Practice R: The Tutorials

Edgar J. Treischl

2024-11-03

Table of contents

Preface

This website gives access to all tutorials of Practice R (Treischl 2023). Practice R is a text book for the social sciences which provides several tutorials supporting students to learn R. Feel free to inspect the tutorials even if you are not familiar with the book, but keep in mind the tutorials are supposed to complement the Practice R book.

The book

Many students learn to analyze data using commercial packages, even though there is an open-source software with cutting-edge possibilities: R, a programming language with countless cool features for applied empirical research.

Practice R introduces R to social science students, inspiring them to consider R as an excellent choice. In a non-technical pragmatic way, this book covers all typical steps of applied empirical research.

Learn how to prepare, analyze, and visualize data in R. Discover how to collect data, generate reports, or automate error-prone tasks.

The book is accompanied by an R package. This provides further learning materials that include interactive tutorials, challenging you with typical problems of applied research. This way, you can immediately practice the knowledge you have learned. The package also includes the source code of each chapter and templates that help to create reports.

Practice R has social science students in mind, nonetheless a broader audience may use Practice R to become a proficient R user.

- Introduces R in a non-technical fashion
- Covers typical steps of applied empirical research
- Complemented by interactive tutorials
- With access to all materials via the Practice R Package

1 Base R

Welcome to the `base` R tutorial (Chapter 2) of the Practice R book (Treischl 2023). Practice R is a text book for the social sciences which provides several tutorials supporting students to learn R. Feel free to inspect the tutorials even if you are not familiar with the book, but keep in mind these tutorials are supposed to complement the Practice R book.

In Chapter 2, I introduced R and we learned the basics about `base` R. Of course, `base` R has more to offer than I could possibly outline, but also much more than is necessary for your first steps with R. Consider how a `while()` loop works. A while loop repeats code until a certain condition is fulfilled. The next console shows the principle. The loop prints `i` and adds one to `i` until `i` is, for example, smaller than four. This example is extremely boring, but it illustrates the concept.

```
# Boring base R example
i <- 1

# while loop
while (i < 4) {
  print(i)
  i <- i + 1
}
```

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

I will introduce further `base` R features when we need them and I will not ask you to assign objects, create simple functions, or other probably boring `base` R examples in this tutorial. Such tasks are important but abstract in the beginning. Instead, we focus on typical errors that may occur while we start to work with R. Why do we not practice the discussed content of `base` R, but concentrate on errors in this tutorial?

To learn a new programming language is a demanding task. It does not even matter which programming language we talk about, there is an abundance of mistakes and errors (new) users face. For this reason we will get in touch with typical errors messages. For example, sometimes an error occurs only because of a spelling mistake. Can you find the typo in the next console?

```
# Fimd the typo
primt("Hello World")
```

```
#> Error in primt("Hello World"): could not find function "primt"
```

```
# Solution:
# Ah c'mon, read my friend ;)
```

RStudio has an auto-completion function which helps us to avoid such syntax errors, but learning R implies that you will come across many and sometimes cryptic errors messages (and warnings). Errors and debugging is hard work as the artwork from Allison Horst clearly shows. Thus, as a new user you will run into many errors and the question is how we can manage the process of debugging.

To support you in this process, we will reproduce errors in this tutorial. We try to understand what they mean and I ask you to fix them. We focus on typical errors that all new users face, explore cryptic errors you will soon come across, and further sources of errors. Finally, I summarize the introduced `base` R functions and I show you where to find more help in case you run into an error.

1.1 Typical error messages

What kind of errors do we need to talk about? Sometimes we introduce errors when we are not cautious enough about the code. Spelling mistakes (e.g., typos, missing and wrong characters, etc.) are easy to fix yet hard to find. For example, I tried to use the assignment operator, but something went wrong. Do you know what might be the problem?

```
#Assigning the values the wrong way
a -< 5
b -< 3

a + b
```

```
#> Error: <text>:2:4: unexpected '<'
#> 1: #Assigning the values the wrong way
#> 2: a -<
#>       ^
```

```
# Keep the short cut for the assignment operator in mind:
#<Alt/Option> + <->

# Solution:
a <- 5
b <- 3

a + b
```

```
#> [1] 8
```

Finding spelling mistakes in your own code can be hard. There are certainly several reasons, but our human nature to complete text certainly is part of it. This ability gives us the possibility to read fast, but it makes it difficult to see our own mistakes. Don't get frustrated, it happens even if you have a lot of experience working with R. Thus, check if there are no simple orthographically mistakes - such as typos, missing (extra) parentheses, and commas - which prevents the code from running.

I highlighted in Chapter 2 that RStudio inserts opening and closing parentheses, which reduces the chance that missing (or wrong) characters create an error, but there is no guarantee that we insert or delete one by chance. Suppose you try to estimate a mean in combination with the `round()` function. I put a parenthesis at a wrong place, which is why R throws an error. Can you see which parenthesis is causing the problem?

```
#Check parenthesis
round(mean(c(1, 4, 6))), digits = 2)
```

```
#> Error: <text>:2:24: unexpected ','
#> 1: #Check parenthesis
#> 2: round(mean(c(1, 4, 6))),
#>                            ^
```

```
# Solution:
round(mean(c(1, 4, 6)), digits = 2)
```

```
#> [1] 3.67
```

This error is hard to spot, but it illustrates that we need to be careful not to introduce mistakes. Moreover, RStudio gives parentheses that belong together the same color which help us to keep

overview. Go to the RStudio menu (via the `<Code>` tab) and select *rainbow parentheses* if they are not displayed in color in the Code pane.

