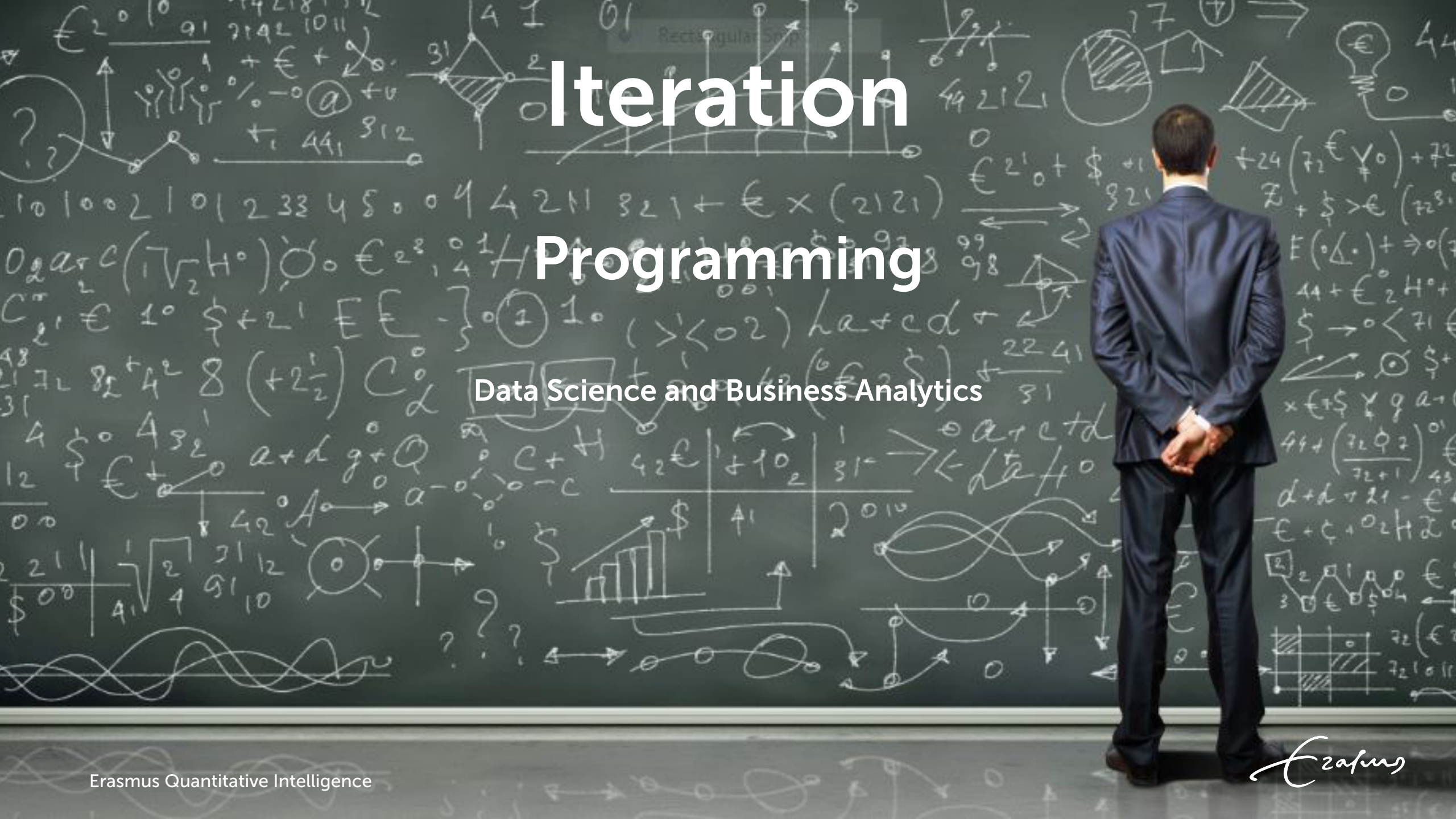
Iteration

Programming

Data Science and Business Analytics



Iteration

- Sometimes, you want to execute the same function on several inputs
  - For example: you want to add 1 to 1, 2, 3 and 4
  - Or: you want to know which element is `NULL` in `list(NULL, NA, 1, -5, 3)`
- Some functions are vectorized
  - `1:4 + 1`
- Some functions are not (as your own functions)
  - `is.null(list(NULL, NA, 1, -5, 3))`

Purrr-package

- the map-function from the purrr-package executes the given function by element
  `map(list(NA, NULL, 1, -5, 3), is.null)`
- the map-function returns a list
  `list(FALSE, TRUE, FALSE, FALSE, FALSE)`
- available through tidyverse

We will cover `map()` later.
We first start with `for` and `while` loops to get a better understanding of iteration.

