code



## fct\_collapse()

We want to display "Highest" vs. others combined

fct\_collapse()

fct\_collapse() + fct\_rev()

Code

Syntax: `fct_collapse(<factor_column>, <new_level>, c(<old_level1>, <old_level2>, ...))`

```
income %>%
  filter(year == 2019, race == "All Races", income_quintile != "Top 5%") %>%
  mutate(income_category = fct_collapse(
    income_quintile,
    "Other" = c("Second", "Third", "Fourth", "Lowest")
  )) %>%
  ggplot(aes(x = race, y = income_share, fill = fct_rev(income_category))) +
  geom_col() +
  scale_fill_manual(values = c("gray70", "green4"))
```

# Dates and times

The Tidyverse package `lubridate` provides functions to work with dates and times. Since `lubridate` is not a core Tidyverse packages, we have to load it separately.

```
library(lubridate)
today() # date

## [1] "2021-04-02"

today() %>% class()

## [1] "Date"

now() # date-time

## [1] "2021-04-02 15:21:09 CEST"

now() %>% class()

## [1] "POSIXct" "POSIXt"

today() %>% typeof()

## [1] "double"

now() %>% typeof()

## [1] "double"
```



# Creating dates from character strings

```
tibble(x = c("2021-01-31", "2021-02-05", "2021-03-31")) %>%
  mutate(dte = ymd(x))
```

```
## # A tibble: 3 x 2
##   x           dte
##   <chr>       <date>
## 1 2021-01-31 2021-01-31
## 2 2021-02-05 2021-02-05
## 3 2021-03-31 2021-03-31
```

```
tibble(x = c("31.01.21", "05.02.21", "31.03.21")) %>%
  mutate(dte = dmy(x))
```

```
## # A tibble: 3 x 2
##   x           dte
##   <chr>       <date>
## 1 31.01.21 2021-01-31
## 2 05.02.21 2021-02-05
## 3 31.03.21 2021-03-31
```

```
tibble(x = c("January 31 21", "Feb 05 21", "Mar 31 21")) %>%
  mutate(dte = mdy(x))
```

```
## # A tibble: 3 x 2
##   x           dte
##   <chr>       <date>
## 1 January 31 21 2021-01-31
## 2 Feb 05 21     2021-02-05
## 3 Mar 31 21    2021-03-31
```

# Extracting date/time components

```
tibble(x = c("2021-01-31 11:59:59", "2021-02-05 02:11:20", "2021-03-31 00:00:00")) %>%
  transmute(dte_tme = ymd_hms(x)) %>%
  mutate(
    year = year(dte_tme),
    month = month(dte_tme),
    day = day(dte_tme),
    hour = hour(dte_tme),
    minute = minute(dte_tme),
    second = second(dte_tme),
  )
```

```
## # A tibble: 3 x 7
##   dte_tme           year month   day hour minute second
##   <dttm>       <dbl> <dbl> <int> <int> <int>  <dbl>
## 1 2021-01-31 11:59:59  2021     1     31     11      59      59
## 2 2021-02-05 02:11:20  2021     2      5      2     11      20
## 3 2021-03-31 00:00:00  2021     3     31      0      0       0
```

# Arithmetic operations on dates

Differences between two dates/times are of class `difftime`:

```
dft <- ymd(20210301) - ymd(20210201)  
dft
```

```
## Time difference of 28 days
```

```
as.double(dft, units = "weeks")
```

```
## [1] 4
```

```
class(dft)
```

```
## [1] "difftime"
```

`lubridate` implements three classes to perform arithmetic operations on **time spans**, i.e. addition, subtraction, and division:

- `Duration`: exact number of seconds
- `Period`: human units like weeks and months
- `Interval`: represent a starting and ending point

# Durations

```
ymd("2021-03-27") + ddays(1)
```

```
## [1] "2021-03-28"
```

```
ymd("2021-03-27") + dyears(1)
```

```
## [1] "2022-03-27 06:00:00 UTC"
```

Why 06:00:00?

```
dyears(1) / ddays(1)
```

```
## [1] 365.25
```

# Periods

```
ymd("2021-03-27") + years(1)

## [1] "2022-03-27"

ymd("2021-02-01") + dmonths(1) # duration

## [1] "2021-03-03 10:30:00 UTC"

ymd("2021-02-01") + months(1) # period

## [1] "2021-03-01"

dmonths(1) / ddays(1)

## [1] 30.4375

months(1) / days(1)

## [1] 30.4375
```

Why are `dmonths(1)` and `months(1)` seemingly equal?

# Intervals

```
first_of_march <- ymd("2021-02-01") + months(1)
first_of_march

## [1] "2021-03-01"

(ymd("2021-02-01") %--% first_of_march) / dmonths(1)

## [1] 0.9199179

(ymd("2021-02-01") %--% first_of_march) / months(1)

## [1] 1
```

# Functions

# COVID-19 data for Germany

How did the proportion of different age groups to the number of infected persons in German states change over the period of the pandemic?

---

COVID-19 data

Data description

Plot

Code

---

```
covid <- read_rds(here::here("data/RKI_COVID19.rds"))
glimpse(covid)

## # Rows: 1,509,819
## # Columns: 10
## $ row_id      <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ~
## $ ref_date    <date> 2021-01-27, 2021-01-28, 2021-02-05, 2021-02-16, 2021-02-21, 2021-~
## $ federal_state <fct> Schleswig-Holstein, Schleswig-Holstein, Schleswig-Holstein, Schlesw~
## $ district     <chr> "SK Flensburg", "SK Flensburg", "SK Flensburg", "SK Flensburg", "S~
## $ age_group    <chr> "A35-A59", "A35-A59", "A35-A59", "A35-A59", "A35-A59", "A35-A59", ~
## $ sex          <fct> unknown, unknown, unknown, unknown, unknown, f, f, f, f, f, ~
## $ cases         <int> 2, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
## $ deaths        <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
## $ convalescents <int> 2, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, ~
## $ data_date     <date> 2021-03-28, 2021-03-28, 2021-03-28, 2021-03-28, 2021-03-28, 2021-~
```

Source: [Robert Koch Institute](#)

# COVID-19 data for Germany

How did the proportion of different age groups to the number of infected persons in German states change over the period of the pandemic?