Unfortunately, RStudio cannot help us all the time because some R errors messages (and warnings) are cryptic. There are even typical errors messages that are quite obscure for beginners. For example, R tells me all the time that it can't find an object, functions, and data. There are several explanations why R throws such an error. If R cannot find an object, check if the object is listed in the environment. If so, you know for sure that the object exists and that other reasons cause the error. R cannot find an object even in the case of a simple typo.

```
# R cannot find an object due to typos
mean_a <- mean(1, 2, 3)
maen_a
```

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

```
# Solution:
mean_a <- mean(1, 2, 3)
mean_a
```

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

R tells us that a function (an object) cannot be found if different notations are used. Keep in mind that R is case-sensitive (`r` vs. `R`) and cannot apply a function (or find an object) that does not exist, as the next console illustrates. Of course, the same applies if you forgot to execute the function before using it or if the function itself includes an error and cannot be executed. In all these examples R cannot find the function (or object).

```
# R is case-sensitive
return_fun <- function(x) {
  return(x)
}

Return_fun(c(1, 2, 3))
```

```
#> Error in Return_fun(c(1, 2, 3)): could not find function "Return_fun"
```

```
# Solution:
return_fun(c(1, 2, 3))
```

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

What is the typical reason why a function from an R package cannot be found? I started to introduce the `dplyr` package in Chapter 2 (Wickham et al. 2022). Suppose we want to use the `select` function from the package. To use anything from an R package, we need to load the package with the `library()` function each time we start (over). Otherwise, R cannot find the function.

```
# Load the package to use a function from a package
library(palmerpenguins)
select(penguins, species)
```

```
#> Error in select(penguins, species): could not find function "select"
```

```
# Solution:
dplyr::select(penguins, species)
```

```
#> # A tibble: 344 x 1
#>    species
#>    <fct>
#>  1 Adelie
#>  2 Adelie
#>  3 Adelie
#>  4 Adelie
#>  5 Adelie
#>  6 Adelie
#>  7 Adelie
#>  8 Adelie
#>  9 Adelie
#> 10 Adelie
#> # i 334 more rows
```

The same applies to objects from a package (e.g., data). The `.packages()` function returns all loaded (attached) packages, but there is no need to keep that in mind. Go to the packages pane and check if a package is installed and loaded. R tells us only that the function cannot be found if we forget to load it first.

```
# Inspect the loaded packages via the Packages pane
loaded_packages <- .packages()
loaded_packages
```

```
#> [1] "palmerpenguins" "stats"          "graphics"       "grDevices"
#> [5] "utils"          "datasets"       "methods"        "base"
```

Ultimately, suppose we try to import data. Never mind about the code, we focus on this step in Chapter 5 in detail, but R tells us that it *cannot open the connection* if the file cannot be found in the current working directory.

```
# Load my mydata
read.csv("mydata.csv")
```

```
#> Warning in file(file, "rt"): cannot open file 'mydata.csv': No such file or
#> directory
```

```
#> Error in file(file, "rt"): cannot open the connection
```

R tells that data, or other files cannot be found because we provided the wrong path to the file. We will learn how to import data later, but keep in mind that R cannot open a file if we search in the wrong place. In Chapter 2, I outlined many possibilities to change the work directory for which RStudio supplies convenient ways. In addition, the `getwd()` function returns the current work directory in case of any doubts.

```
# Do we search for files in the right place
getwd()
#> [1] "C:/Users/Edgar/R/Practice_R/Tutorial/02"
```

```
#> [1] "C:/Users/Edgar/R/Practice_R/Tutorial/02"
```

Loading the right packages and searching in the right place does not imply that we cannot inadvertently introduce mistakes. Suppose you want to apply the `filter` function from the `dplyr` package. You copy and adjust the code from an old script, but R returns an error. Can you see where I made the mistake? I tried to create a subset with `Adelie` penguins only, but `dplyr` seems to know what the problem might be.

```
# Mistakes happen all the time ...
library(dplyr)
filter(penguins, species = "Adelie")
```

```
#> Error in `filter()`:
#> ! We detected a named input.
#> i This usually means that you've used `=` instead of `==`.
#> i Did you mean `species == "Adelie"`?
```

```
# Solution:
library(dplyr)
filter(penguins, species == "Adelie")
```

```
#> # A tibble: 152 x 8
#>    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
#>    <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
#>  1 Adelie  Torgersen           39.1          18.7               181        3750
#>  2 Adelie  Torgersen           39.5          17.4               186        3800
#>  3 Adelie  Torgersen           40.3          18                 195        3250
#>  4 Adelie  Torgersen           NA            NA                  NA          NA
#>  5 Adelie  Torgersen           36.7          19.3               193        3450
#>  6 Adelie  Torgersen           39.3          20.6               190        3650
#>  7 Adelie  Torgersen           38.9          17.8               181        3625
#>  8 Adelie  Torgersen           39.2          19.6               195        4675
#>  9 Adelie  Torgersen           34.1          18.1               193        3475
#> 10 Adelie  Torgersen           42            20.2               190        4250
#> # i 142 more rows
#> # i 2 more variables: sex <fct>, year <int>
```

Typos, missing functions (objects), and confusion about operators are typical mistakes and some packages return suggestions to fix the problem. Unfortunately, R can also return cryptic error messages, which are often harder to understand.

1.2 Cryptic errors

Not all error R messages and warnings are cryptic. Suppose you wanted to estimate a mean of an `income` variable. The variable is not measured numerically which implies that the mean cannot be estimated. Consequently, R warns us about wrong and inconsistent data types.

```
# Warning: argument is not numeric or logical
income <- c("More than 10000", "0 - 999", "2000 - 2999")
mean(income)
```

```
#> Warning in mean.default(income): argument is not numeric or logical: returning
#> NA
```

Unfortunately, some errors and warnings seem more like an enigma than useful feedback. Imagine, R tells you that a *non-numeric argument* has been applied to a *binary operator*. The

next console reproduces the error with two example vectors. The last value of the vector `y` is a character (e.g., a missing value indicator: `NA`) and for obvious reasons we cannot multiply `x` with `y` as long as we do clean the latter.

```
# Cryptic error: A non-numeric argument to binary operator
x <- c(3, 5, 3)
y <- c(1, 4, "NA")

result <- x * y
```

```
#> Error in x * y: non-numeric argument to binary operator
```

```
result
```

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

We will learn how to fix such problem in a systematic manner later, for now just keep in mind that such an error message might be due to messy, not yet prepared data. Or suppose you tried to estimate the sum but R tells you that the code includes an *unexpected numeric constant*. Any idea what that means and how to fix the example code of the next console?

```
#Cryptic error: Unexpected numeric constant
sum(c(3, 2 1))
```

```
#> Error: <text>:2:12: unexpected numeric constant
#> 1: #Cryptic error: Unexpected numeric constant
#> 2: sum(c(3, 2 1
#>               ^
```

```
# Solution:
sum(c(3, 2, 1))
```

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

R finds an unexpected numeric constant (here `1`) because I forgot the last comma inside the `c()` function. The same applies to strings and characters. R tells us that there is an unexpected *string constant*. Can you see where?

```
#Cryptic error: Unexpected string constant
names <- c("Tom", "Diana"___"Pete")
names
```

```
#> Error: <text>:2:26: unexpected input
#> 1: #Cryptic error: Unexpected string constant
#> 2: names <- c("Tom", "Diana"_
#>                             ^
```

```
# Solution:
names <- c("Tom", "Diana", "Pete")
names
```

```
#> [1] "Tom"   "Diana" "Pete"
```

