The Practice R Tutorials

Edgar J. Treischl

2024-11-20

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.

```r
# 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?

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

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

```r
# 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?

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

a + b
```

```
#> Error in parse(text = input): <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 in parse(text = input): <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
# R cannot find an object due to typos
mean_a <- mean(1, 2, 3)
maen_a


#> Error: 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
# 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))
```

8

```
#> [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, François, 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.

```r
# 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
```

```r
result
```

```
#> Error: 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?

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

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

```r
# 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 in parse(text = input): <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 in parse(text = input): <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?

```r
# 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?

```r
# 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
```

```r
# 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, François, 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)
```
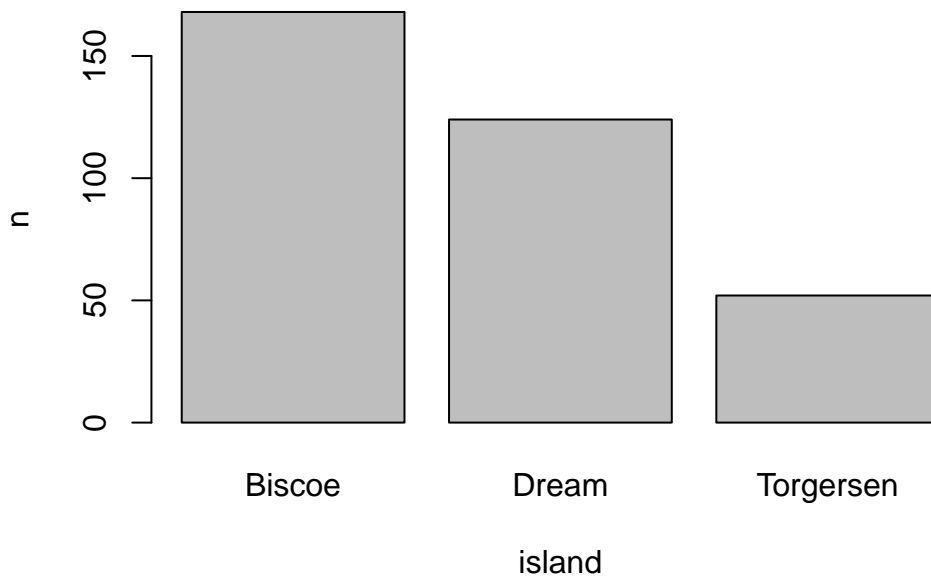

In a similar vein, I introduced functions from the `DataExplorer` package that help us to get a quick overview (Cui 2020). For example, use the `plot_bar()` function to depict several or all discrete variables of a data frame.

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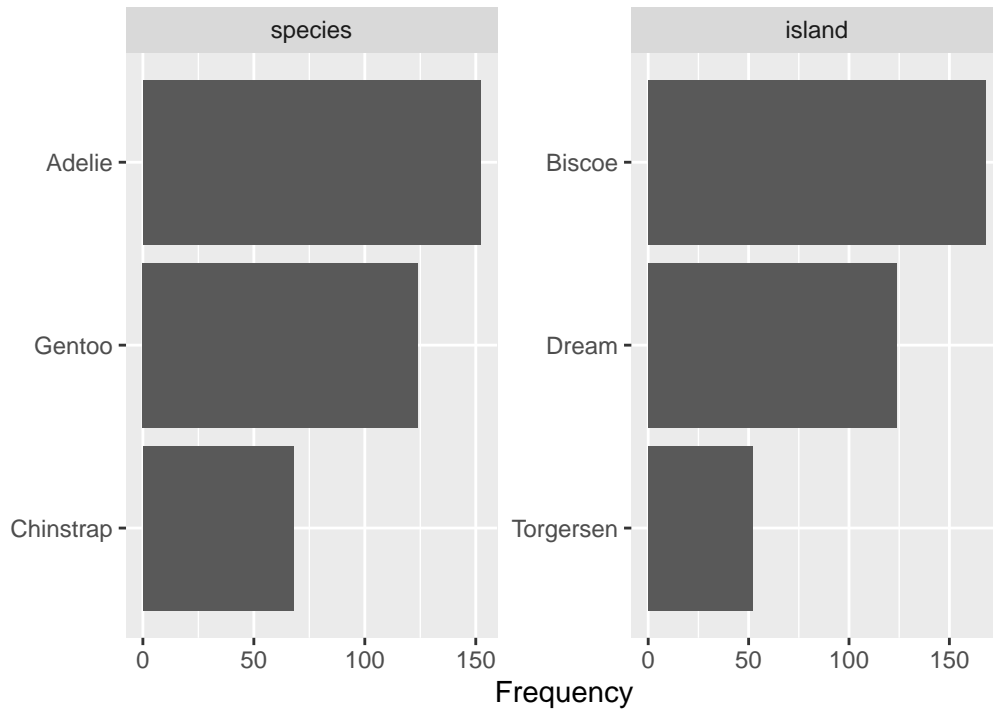
2.2 Continuous variables

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)
```