COVID-19 data

Data description

Plot

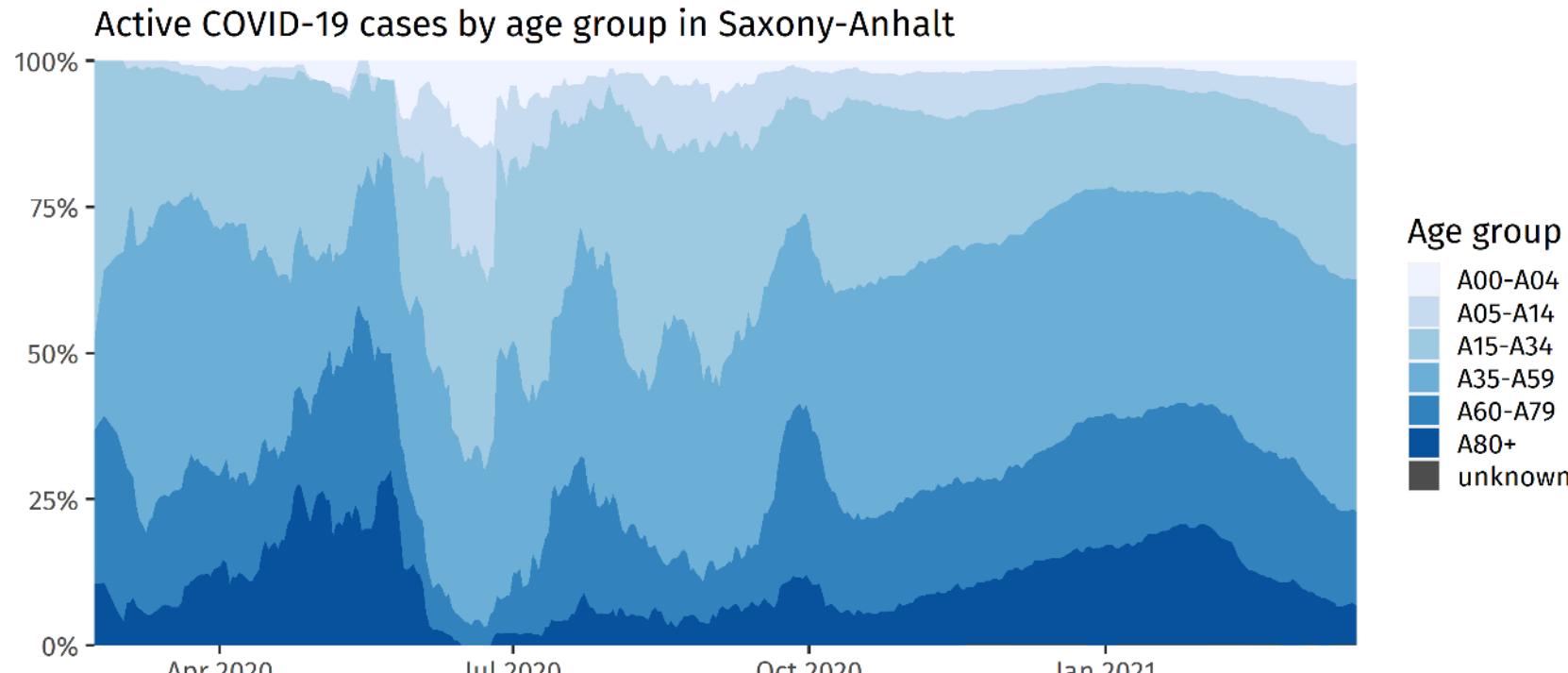
Code

Column	Description
row_id	Row number
ref_date	Date of infection or, if this is not known, the date of notification.
federal_state	Federal state (German: Bundesland)
district	Administrative district (German: Landkreis)
age_group	Age group
sex	Sex
cases	Number of active cases
deaths	Number of new deaths
convalescents	Number of new recovered
data_date	Date of last update (German: Datenstand)

Source: [Dataset description \(in German\)](#)

# COVID-19 data for Germany

How did the proportion of different age groups to the number of infected persons in German states change over the period of the pandemic?

[COVID-19 data](#)[Data description](#)[Plot](#)[Code](#)

# COVID-19 data for Germany

How did the proportion of different age groups to the number of infected persons in German states change over the period of the pandemic?

[COVID-19 data](#)[Data description](#)[Plot](#)[Code](#)

```
covid %>%
  filter(federal_state == "Saxony-Anhalt") %>%
  complete(ref_date, age_group, fill = list(cases = 0)) %>%
  group_by(ref_date, age_group) %>%
  summarize(n_cases = sum(cases)) %>% # average over districts and sexes
  group_by(age_group) %>%
  arrange(ref_date) %>%
  mutate(rolling_mean = RcppRoll::roll_mean(n_cases, n = 14, fill = NA)) %>%
  ungroup() %>% drop_na() %>%
  ggplot(aes(x = ref_date, y = rolling_mean, fill = age_group)) +
  geom_area(position = "fill") +
  scale_fill_manual(values = c(RColorBrewer::brewer.pal(6, "Blues"), "gray30")) +
  labs(
    x = NULL, y = NULL, fill = "Age group",
    title = "Active COVID-19 cases by age group in Saxony-Anhalt",
    caption = paste0("Source: Robert Koch Institute (", format(covid$data_date[1], "%d.%m.%y"), ")")
  ) +
  coord_cartesian(expand = FALSE) +
  scale_y_continuous(labels = scales::percent) +
  theme(plot.title = element_text(size = rel(1.05)))
```

We want to create this plot also for the other 15 states. Is there an a better way than copy-and-pasting?

# Why functions?

From [R4DS](#):

- A function with an expressive name makes your code easier to understand.
- If you want to make changes to the code, you only need to update in one place instead of many.
- You eliminate the chance of making inadvertently mistakes because of copy and paste.

In the long run, it is advisable to make your code available in functions (*within packages*) so that future you and others can use your code.