Or consider *unexpected symbols*. Can you find the problem of the next console. I used to `round` function but something went wrong with the `digits` option.

```
#Cryptic error: Unexpected symbol
x <- mean(c(1:3))
round(x digits = 2)
```

```
#> Error: <text>:3:9: unexpected symbol
#> 2: x <- mean(c(1:3))
#> 3: round(x digits
#>            ^
```

```
# Solution:
x <- mean(c(1:3))
round(x, digits = 2)
```

```
#> [1] 2
```

Thus, we introduce a mistake with a function argument because the comma is missing. A similar mistake happens if we forget to provide a necessary argument or provide a wrong one. For example, there is no `numbers` option of the `round` function as the next console (and the help files `?round`) outline.

```
# Cryptic error: Unused argument
x <- mean(c(1:3))
round(x, numbers = 2)
```

```
#> Error in round(x, numbers = 2): unused argument (numbers = 2)
```

```
# Solution:
x <- mean(c(1:3))
round(x, digits = 2)
```

```
#> [1] 2
```

Try to be patient and be kind to yourself should you run into such an error. You will become better to solve errors, but they will happen all the time. Let me give you one more for the road. Consider the error message: *object of type 'closure' is not subsettable*. R returns this error message if we try to slice a variable that does not exist or if we try to slice a function instead of providing a column vector. Can you fix the next console and provide a column vectors instead of slicing the `mean()` function?

```
# Cryptic error: Object of type 'closure' is not subsettable
mean[1:5]
```

```
#> Error in mean[1:5]: object of type 'closure' is not subsettable
```

```
# Solution:
mean(1:5)
```

```
#> [1] 3
```

1.3 Further sources of errors

There are further errors and mistakes and this tutorial cannot capture them all. As a minimum, I try to give you a heads-up that it takes time and experience to overcome such problems. For example, consider one more time the small data that we used to slice data in Practice R.

```r
# Save data as df
df <- tibble::tribble(
  ~names, ~year, ~sex,
  "Bruno", 1985, "male",
  "Justin", 1994, "male",
  "Miley", 1992, "female",
  "Ariana", 1993, "female"
)
```

Do you still remember how to slice the data? Give it a try with the following examples:

```r
# Slice the first column (variable)
df[1]
```

```
#> # A tibble: 4 x 1
#>   names
#>   <chr>
#> 1 Bruno
#> 2 Justin
#> 3 Miley
#> 4 Ariana
```

```r
# First row
df[1, ]
```

```
#> # A tibble: 1 x 3
#>   names  year sex
#>   <chr> <dbl> <chr>
#> 1 Bruno  1985 male
```

Suppose that you have not worked with R for a few weeks, would you still be able to remember how slicing works? We all face the same problems when we start to learn something new: you need several attempts before you understand how to get the desired information. Later, after slicing data many times, you will no longer think about how it works. Thus, be patient and kind to yourself, because some concepts need time and experience to internalize them.

Moreover, there are often several approaches to reach the same goal and - depending on your preferred style - some are harder or easier to apply. Say you need the `names` of the stars as a column vector. Can you slice the data or use the `$` operator to get the `names` variable from the data frame?

```
# Slice or use the $ operator
names <- df$names
names <- df[1]
names
```

```
#> # A tibble: 4 x 1
#>   names
#>   <chr>
#> 1 Bruno
#> 2 Justin
#> 3 Miley
#> 4 Ariana
```

Unfortunately, some mistakes are logical in nature and pure practice cannot help us to overcome such problems. Consider the next console. I created a slice function (`slice_function`) which is supposed to return an element of a vector `x`, but so far it only returns non-sense. Why does it not return the second element of the input data?

```
# A pretty messed up slice_function
data <- c(3, 9, 1, 5, 8, "999", 1)

slice_function <- function(data, x) {
  data[x]
}

slice_function(2)
```

```
#> [1] 2
```

```
# Solution:
data <- c(3, 9, 1, 5, 8, 1)

slice_function <- function(data, x) {
  data[x]
}

slice_function(data, x = 2)
```

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

Soon, your code will encompass several steps, try to break it into its separate elements and then examine each step carefully. For example, inspect the vector `x` to see if error was introduced in the first step. Use the `class()` function to examine if the input of a variable is as expected (e.g. numerical). If we are sure about the input, we would go on to the next step and so on. Certainly, the last example is not complicated but the complexity of code (and the tasks) will increase from the chapter to chapter. By breaking down all steps into elements, you may realize where the error occurs and how you can fix it.

1.4 Summary

All tutorials of Practice R will end with a short code summary of the corresponding book chapter. The summary only contains the function name from the R help file and code example of the most important functions and packages. In connection with Chapter 2, keep the following functions in mind:

- Install packages from repositories or local files (`install.packages`)
- Loading/attaching and listing of packages(`library`)
- Inspect the help file (`?function`)
- Combine Values into a vector or list (`c`)
- Compare objects (`<=, >=, ==, !=`)
- Replicate elements of vectors and lists (`rep`)
- Sequence generation (`seq`)
- Sum of vector elements (`sum`)
- Length of an object (`length`)
- Object classes (`class`)
- Data frames (`data.frame`)
- Build a data frame (`tibble::tibble`, Müller and Wickham 2022b)
- Row-wise tibble creation (`tibble::tribble`)
- The number of rows/columns of an array (`nrow`/`ncol`)

`Base` R and many R packages have cheat sheets that summarize the most important features. You can inspect them directly from RStudio (via the `<help>` tab) and I included the link to the `base` R cheat sheet in the `PracticeR` package.

```
# Cheat sheets summarize the most important features
# The base R cheat sheet
PracticeR::show_link("base_r")
```

2 Data Exploration

Welcome to the data exploration tutorial of the Practice R book (Treischl 2023). Practice R is a text book for the social sciences which provides several tutorials supporting students to learn R. Feel free to inspect the tutorials even if you are not familiar with the book, but keep in mind these tutorials are supposed to complement the Practice R book.