Histogram of df$body_mass_g
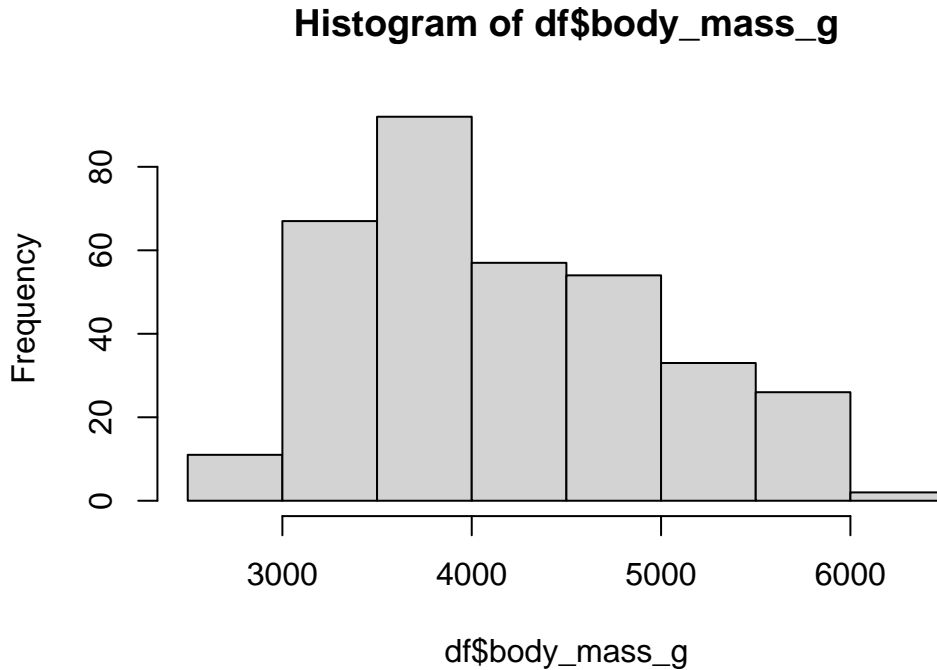
Keep in mind that we only explored the data for the first time. We did not clean the data nor did we prepare the variables. We have to be explicit about missing values when we want to apply functions such as the `mean`. The function returns `NA`, but only because of a missing values problem. Can you 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.

```r
library(visdat)

# Left plot: vis_dat()
vis_dat(df)

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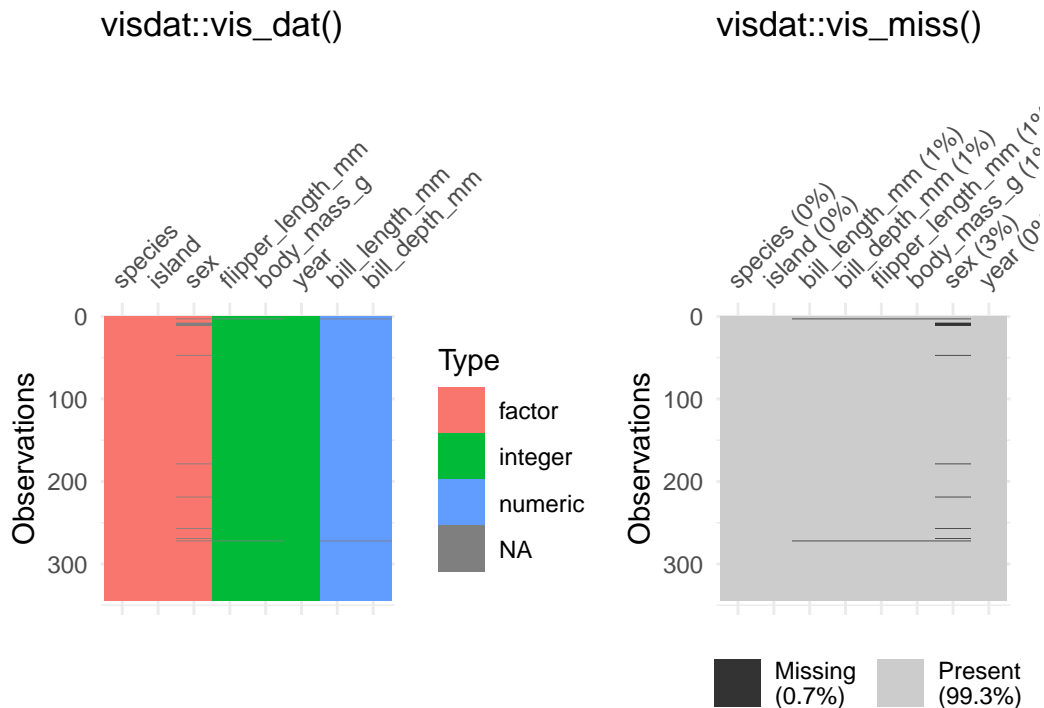

2.3 Explore effects

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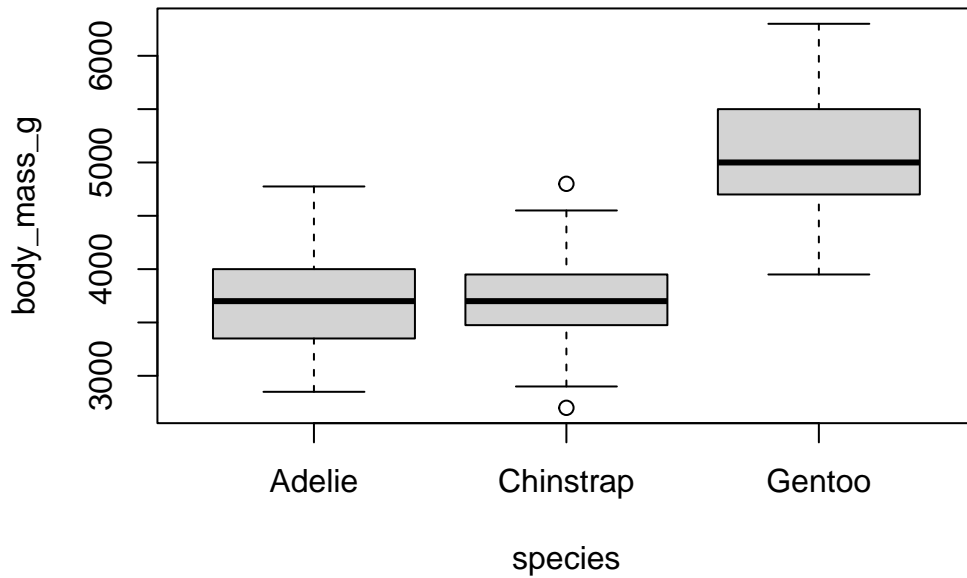