# Writing functions

## Function scaffolding:

```
plot_covid_cases_by_age_groups <- function(state) {  
  # code from an earlier slide goes here  
}
```

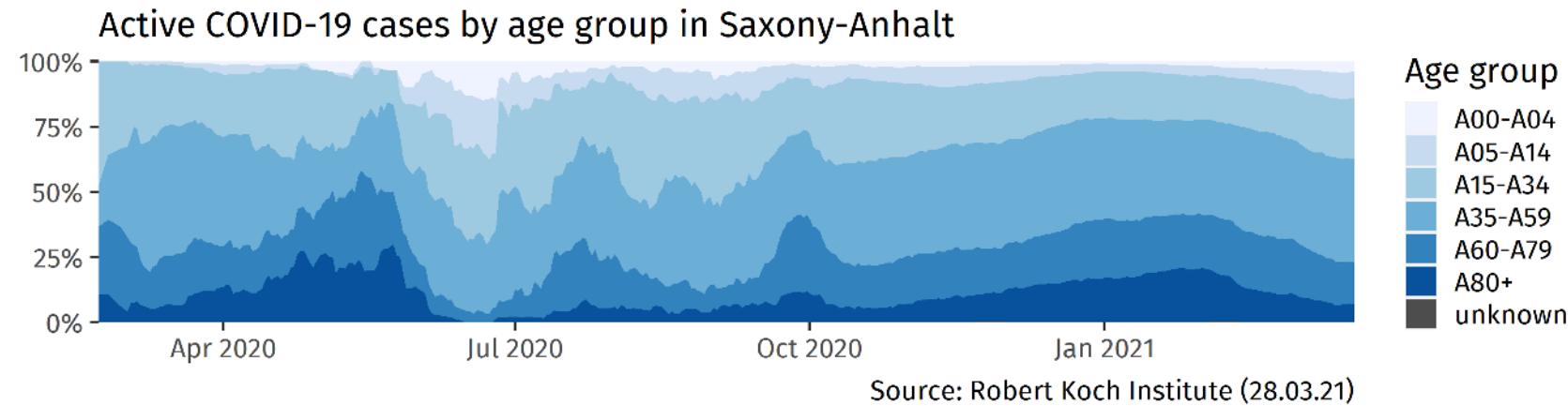
- `plot_covid_cases_by_age_groups` is the function **name**
  - Try to find informative, expressive names, preferably a verb
- Since a function is an object, we use the `<-` operator
- `state` is the only **argument** of the function
  - If we had more than one argument, we would write `function(state, arg2, arg3)`
  - We can specify default values, e.g. `function(state = "Berlin")`. If an argument does not have a default value, its value must be given in the function call.
- The actual code is placed in the **body** of the function enclosed by `{}`. The opening curly brackets `{` must follow immediately after `function()`.
- Use `return(some_object)` to return the object `some_object` (early). If `return()` is not used, the result of the function's last command will be returned.

## plot\_covid\_cases\_by\_age\_groups()

```
plot_covid_cases_by_age_groups <- function(state) {
  covid %>%
    filter(federal_state == state) %>%
    complete(ref_date, age_group, fill = list(cases = 0)) %>%
    group_by(ref_date, age_group) %>%
    summarize(n_cases = sum(cases)) %>% # average over districts and sexes
    group_by(age_group) %>%
    arrange(ref_date) %>%
    mutate(rolling_mean = RcppRoll::roll_mean(n_cases, n = 14, fill = NA)) %>%
    ungroup() %>% drop_na() %>%
    ggplot(aes(x = ref_date, y = rolling_mean, fill = age_group)) +
    geom_area(position = "fill") +
    scale_fill_manual(values = c(RColorBrewer::brewer.pal(6, "Blues"), "gray30")) +
    labs(
      x = NULL, y = NULL, fill = "Age group",
      title = paste("Active COVID-19 cases by age group in", state),
      caption = paste0("Source: Robert Koch Institute (", format(covid$data_date[1], "%d.%m.%y"), ")")
    ) +
    coord_cartesian(expand = FALSE) +
    scale_y_continuous(labels = scales::percent) +
    theme(plot.title = element_text(size = rel(1.05)))
}
```

# Calling the function

```
plot_covid_cases_by_age_groups("Saxony-Anhalt")
```



```
plot_covid_cases_by_age_groups("Bavaria")
```

# More function arguments

Let's improve the function by adding the moving average window length as the second function argument.

---

Code      w = 14 (default)      w = 7      w = 50

---

```
plot_covid_cases_by_age_groups <- function(state, w = 14) {  
  covid %>%  
    filter(federal_state == state) %>%  
    complete(ref_date, age_group, fill = list(cases = 0)) %>%  
    group_by(ref_date, age_group) %>%  
    summarize(n_cases = sum(cases)) %>% # average over districts and sexes  
    group_by(age_group) %>%  
    arrange(ref_date) %>%  
    mutate(rolling_mean = RcppRoll::roll_mean(n_cases, n = w, fill = NA)) %>%  
    ungroup() %>% drop_na() %>%  
    ggplot(aes(x = ref_date, y = rolling_mean, fill = age_group)) +  
    geom_area(position = "fill") +  
    scale_fill_manual(values = c(RColorBrewer::brewer.pal(6, "Blues"), "gray30")) +  
    labs(  
      x = NULL, y = NULL, fill = "Age group",  
      title = paste("Active COVID-19 cases by age group in", state),  
      caption = paste0("Source: Robert Koch Institute (", format(covid$data_date[1], "%d.%m.%y"), ")"),  
      subtitle = paste("Moving average window length =", w, "days")  
    ) +  
    coord_cartesian(expand = FALSE) +  
    scale_y_continuous(labels = scales::percent) +  
    theme(plot.title = element_text(size = rel(1.05)))  
}
```

# More function arguments

Let's improve the function by adding the moving average window length as the second function argument.