In this tutorial we recapture the most important functions to explore data, but this time you will explore the `palmerpenguins` package and the `penguins` data (Horst, Hill, and Gorman 2022). The latter contains information about three different penguins species (Adélie, Chinstrap, and Gentoo) and Allison Horst has made some wonderful illustrations of them. Click on the hex sticker to inspect the package website.

```
# Tutorial 03: Explore data
library(dplyr)
library(GGally)
library(summarytools)
library(skimr)
library(palmerpenguins)
library(visdat)
```

The tutorial has the same structure as Chapter 3: We explore categorical variables, continuous variables, and effects. Before we start with variables, it is always a good idea to explore the data in general terms. First, I assigned the data as `df`, which makes it possible for us to recycle a lot of code from Chapter 3. Next, explore which variables does the `penguins` data contain. Use the `glimpse()` or the `str()` function for a first look of the `penguins` data. The `glimpse()` function is loaded via the `dplyr` package, but comes from the `pillar` package (Müller and Wickham 2022a).

```
# Use glimpse, head, or the str function for a first look
df <- penguins
glimpse(df)
```

```
#> Rows: 344
#> Columns: 8
#> $ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel~
```

```
#> $ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse~
#> $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
#> $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
#> $ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
#> $ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
#> $ sex               <fct> male, female, female, NA, female, male, female, male~
#> $ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007~
```

Thus, there are several factor variables such as penguin's `species` or `island`; numerical variables such as bill (`bill_length_mm`) and flipper length (`flipper_length_mm`); and integers such as the `year` variable. Keep in mind that R packages come with help files that show us how functions work and they provide more information about data. Use the help function (`?penguins`) if you feel insecure about the content of the data.

2.1 Categorical variables

We started to explore categorical variables in Chapter 3 and I outlined a few basics about factor variables. Suppose we want to explore the factor variable `island`, which indicates where the penguins live. How can you examine unique group levels?

```
# Inspect the levels() of the penguin's home island
levels(df$island)
```

```
#> [1] "Biscoe"    "Dream"     "Torgersen"
```

We will deepen our knowledge about factor variables in Chapter 5, but keep in mind that we can (re-) create and adjust `factor()` variables. For example, suppose the data looks like a messy character vector for penguin's `sex` that I have created in the next console. In such a case it is good to remember that we can give the variable proper text `labels` (e.g., `female` for `f`) and examine the results.

```
# Example of a messy factor variable
sex <- c("m", "f", "f")

# Give clearer labels
sex <- factor(sex,
  levels = c("f", "m"),
  labels = c("female", "male"),
)
head(sex)
```

```
#> [1] male   female female
#> Levels: female male
```

Tables help us to explore data and we used the `summarytools` package to make frequency and cross tables (Comtois 2022). Keep in mind that we will learn how to create text documents with tables and graphs in Chapter 8. For the moment it is enough to remember that we can create different sort of tables with the `summarytools` package. For example, create a frequency (`freq`) table to find out on which `island` most of the penguins live.

```
# Create a frequency table
freq(df$island)
```

```
#> Frequencies
#> df$island
#> Type: Factor
#>
#>                   Freq   % Valid   % Valid Cum.   % Total   % Total Cum.
#> --------------- ------ --------- -------------- --------- --------------
#>          Biscoe    168     48.84          48.84     48.84          48.84
#>           Dream    124     36.05          84.88     36.05          84.88
#>       Torgersen     52     15.12         100.00     15.12         100.00
#>            <NA>      0                               0.00         100.00
#>           Total    344    100.00         100.00    100.00         100.00
```

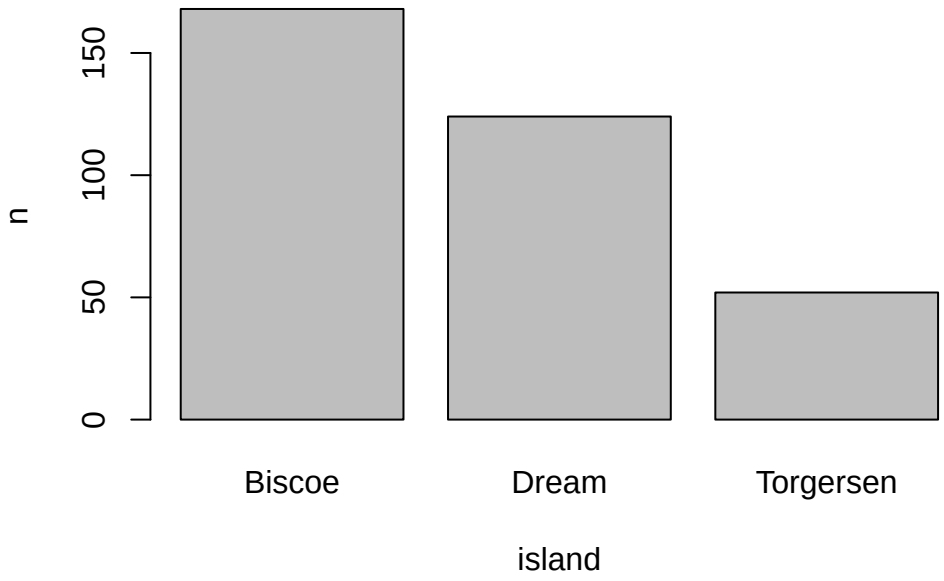
As outlined in the book, we can use the `table()` function to count categorical variables and plot the result as a bar graph. I introduced the latter approach because it is very easy to apply, but our code becomes clearer if we make the necessary steps visible. First, we need to count the levels before we can plot the results. The `count()` function from the `dplyr` package does this job (Wickham et al. 2022). It needs only the data frame and the factor variable.

```
# Count islands with dplyr
count_island <- dplyr::count(df, island)
count_island
```

```
#> # A tibble: 3 x 2
#>   island        n
#>   <fct>     <int>
#> 1 Biscoe      168
#> 2 Dream       124
#> 3 Torgersen    52
```