Iteration: for loop

```
series <- list(NULL, NA, 1, -5, 3)
is_null <- vector('logical', 5)
for (i in 1:5) {
        is_null[i] <- is.null(series[[i]])
}
```

- The 'incrementor' (*i* in this case) gets the values 1 to 5, sequentially.
- The 'sequence' (*1 to 5* in this case) can be a vector or a list of values
  (we could also have said `i in series` in this case)
- The body of the for loop is executed sequentially over the incrementor

Iteration: while loop

```
series <- c(3)
while (max(series) < 1000) {
        series <- c(series, last(series)^3)
}
```

- The while loop has a conditional statement; as long as the condition is TRUE, the loop continues (infinitely is programmed incorrectly!)
- A (TRUE) starting value is needed to start your while loop

Exercises

Do Exercises 1 and 2

Efficiency

- The code inside a loop should be very efficient: the code is repeated possibly many times!
- Each time in the loop, R copies series. And thus each time stored, using memory
  see ‘while’-example
- Better: initialize the vector with the size it has to end with,
  see ‘for’-example

```
series <- list(NULL, NA, 1, -5, 3)
is_null <- vector(‘logical’, 5)
for (i in 1:5) {
        is_null[i] <- is.null(series[[i]])
}
```

```
series <- c(3)
while (max(series) < 1000 {
        series <- c(series, last(series)^3)
}
```

Efficiency: Fibonacci

```
> get_fibonacci_slow <- function(length) {
+     # initialize series
+     fib <- c(0, 1)
+
+     # loop from 3 to length
+     for (i in 3:length) {
+         # add the sum of the last two values to the series
+         fib <- c(fib, fib[i-1] + fib[i-2])
+     }
+
+     # output
+     fib
+ }
> tictoc::tic()
> fib <- get_fibonacci_slow(10000)
> tictoc::toc()
0.415 sec elapsed
```

```
> get_fibonacci_fast <- function(length) {
+     # initialize series
+     fib <- vector('numeric', length = length)
+     fib[1:2] <- c(0, 1)
+
+     # loop from 3 to length
+     for (i in 3:length) {
+         # add the sum of the last two values to the series
+         fib[i] <- fib[i-1] + fib[i-2]
+     }
+
+     # output
+     fib
+ }
>
> tictoc::tic()
> fib <- get_fibonacci_fast(10000)
> tictoc::toc()
0.009 sec elapsed
```