Code

w = 14 (default)

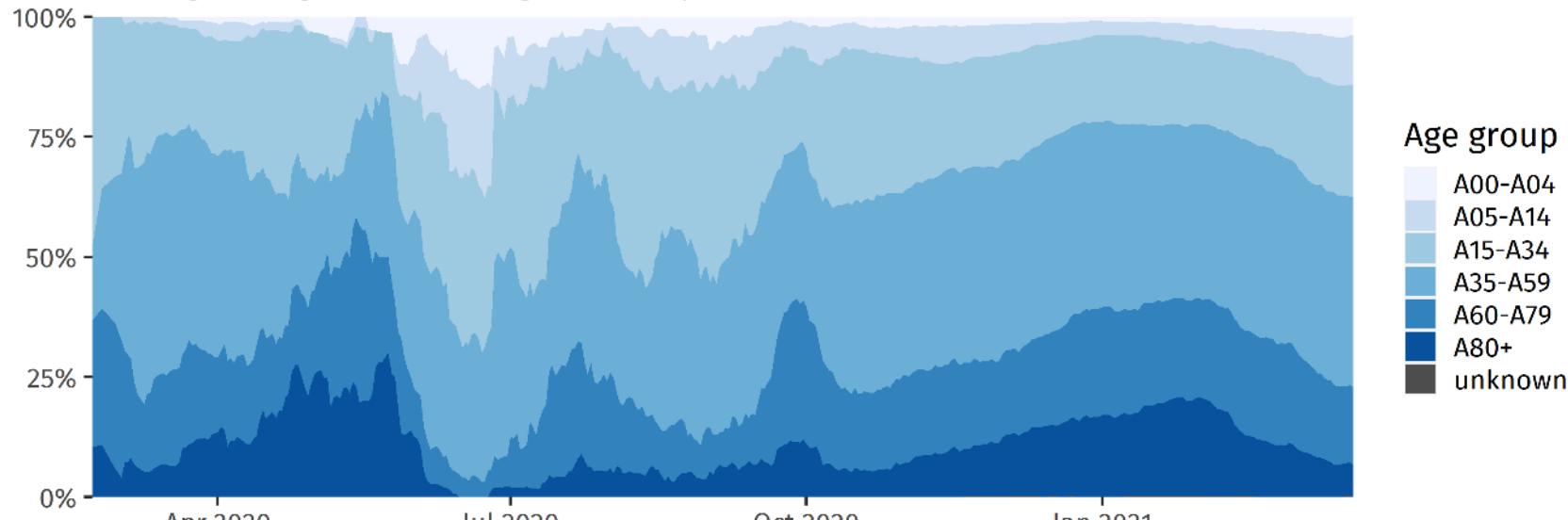
w = 7

w = 50

```
plot_covid_cases_by_age_groups("Saxony-Anhalt")
```

Active COVID-19 cases by age group in Saxony-Anhalt

Moving average window length = 14 days



Source: Robert Koch Institute (28.03.21)

# More function arguments

Let's improve the function by adding the moving average window length as the second function argument.

Code

w = 14 (default)

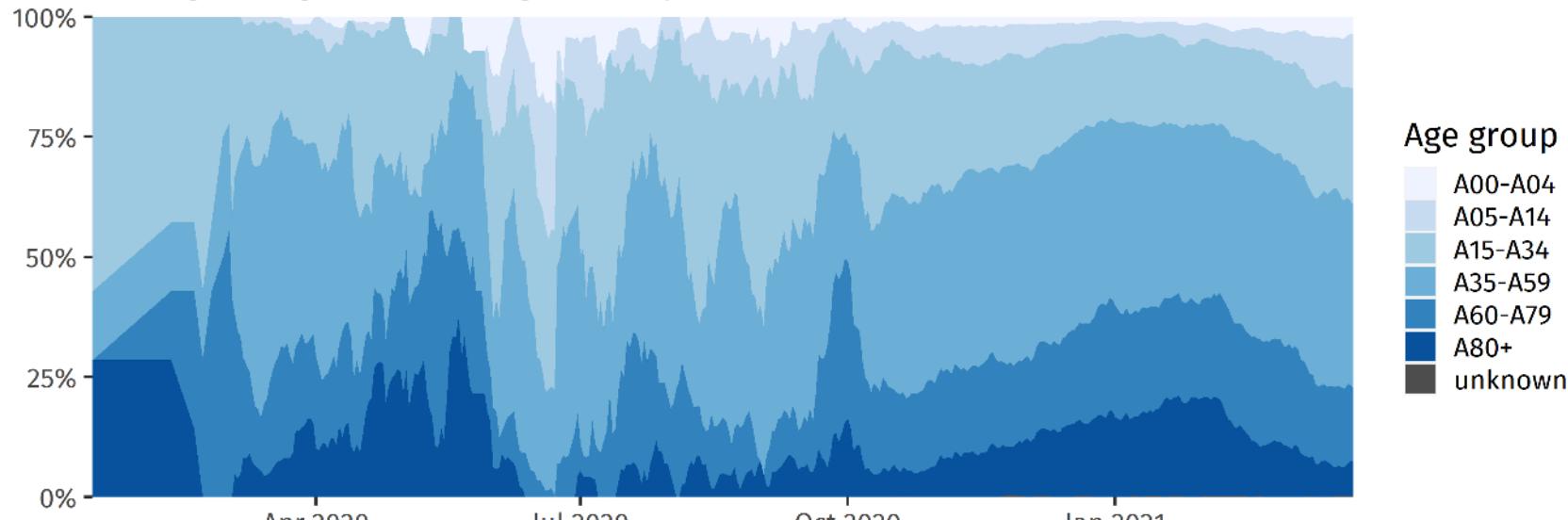
w = 7

w = 50

```
plot_covid_cases_by_age_groups("Saxony-Anhalt", w = 7)
```

Active COVID-19 cases by age group in Saxony-Anhalt

Moving average window length = 7 days



Source: Robert Koch Institute (28.03.21)

# More function arguments

Let's improve the function by adding the moving average window length as the second function argument.