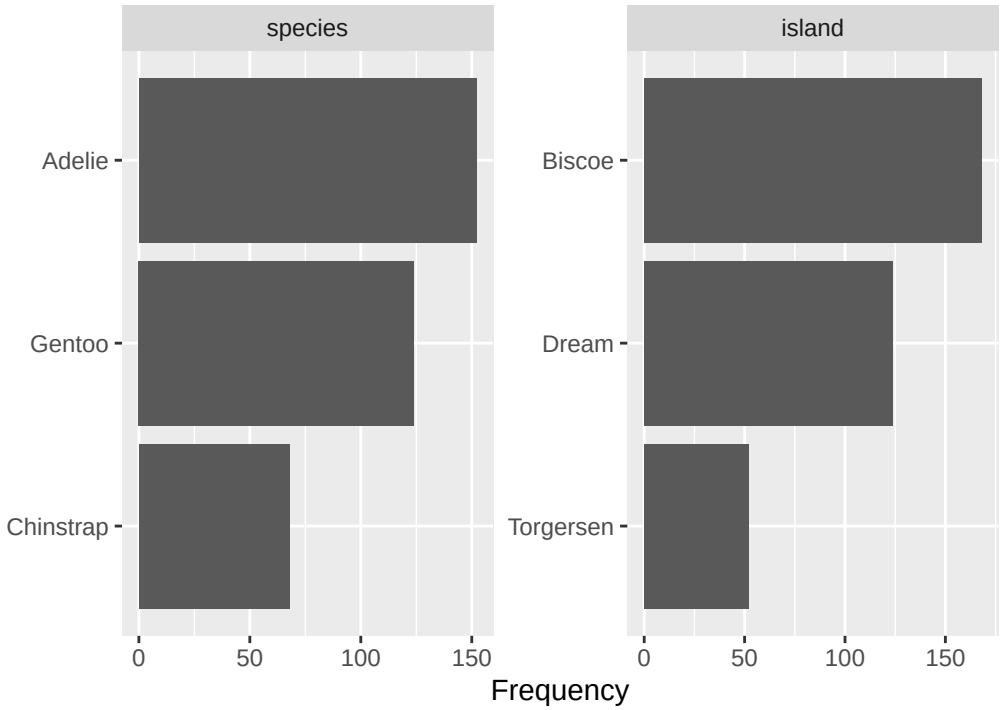
Next, use the assigned results (`count_island`) and insert the variables into the `barplot()` function (with the formula `y ~ x`).

```
# Create a barplot
barplot(n ~ island, data = count_island)
```



In a similar vein, I introduced functions from the `DataExplorer` package that help us to get a quick overview (Cui 2020). For example, use the `plot_bar()` function to depict several or all discrete variables of a data frame.

```
# Inspect all or several plots at once
DataExplorer::plot_bar(df[1:2])
```



2.2 Continuous variables

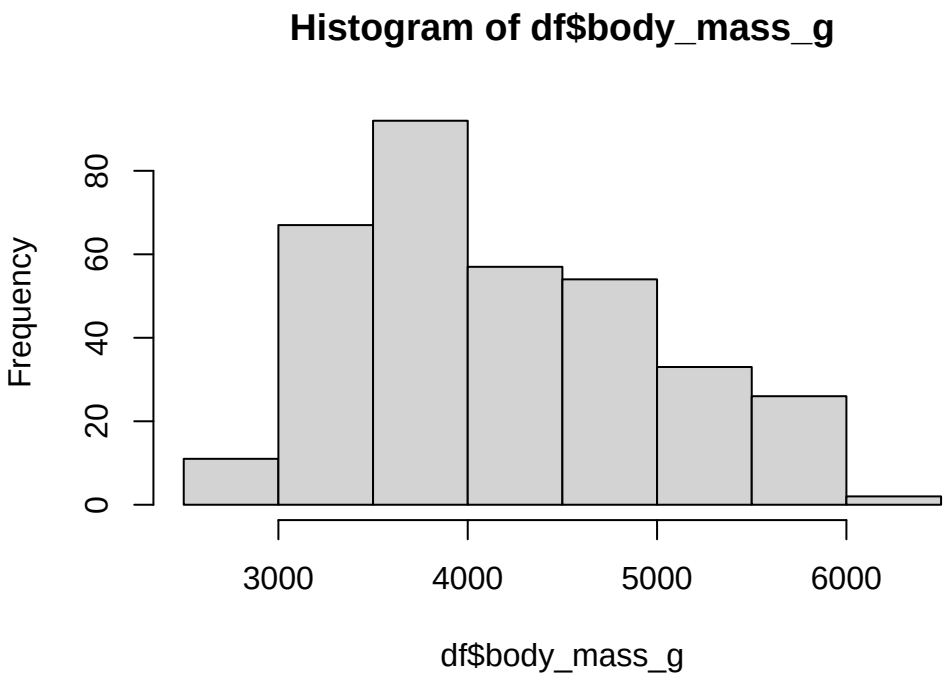
To explore continuous variables, estimate the summary statistics with the `summary()` function. Pick one variable such as penguin's body mass in gram (`body_mass_g`) or use the entire data frame.

```
# Get a summary
summary(df[1:4])
```

```
#>       species          island    bill_length_mm  bill_depth_mm
#>  Adelie   :152   Biscoe   :168   Min.   :32.10   Min.   :13.10
#>  Chinstrap: 68   Dream    :124   1st Qu.:39.23   1st Qu.:15.60
#>  Gentoo   :124   Torgersen: 52   Median :44.45   Median :17.30
#>                                  Mean   :43.92   Mean   :17.15
#>                                  3rd Qu.:48.50   3rd Qu.:18.70
#>                                  Max.   :59.60   Max.   :21.50
#>                                  NA's   :2       NA's   :2
```

The classic approach to visualize the distribution of a continuous variable is a histogram. Use the `hist()` function to display the distribution of the penguins body mass.

```
# Create a histogram
hist(df$body_mass_g)
```



Keep in mind that we only explored the data for the first time. We did not clean the data nor did we prepare the variables. We have to be explicit about missing values when we want to apply functions such as the `mean`. The function returns `NA`, but only because of a missing values problem. Can you remember how to fix this problem and estimate, for example, the mean?