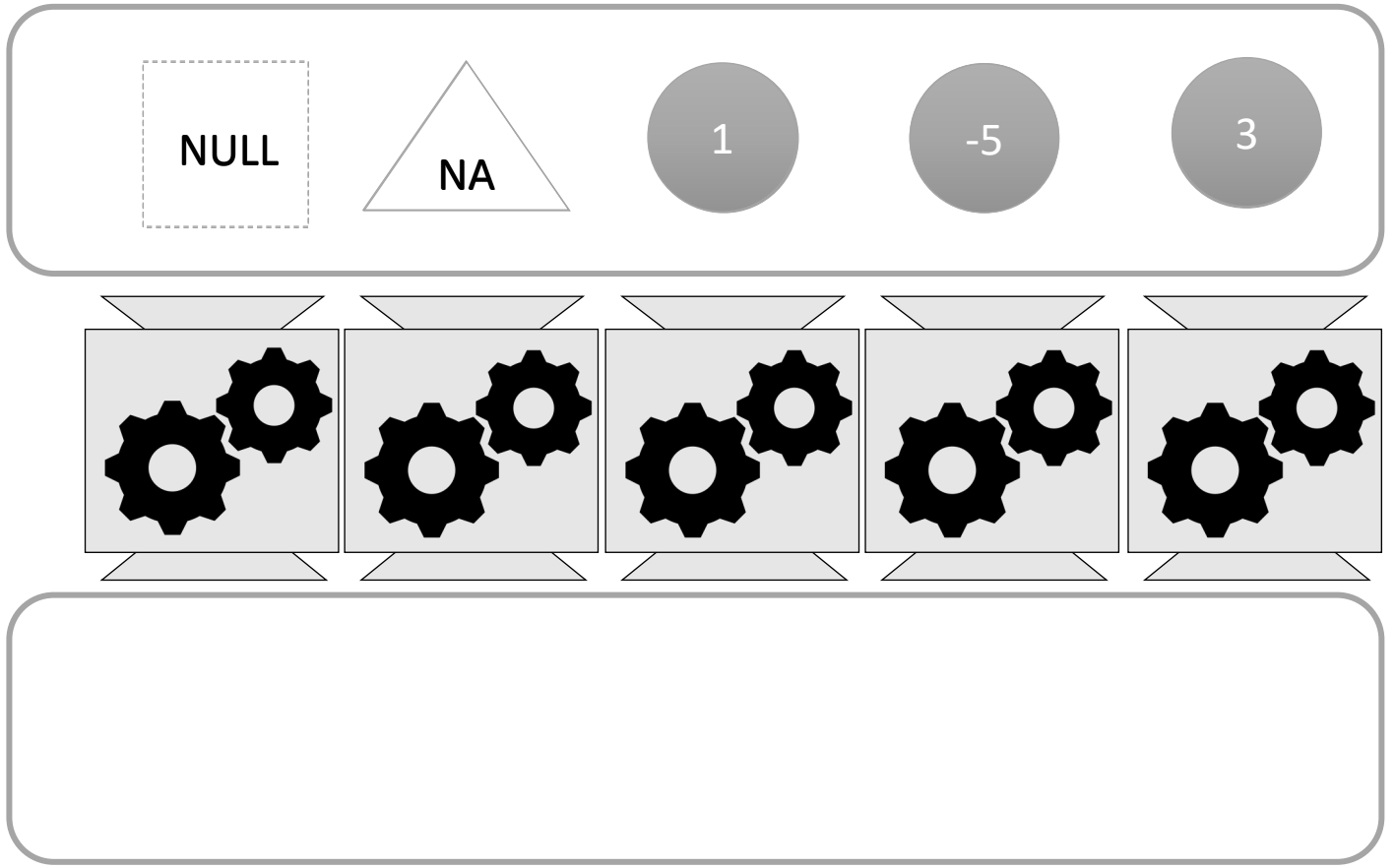
Iteration: purrr

- You can repeat code by using for and while loops
- Often, you want to execute the same piece of code for each element of
  - a vector
  - a list
  - a data.frame (each column)
- The function `map()` from the purrr-package executes a function looping over all elements of the argument

If your input is a vector, list or data frame
and your output is a vector list of data frame of the same length
and every execution is independent from other executions
then a `map()`-function is most applicable.

Iteration: purrr

```
test_list <- list(NULL, NA, 1, -5, 3)
map(test_list, is.null)
```



Iteration: purrr

- `map(.x, .f, ...)`
  - `.x` elements to execute the same function over
    can be a list, vector or data frame (elements are columns of the data frame)
  - `.f` the function to be executed
  - `...` other arguments to be given to the function

- Map can also return other formats than lists
  - `map_lgl(.x, .f, ...)` returns a logical vector (TRUE and FALSE)
  - `map_dbl(.x, .f, ...)` returns a numeric vector
  - `map_chr(.x, .f, ...)` returns a character vector
  - `map_df(.x, .f, ...)` returns a data frame

Exercises

- Do Exercise 3 and 4

Extra arguments in `map()`

- Often, a function needs more than 1 input argument

  For example, in the `mean()`-function we often add the second argument `na.rm = TRUE`

- `map(.x, .f, ...)`

  That is where we can use the `...`

  Room for other arguments to the function `.f`

  `.x` is the first argument of the function `.f`

- `map(data, mean, na.rm = TRUE)`

Iteration: purrr, using your own function

Source on Save

```
132 # purrr
133 test_list <- list(NA, NULL, 1, -5, 3)
134 map(test_list, is.null)
135
136 # map with own function
137 get_null <- function(x) {
138     if (is.null(x)) {
139         'This one is NULL'
140     } else {
141         'This one is not NULL'
142     }
143 }
144
145 map_chr(test_list, get_null)
146
147
148
149
150
151
152
```

Source on Save

```
147 # map with own anonymous function
148 map_chr(test_list, function(x) {
149     if (is.null(x)) {
150         'This one is NULL'
151     } else {
152         'This one is not NULL'
153     }
154 })
155
156
157
158
159
160
161
162
163
164
165
166
167
```