Code

w = 14 (default)

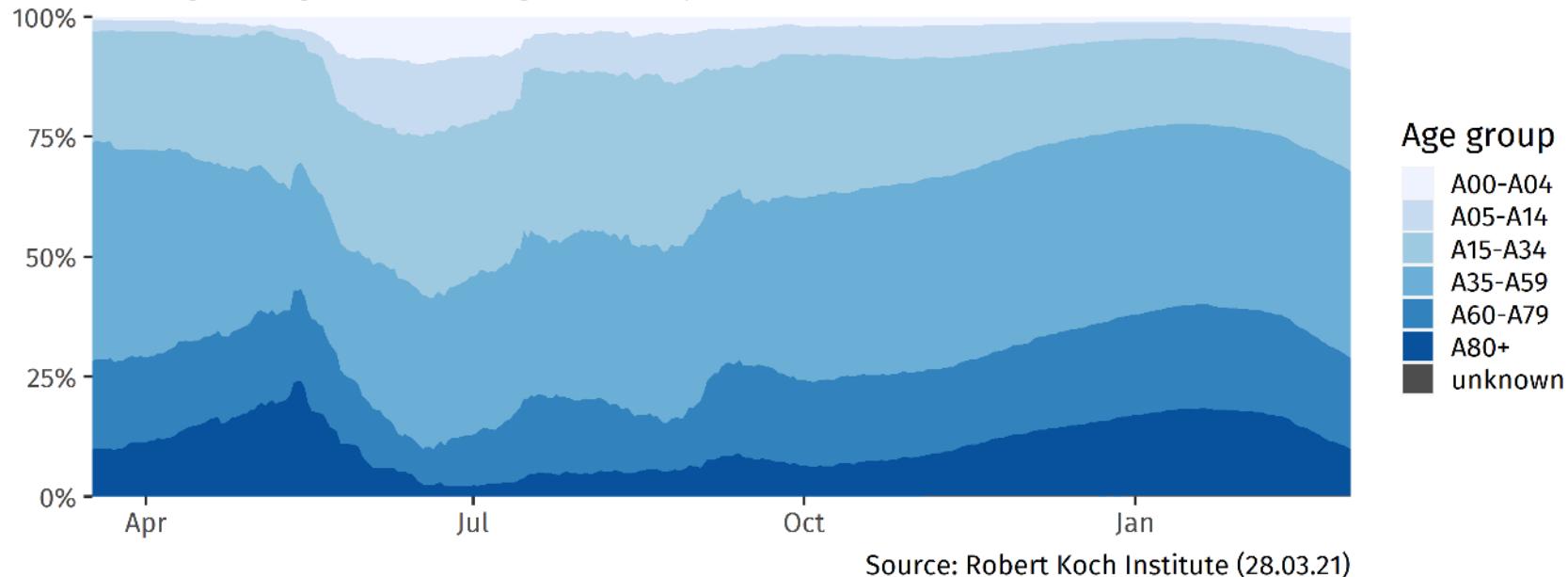
w = 7

w = 50

```
plot_covid_cases_by_age_groups("Saxony-Anhalt", w = 50)
```

Active COVID-19 cases by age group in Saxony-Anhalt

Moving average window length = 50 days



# Return value

Recall that by default, the result of the function's last evaluated command will be returned. We can use `return(some_object)` to return the object `some_object` early.

---

Example without `return()`

---

```
convert_cm_to_inch <- function(cm) {  
  0.393701 * cm  
  42  
}
```

Example with `return()`

---

```
convert_cm_to_inch(10)
```

```
## [1] 42
```

```
convert_cm_to_inch(100)
```

```
## [1] 42
```

# Return value

Recall that by default, the result of the function's last evaluated command will be returned. We can use `return(some_object)` to return the object `some_object` early.

Example without `return()`

```
convert_cm_to_inch <- function(cm) {  
  return(0.393701 * cm)  
  42  
}
```

Example with `return()`

```
convert_cm_to_inch(10)
```

```
## [1] 3.93701
```

```
convert_cm_to_inch(100)
```

```
## [1] 39.3701
```

# Naming things

In `R`, **functions are objects**. Objects must have **syntactically valid** names:

- Names can only consist of letters, digits, `_` and `.`
- Names must begin with a letter or with `.` not followed by a digit
  - Example: `.7up` is not valid, but `.sevenup` is
- Names must not be one of the reserved words, e.g. `if`, `else`, `for`, `TRUE`, `NULL`...

For names consisting of multiple words, it is recommended to use **snake\_case**  opposed to **camelCase** . You are free to use `,`, but please mind consistency.

```
# good
crawl_corona_data
crawl_covid19_data

# bad
crawl_data_on_sars_cov_2 # a bit too long
ccd # too short and uninformative
CrawlCoronaData # rather start with lowercase letter
crawlcoronadata # rather separated words
data # don't overwrite existing popular/base functions
```

## Documentation:

- `?make.names` describes all requirements for syntactically valid names
- `?reserved` lists all reserved words in R's parser.

Further reading: [The tidyverse style guide](#)

# Lexical scoping

**Scoping** describes in which order `R` searches for objects.

```
a <- 3
f <- function() {
  a <- 5
  b <- 2
  a^b
}
```

⌚ "What is the result of running `f()`?"

```
f()
```

```
## [1] 25
```

⌚ "Why is the result not 9?"

First, `R` searches for object names in the environment of the called function.

If the name does not exist in this environment, `R` searches in the next higher environment level.

# Lexical scoping

```
a <- 3
f <- function() {
  a <- 5
  b <- 2
  a^b
}
f()
```

```
## [1] 25
```

```
a <- 3
g <- function() {
  b <- 2
  a^b
}
g()
```

```
## [1] 9
```

# Lexical scoping

Each function call starts with a new environment:

```
h <- function() {  
  if (!exists("x")) {  
    x <- 1  
  } else {  
    x <- x + 1  
  }  
  x  
}
```

```
h()
```

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

```
h()
```

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

```
x
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

```
x <- 5  
h()
```

```
## [1] 6
```

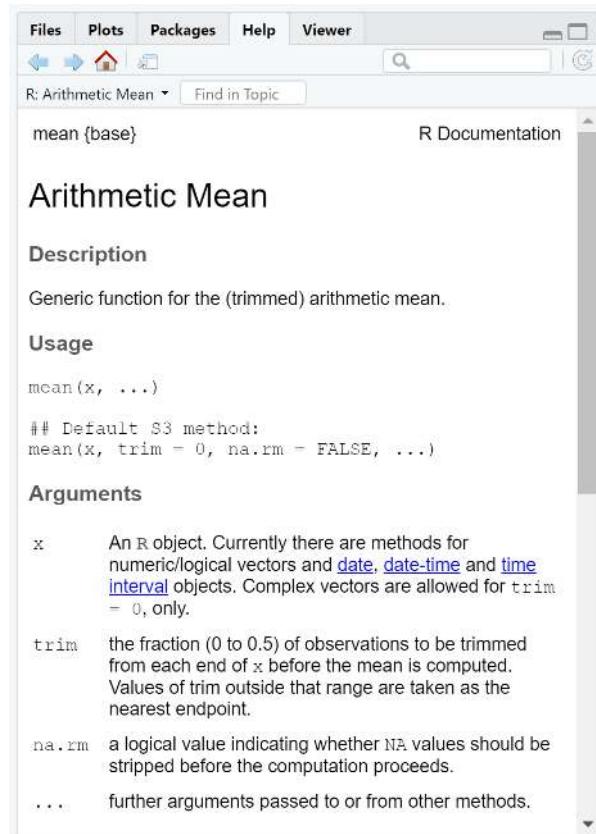
```
h()
```

```
## [1] 6
```

# Calling functions

Basic function call scheme:

```
some_function(arg_1 = val_1,  
              arg_2 = val_2,  
              ...)
```



Example: the `mean()` function:

- `x` is the only mandatory argument
- arguments `trim` and `na.rm` have default values

```
x <- 1:10  
mean(x) # trim = 0 and na.rm = FALSE  
  
## [1] 5.5  
  
x <- c(1:10, NA)  
mean(x)  
  
## [1] NA  
  
mean(x, na.rm = TRUE) # NA's will be ignored  
  
## [1] 5.5  
  
mean(x, TRUE) # match unnamed args to their position  
  
## Error in mean.default(x, TRUE): 'trim' must be numeric of length one
```

# Iteration

# Define the task

**Goal:** Create and save a stacked area chart on Covid-19 cases by age groups for each of the 16 German states.

So far, we can do this:

```
plot_covid_cases_by_age_groups("Baden-Wuerttemberg")
plot_covid_cases_by_age_groups("Bavaria")
plot_covid_cases_by_age_groups("Berlin")
plot_covid_cases_by_age_groups("Brandenburg")
# ...
```

# Iteration

How can we apply our `plot_covid_cases_by_age_groups()` function to each federal state?

1. Option: using a `for` loop

```
states <- sort(unique(covid$federal_states))
plots <- vector("list", length = length(states))
for(i in seq_along(states)) {
  plots[[i]] <- plot_covid_cases_by_age_groups(states[i])
}
```

2. Option: using **functionals** (recommended)

# Functionals

Suppose we have the ratings of four persons for three TV series stored in a data frame:

```
ratings <- tibble(  
  breaking_bad = c(1, 9, 4, 8),  
  the_crown = c(4, 2, 0, 5),  
  vikings = c(9, 9, 4, 1)  
)
```

We can calculate the average rating for each TV series with `map()`:

```
map(ratings, mean)
```

```
## $breaking_bad  
## [1] 5.5  
##  
## $the_crown  
## [1] 2.75  
##  
## $vikings  
## [1] 5.75
```

Return the result as double vector:

```
map_dbl(ratings, mean)
```

```
## breaking_bad      the_crown      vikings  
##      5.50          2.75          5.75
```

Return the result as character vector:

```
map_chr(ratings, mean)
```

```
## breaking_bad      the_crown      vikings  
## "5.500000"      "2.750000"      "5.750000"
```

# The `purrr` package

The `purrr` package is part of the core Tidyverse and provides the `map` function family. A `map` function applies a given function to each element of a vector.

`map*` () takes as arguments

1. a vector and
2. a function.

It return a new vector of the **same length** as the input. The **type** of the vector is specified by the **suffix** of the `map*` () function.

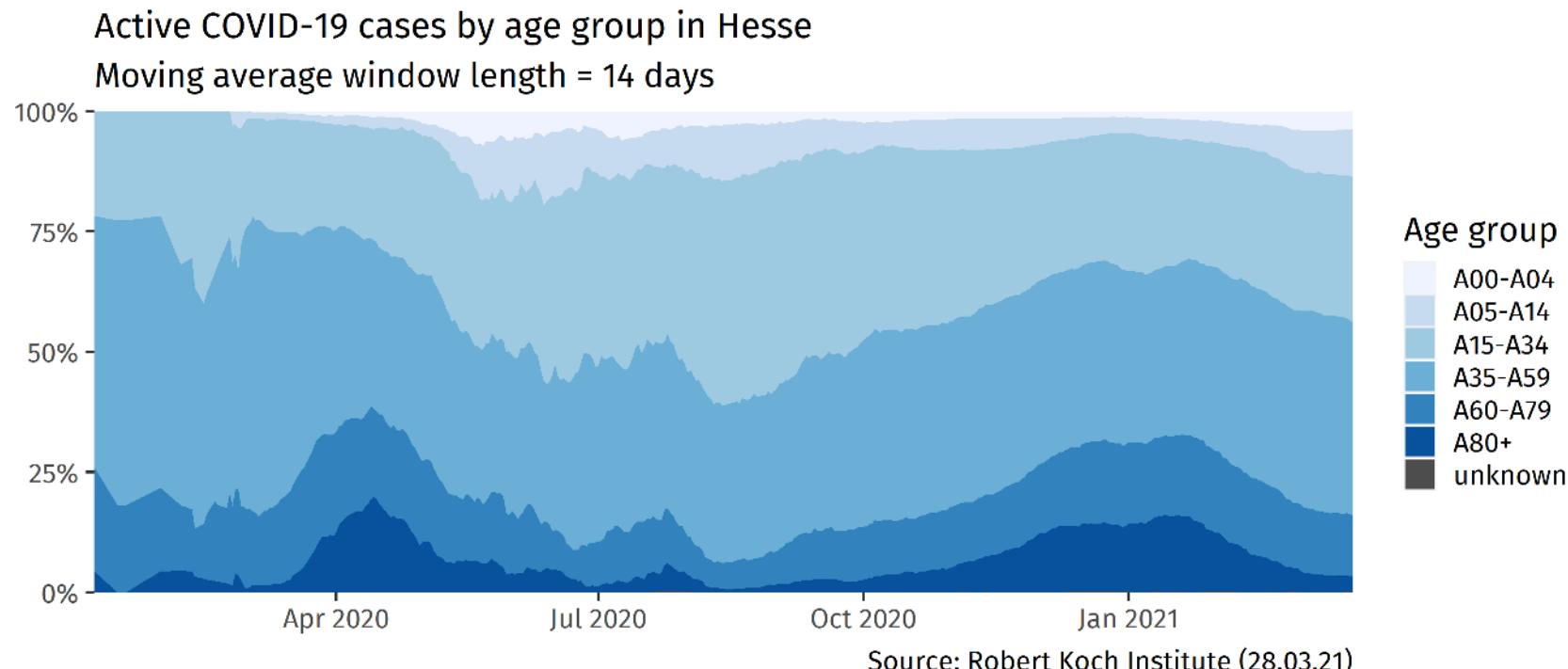
- `map()` returns a list
- `map_lgl()` returns a logical vector
- `map_int()` returns an integer vector
- `map_dbl()` returns a double vector
- `map_chr()` returns a character vector
- `map_dfr()` returns a data frame by row binding
- `map_dfc()` returns a data frame by column binding
- ...



# Create a plot for each federal state

Map the function `plot_covid_cases_by_age_groups` to each federal state and return a list of plots.