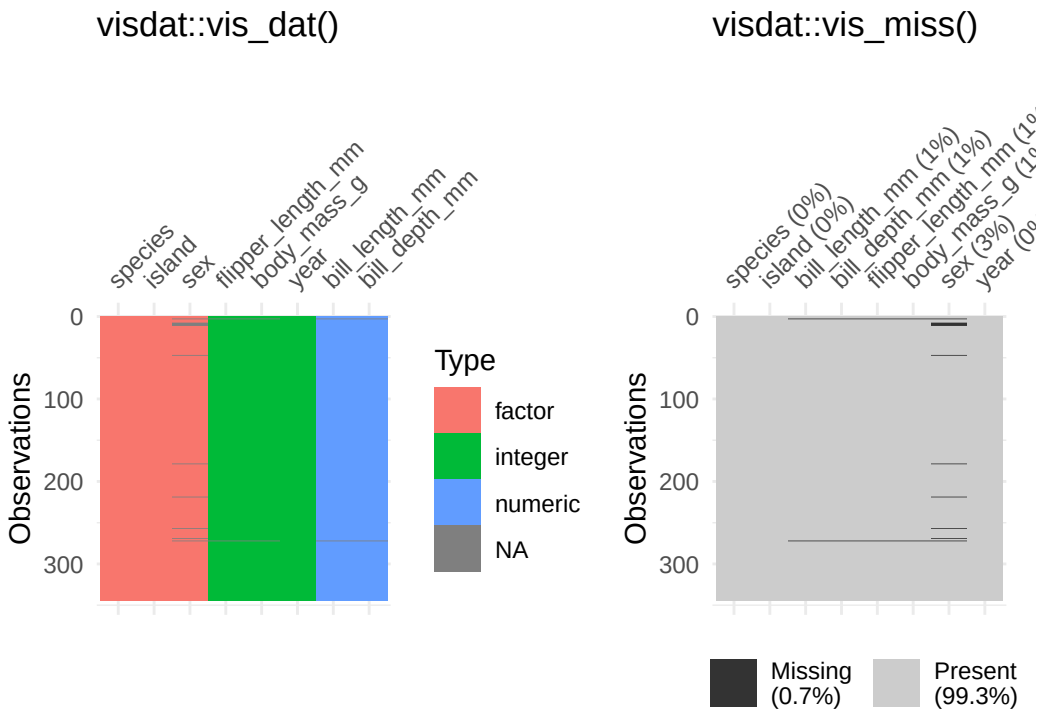
```
# Calculate the mean, but what about missing values (na.rm)?
mean(df$body_mass_g, na.rm = TRUE)
```

```
#> [1] 4201.754
```

I picked data that was more or less prepared to be explored, because data preparation needs more time and effort especially in the beginning. For this reason we will learn how to manipulate data in Chapter 4; and Chapter 5 tries to prepare you for own journey. For example,

we use packages such as `visdat` and `naniar` to identify missing values, as the next console illustrates with two examples (Tierney et al. 2021). The `vis_dat()` function from the corresponding packages shows us which type of data we have with missing values in gray; while `vis_miss()` visualizes missing values in general terms. Keep in mind that Chapter 3 did not introduce data preparation steps which are often necessary to explore data and effects between variables.

```
library(visdat)

# Left plot: vis_dat()
vis_dat(df)

# Right plot: vis_miss()
vis_miss(df)
```



2.3 Explore effects

Let's start with an *effect between two categorical variables*. There are different packages that provides functions to create (cross) tables, but we used the `summarytools` package. It even

provides a simulated data set which we will use the repeat the steps to create a cross table. The package comes with the `tobacco` data, which illustrates that smoking is harmful. As the next console shows, it indicates if a person is a `smoker` and if the person is `diseased`.

```
head(tobacco)[1:8]
```

```
#>   gender age age.gr      BMI smoker cigs.per.day diseased      disease
#> 1      M  75   71 + 29.50225     No            0       No         <NA>
#> 2      F  35  35-50 26.14989     No            0      Yes Neurological
#> 3      F  70  51-70 27.53183     No            0       No         <NA>
#> 4      F  40  35-50 24.05832     No            0       No         <NA>
#> 5      F  75   71 + 22.77486     No            0      Yes      Hearing
#> 6      M  38  35-50 21.46412     No            0       No         <NA>
```

Use the `ctable` function from the `summarytools` package to make a cross table for these variables. See also what happens if you adjust the `prop` option. Insert `c` or `t`. Furthermore, explore what happens if you set the `chisq`, `OR`, or `RR` option to `TRUE`.

```
# Create a cross table with summarytools
summarytools::ctable(
  x = tobacco$smoker,
  y = tobacco$diseased,
  prop = "r",
  chisq = TRUE,
  OR = TRUE
)
```

```
#> Cross-Tabulation, Row Proportions
#> smoker * diseased
#> Data Frame: tobacco
#>
#>
#> -------- ---------- ------------- ------------- ---------------
#>            diseased           Yes            No           Total
#>   smoker
#>      Yes              125 (41.9%)   173 (58.1%)    298 (100.0%)
#>       No               99 (14.1%)   603 (85.9%)    702 (100.0%)
#>    Total              224 (22.4%)   776 (77.6%)   1000 (100.0%)
#> -------- ---------- ------------- ------------- ---------------
#>
#> ----------------------------
```

```
#>  Chi.squared   df   p.value
#> ------------- ---- ---------
#>    91.7088     1       0
#> ---------------------------
#>
#> ---------------------------------
#>  Odds Ratio   Lo - 95%   Hi - 95%
#> ------------ ---------- ----------
#>     4.40        3.22       6.02
#> ---------------------------------
```

The `prop` option lets you determine the proportions: rows (`r`), columns (`c`), total (`t`), or none (`n`). Furthermore, the function even adds the chi-square statistic (`chisq`); the odds ratio (`OR`) or the relative risk (`RR`) if we set them to `TRUE`. Never mind if you are not familiar with the latter, the discussed options only illustrated how the `summarytools` package helps us to explore data and effects.

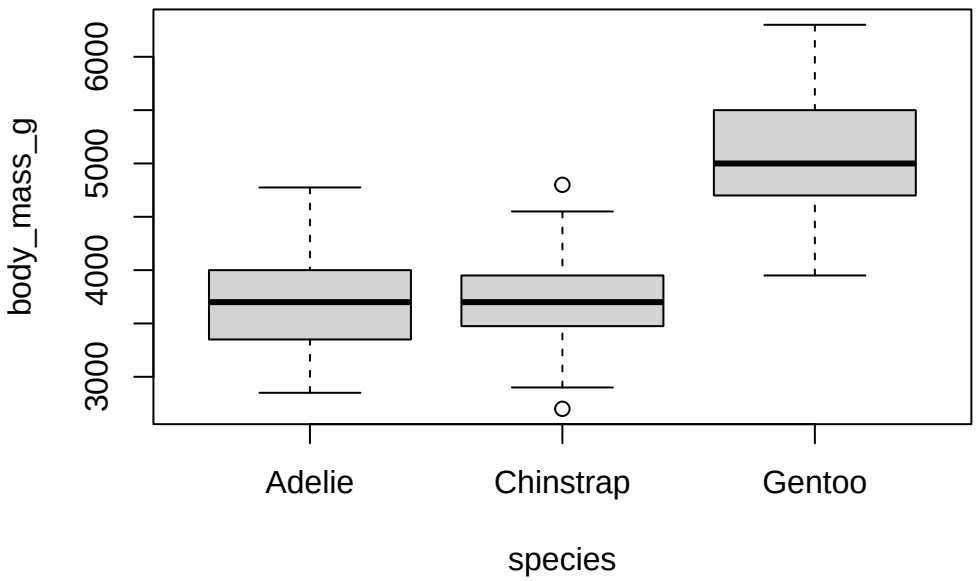
In the social sciences we are often interested in comparing *numerical outcomes between categorical variables* (groups). For example, one of the penguin's species has a higher body mass and we can examine which penguins `species` differ in terms of their body mass (`body_mass_g`). With `base` R, the `aggregate()` function lets us split the data and we are able to estimate the mean for each species.

```
# Aggregate splits the data into subsets and computes summary statistics
aggregate(df$body_mass_g, list(df$species), FUN = mean, na.rm = TRUE)
```

```
#>     Group.1        x
#> 1    Adelie 3700.662
#> 2 Chinstrap 3733.088
#> 3    Gentoo 5076.016
```