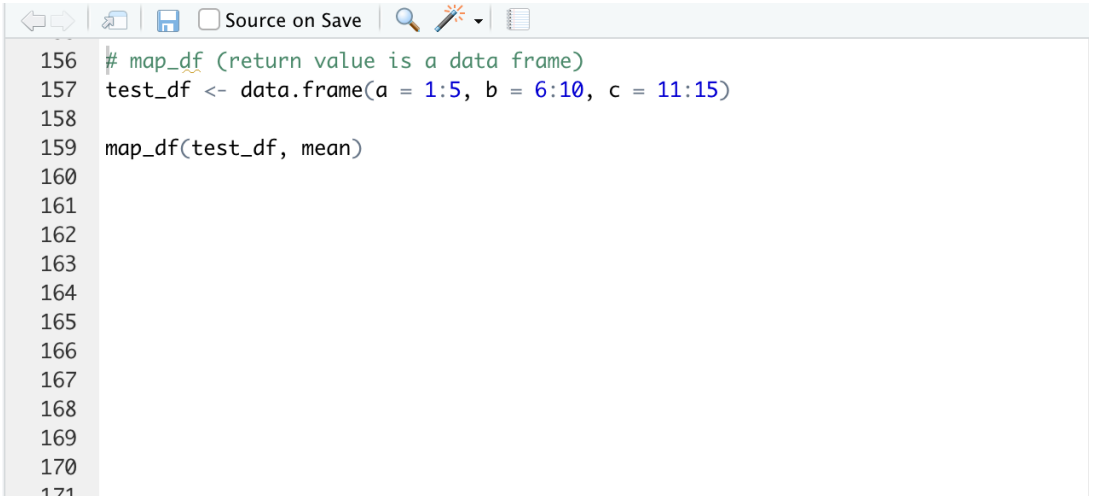
- An anonymous function is not stored in your working environment
- Use an anonymous function if you are not going to reuse the function later on

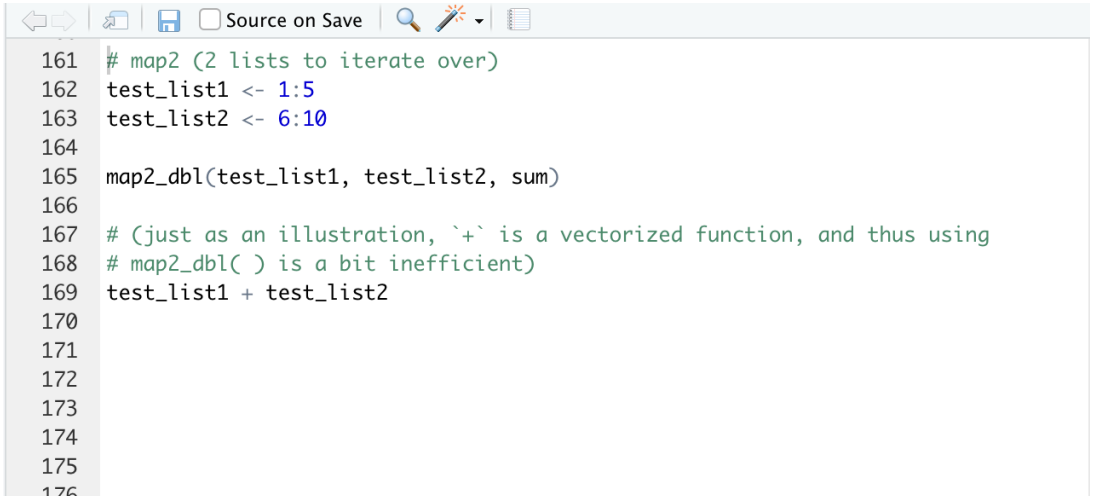
Iteration: other functions in purrr

- `map_df( )`
  - output is a data frame instead of a list
    (a data frame is a special type of list)
  - input can be of any form. If data frame: iterates over columns
  - the return value of the function should be vectors of equal length for all columns
  - the original names of the columns will be the names of the returned data frame

- `map2( )`
  - In this variant, the function is iterated over 2 input lists.
  - First, the first element of both lists is used in the function, than the second, etc.
  - The input lists should be of equal length

- `pmap( )`
  - Generic form of map2, iterates over p lists at the same time





Exercises

- Do Exercises 5-6

Summary

- If you want to execute the same function over several inputs, use iteration

Decide upon way of iteration in this order:

- If a function is vectorized, do not use iteration!

  `1:4 + 1`
- If your input is a vector, list or data frame

  and your output is a vector list of data frame of the same length

  and every execution is independent from other executions

  use the `map()`-function
- If the above is not possible, use a for loop

  least efficient way of programming