```
states <- sort(unique(covid$federal_state))
plots <- map(states, plot_covid_cases_by_age_groups)
plots[[7]]
```



# List columns in data frames

`map()` helps to work with list columns in dplyr pipelines.

For example, a ggplot2 plot is a list object.

Suppose we want to store the plots in a data frame where each row contains the name of the federal state and the area plot.

```
tibble(state = sort(unique(covid$federal_state))) %>%  
  mutate(plot_covid_cases_by_age_groups(state))  
  
## Warning in `==.default`(federal_state, state): longer object length is not a multiple of  
## shorter object length  
  
## Warning in is.na(e1) | is.na(e2): longer object length is not a multiple of shorter object  
## length  
  
## Error: Problem with `mutate()` input `..1`.  
## x Input `..1` must be a vector, not a `gg/ggplot` object.  
## i Input `..1` is `plot_covid_cases_by_age_groups(state)`.
```

The code yields an error because `dplyr` functions generally expect the output of functions to be of atomic vector type, i.e., logical, integer, double or character.

We can leverage `map` functions to get it work.

```
covid_plots <- tibble(state = sort(unique(covid$federal_state))) %>%
  mutate(plot = map(state, plot_covid_cases_by_age_groups))
covid_plots
```

```
## # A tibble: 16 x 2
##   state                  plot
##   <fct>                 <list>
## 1 Baden-Wuerttemberg    <gg>
## 2 Bavaria               <gg>
## 3 Berlin                <gg>
## 4 Brandenburg           <gg>
## 5 Bremen                <gg>
## 6 Hamburg               <gg>
## 7 Hesse                 <gg>
## 8 Mecklenburg-Western Pomerania <gg>
## 9 Lower Saxony           <gg>
## 10 North Rhine-Westphalia <gg>
## 11 Rhineland-Palatinate <gg>
## 12 Saarland              <gg>
## 13 Saxony                <gg>
## 14 Saxony-Anhalt         <gg>
## 15 Schleswig-Holstein    <gg>
## 16 Thuringia             <gg>
```

# Other useful `purrr` functions

`pluck()`

One-sided formulas

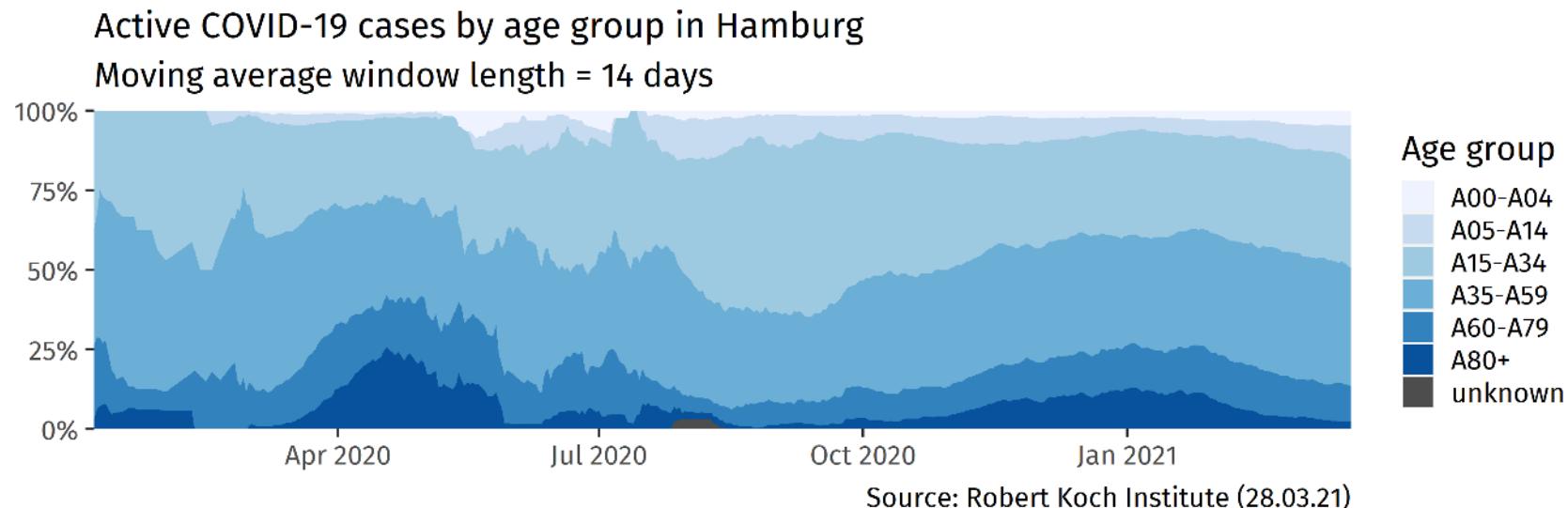
`map2()`

`walk*()`

Use `pluck()` to index into data structures. The function is particularly useful within a pipeline.

Suppose we want to extract the plot for Hamburg:

```
covid_plots %>%
  filter(state == "Hamburg") %>%
  pluck("plot", 1)
```



# Other useful `purrr` functions

`pluck()`

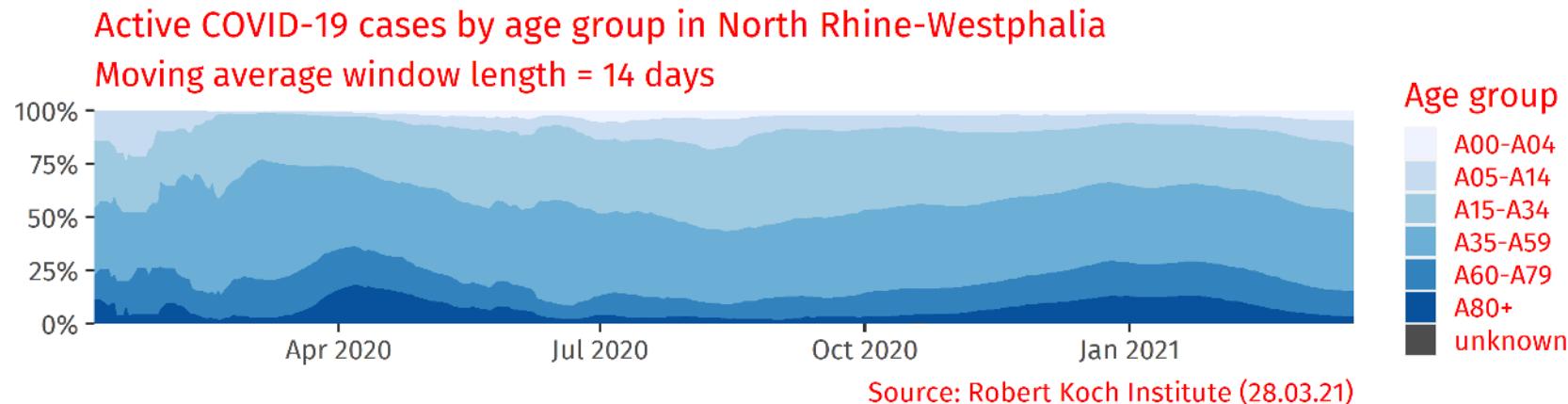
One-sided formulas

`map2()`

`walk*()`

To save some typing, use `~` (**pronounced twiddle**) for inline (anonymous) functions you want to use in `map()`:

```
covid_plots %>%
  mutate(plot = map(plot, ~ .x + theme(text = element_text(color = "red")))) %>%
  pluck("plot", 10)
```



```
# ...which is equivalent to:
covid_plots %>%
  mutate(plot = map(plot, function(x) {x + theme(text = element_text(color = "red"))})) %>%
  pluck("plot", 10)
```

# Other useful `purrr` functions

`pluck()`

One-sided formulas

`map2()`

`walk*()`

Use `map2()` if you want to map over **two** vectors.

Suppose we want to save our plots as png files, using the federal state's name as file name.

```
save_plot <- function(gg, name) {  
  ggsave(  
    filename = paste0(name, ".png"), plot = gg,  
    width = 28, height = 12, units = "cm", dpi = 300  
  )  
}  
covid_plots %>% mutate(save_plot = map2(plot, state, save_plot))
```

```
## # A tibble: 16 x 3  
##   state          plot  save_plot  
##   <fct>        <list> <list>  
## 1 Baden-Wuerttemberg <gg>   <NULL>  
## 2 Bavaria         <gg>   <NULL>  
## 3 Berlin          <gg>   <NULL>  
## 4 Brandenburg    <gg>   <NULL>  
## 5 Bremen          <gg>   <NULL>  
## 6 Hamburg         <gg>   <NULL>  
## 7 Hesse           <gg>   <NULL>  
## 8 Mecklenburg-Western Pomerania <gg>   <NULL>  
## 9 Lower Saxony    <gg>   <NULL>  
## 10 North Rhine-Westphalia <gg>   <NULL>
```

# Other useful `purrr` functions

`pluck()`

One-sided formulas

`map2()`

`walk*()`

Use `walk()` to apply a function for its **side-effect** to each element of a vector.

For each map function, there is an equivalent walk function, e.g. `map2() → walk2()`

```
walk2(covid_plots$plot, covid_plots$state, save_plot)
```

# Session info

```
## setting value
## version R version 4.0.4 (2021-02-15)
## os       Windows 10 x64
## system  x86_64, mingw32
## ui       RStudio
## language (EN)
## collate English_United States.1252
## ctype   English_United States.1252
## tz      Europe/Berlin
## date   2021-04-02
```

package	version	date	source
broom	0.7.5	2021-02-19	CRAN (R 4.0.3)
dials	0.0.9	2020-09-16	CRAN (R 4.0.3)
dplyr	1.0.5	2021-03-05	CRAN (R 4.0.4)
forcats	0.5.1	2021-01-27	CRAN (R 4.0.3)
gapminder	0.3.0	2017-10-31	CRAN (R 4.0.3)
ggplot2	3.3.3	2020-12-30	CRAN (R 4.0.3)
infer	0.5.4	2021-01-13	CRAN (R 4.0.3)
kableExtra	1.3.4	2021-02-20	CRAN (R 4.0.3)
knitr	1.31	2021-01-27	CRAN (R 4.0.3)
lubridate	1.7.10	2021-02-26	CRAN (R 4.0.4)
modeldata	0.1.0	2020-10-22	CRAN (R 4.0.3)
palmerpenguins	0.1.0	2020-07-23	CRAN (R 4.0.3)
parsnip	0.1.5	2021-01-19	CRAN (R 4.0.3)

package	version	date	source
readr	1.4.0	2020-10-05	CRAN (R 4.0.3)
recipes	0.1.15	2020-11-11	CRAN (R 4.0.3)
rsample	0.0.9	2021-02-17	CRAN (R 4.0.4)
scales	1.1.1	2020-05-11	CRAN (R 4.0.2)
stringr	1.4.0	2019-02-10	CRAN (R 4.0.2)
tibble	3.1.0	2021-02-25	CRAN (R 4.0.3)
tidymodels	0.1.2	2020-11-22	CRAN (R 4.0.3)
tidyverse	1.1.3	2021-03-03	CRAN (R 4.0.4)
tidyverse	1.3.0	2019-11-21	CRAN (R 4.0.2)
tune	0.1.2	2020-11-17	CRAN (R 4.0.3)
workflows	0.2.2	2021-03-10	CRAN (R 4.0.4)
xaringanBuilder	0.0.7.9001	2021-03-22	Github (jhelvy/xaringanBuilder)
yardstick	0.0.7	2020-07-13	CRAN (R 4.0.3)



**Thank you! Questions?**