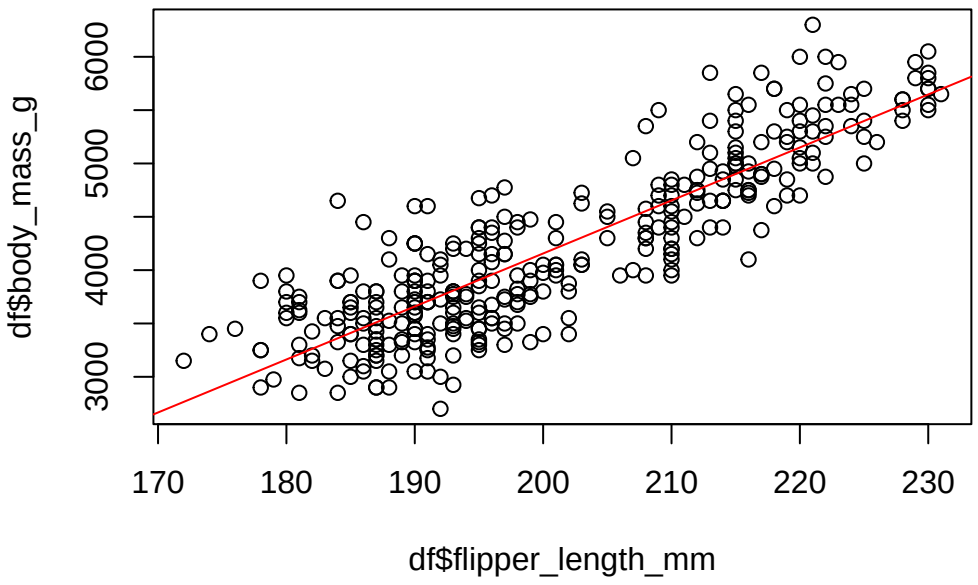
To calculate a group-mean looks quite complicated and I did not introduce the latter since we will systematically work on our skills to manipulate data in the next Chapter. Instead, we used a box plot to explore a continuous outcome between groups. As outlined in the book, box plots can be very helpful to compare groups even though they have graphical limitations since they do not display the data. Keep the `boxplot()` function in mind and practice one more time how it works. Inspect how penguin's body mass differs between the species.

```
# Inspect group differences with a box plot
boxplot(body_mass_g ~ species, data = df)
```



If we examine an *effect between two continuous outcomes*, we have to keep in mind that the `plot` function returns a scatter plot and we may insert a regression line with the `abline` and the `lm` function. Do you still know how it works? Create a scatter plot to examine the association between the body mass (`body_mass_g`) and the flipper length (`flipper_length_mm`) of the `penguins`.

```
# Create a scatter plot
plot(y = df$body_mass_g, x = df$flipper_length_mm)

# And a red regression line
abline(lm(body_mass_g ~ flipper_length_mm, data = df),
  col = "red"
)
```



Furthermore, we learned how to calculate the correlation coefficient. The code of the next console does not work if I apply the `cor()` with the penguins data. Do you have any idea how to fix the problem?

```
# Calculate the correlation between x and y
cor_penguins <- cor(df$body_mass_g, df$flipper_length_mm,
  use = "complete"
)
cor_penguins
```

```
#> [1] 0.8712018
```

By the way, the `cor()` also returns Kendall's or Spearman's if you adjust the `method` option:

```
# estimate a rank-based measure of association
cor(x,
  y = NULL, use = "complete",
  method = c("pearson", "kendall", "spearman")
)
```

Finally, the `effectsize` package helped us with the interpretation of Pearson's r (and other stats, see Chapter 6). I copied the code from the book; can you adjust it to interpret the effect of the examined variables with the `effectsize` package (Ben-Shachar et al. 2022)?

```
#> [1] 0.8712018
```

```
# Use effectsize to interpret R
effectsize::interpret_r(cor_penguins, rules = "cohen1988")
```

```
#> [1] "large"
#> (Rules: cohen1988)
```

There are more R packages to explore data than I could possibly outline. For example, consider the `skimr` package (Waring et al. 2022). It skims a data set and returns, for example, a short summary, summary statistics, and missing values. Inspect the vignette and `skim()` the data frame.

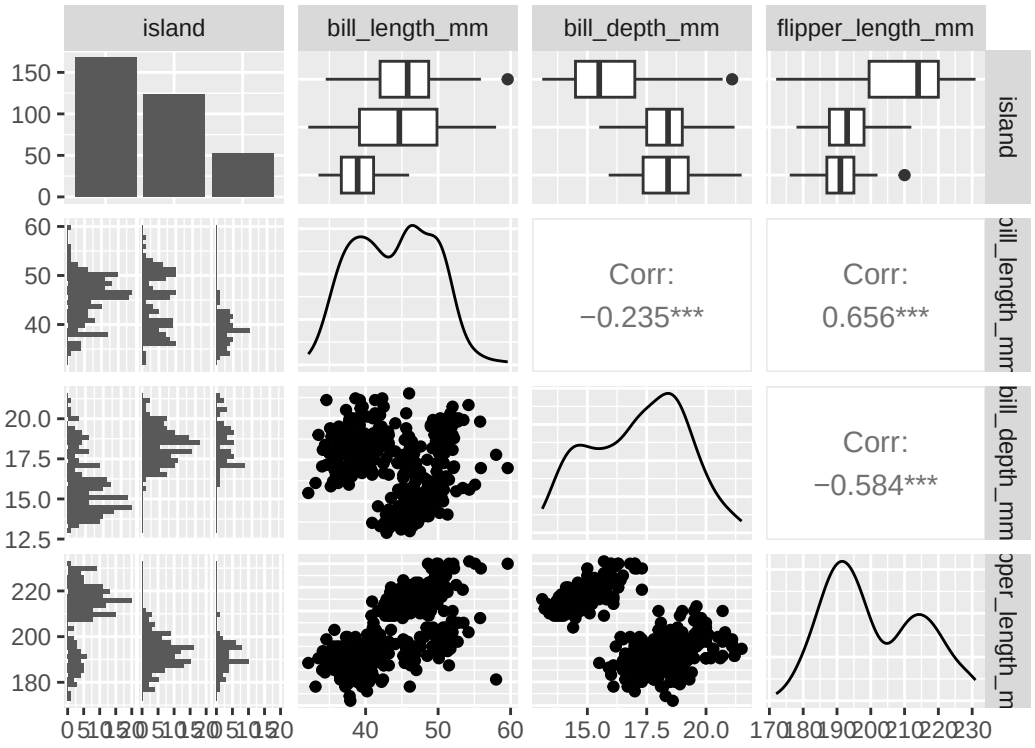
```
# Inspect skimr package (and vignette)
# vignette("skimr")
skimr::skim(df)
```

```
--- Data Summary ------------------------
                           Values
Name                       penguins
Number of rows             344
Number of columns          8
-----------------------
Column type frequency:
  factor                   3
  numeric                  5
-------------------------
Group variables            None

--- Variable type: factor -------------------------------------------------------
  skim_variable n_missing complete_rate ordered n_unique
1 species               0         1     FALSE          3
2 island                0         1     FALSE          3
3 sex                  11         0.968 FALSE          2

  top_counts
```

```
1 Ade: 152, Gen: 124, Chi: 68
2 Bis: 168, Dre: 124, Tor: 52
3 mal: 168, fem: 165

--- Variable type: numeric
------------------------------------------------------------
  skim_variable     n_missing complete_rate   mean      sd     p0    p25    p50
1 bill_length_mm            2         0.994   43.9   5.46    32.1   39.2   44.4
2 bill_depth_mm             2         0.994   17.2   1.97    13.1   15.6   17.3
3 flipper_length_mm         2         0.994  201.   14.1    172    190    197
4 body_mass_g               2         0.994 4202.  802.    2700   3550   4050
5 year                      0         1     2008.    0.818 2007   2007   2008
     p75   p100
1   48.5   59.6
2   18.7   21.5
3  213    231
4 4750   6300
5 2009   2009
```

Or examine the `ggpairs()` function from the `GGally` package (Schloerke et al. 2021). It provides many extensions to create graphs (with `ggplot2` see Chapter 7); and it also has functions to explore data and effects. The `ggpairs()` function returns a graph for a pairwise comparison of all variables. Depending on the data type, it returns bar plots, density plot, or the correlation between variables and combines all plots in one graph.

```
# GGally: https://ggobi.github.io/ggally/
GGally::ggpairs(df[2:5])
```



2.4 Summary

Data exploration can be exciting since we explore something new. Unfortunately, it can be painful if the data is complex or messy. For this reason we used a simple and clean data, but we will start to manipulate complex(er) data and prepare messy data soon. Keep the following functions from Chapter 3 in mind:

- Get a glimpse of your data (`dplyr::glimpse`); display the structure of an object (`str`); and inspect the first or last parts of an object (`head/tail`)
- Create a factor variable (`factor`); levels attributes (`levels`); object labels (`labels`)
- Simple cross table (`table`)
- Get a summary (`summary`)
- Summary statistics (`min, mean, max, sd`)
- Correlation, variance and covariance (matrices) via (`cor`); or with the `correlation` package (Makowski et al. 2022)

- Graphs: Bar plots (`barplot`); histograms (`hist`), spine plot (`spineplot`), box plot (`boxplot`), scatter plot (`plot`), correlation matrix (`corrplot::corrplot`)
- Packages:
  - The `summarytools` package provides many tables: (e.g., `freq, ctable`)
  - The `DataExplorer` to visualize several variable at once: (e.g., `plot_bar`)
  - The `effectsize` package to interpret results: (e.g., `interpret_r`)

2.5 References

References

Ben-Shachar, Mattan S., Dominique Makowski, Daniel Lüdecke, Indrajeet Patil, and Brenton M. Wiernik. 2022. *effectsize: Indices of Effect Size*. https://CRAN.R-project.org/package=effectsize.

Comtois, Dominic. 2022. *summarytools: Tools to Quickly and Neatly Summarize Data*. https://CRAN.R-project.org/package=summarytools.

Cui, Boxuan. 2020. *DataExplorer: Automate Data Exploration and Treatment*. https://CRAN.R-project.org/package=DataExplorer.

Horst, Allison, Alison Hill, and Kristen Gorman. 2022. *palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. https://CRAN.R-project.org/package=palmerpenguins.

Makowski, Dominique, Brenton M. Wiernik, Indrajeet Patil, Daniel Lüdecke, and Mattan S. Ben-Shachar. 2022. *Correlation: Methods for Correlation Analysis*. https://CRAN.R-project.org/package=correlation.

Müller, Kirill, and Hadley Wickham. 2022a. *pillar: Coloured Formatting for Columns*. https://CRAN.R-project.org/package=pillar.

———. 2022b. *tibble: Simple Data Frames*. https://CRAN.R-project.org/package=tibble.

Schloerke, Barret, Di Cook, Joseph Larmarange, Francois Briatte, Moritz Marbach, Edwin Thoen, Amos Elberg, and Jason Crowley. 2021. *GGally: Extension to ggplot2*. https://CRAN.R-project.org/package=GGally.

Tierney, Nicholas, Di Cook, Miles McBain, and Colin Fay. 2021. *naniar: Data Structures, Summaries, and Visualisations for Missing Data*. https://CRAN.R-project.org/package=naniar.

Treischl, Edgar J. 2023. *Practice R: An Interactive Textbook*. De Gruyter Oldenbourg.

Waring, Elin, Michael Quinn, Amelia McNamara, Eduardo Arino de la Rubia, Hao Zhu, and Shannon Ellis. 2022. *skimr: Compact and Flexible Summaries of Data*. https://CRAN.R-project.org/package=skimr.

Wickham, Hadley, Romain François, Lionel Henry, and Kirill Müller. 2022. *dplyr: A Grammar of Data Manipulation*. https://CRAN.R-project.org/package=dplyr.