If we examine an *effect between two continuous outcomes*, we have to keep in mind that the `plot` function returns a scatter plot and we may insert a regression line with the `abline` and the `lm` function. Do you still know how it works? Create a scatter plot to examine the association between the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

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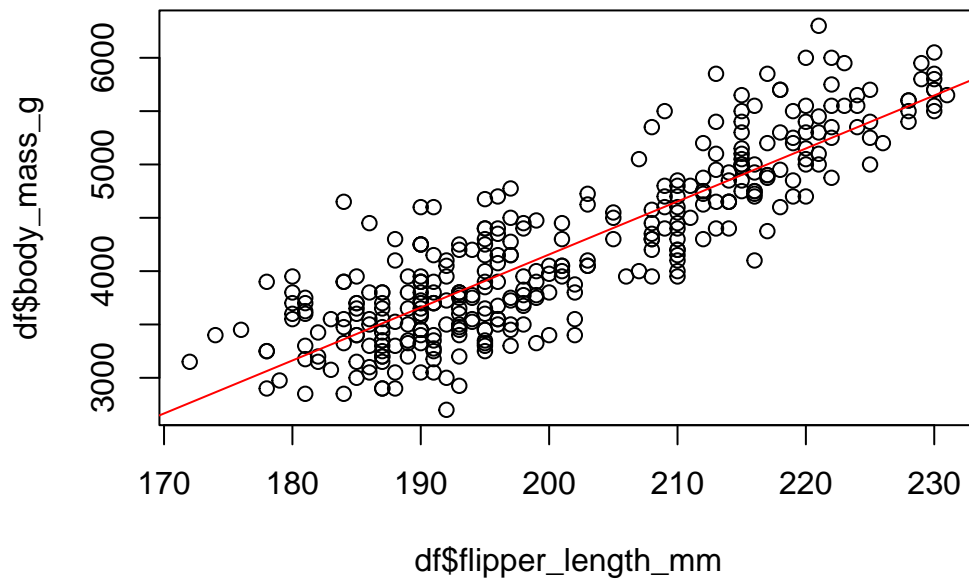
Furthermore, we learned how to calculate the correlation coefficient. The code of the next console does not work if I apply the `cor()` with the penguins data. Do you have any idea how to fix the problem?

```r
# 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:

```r
# 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])
```
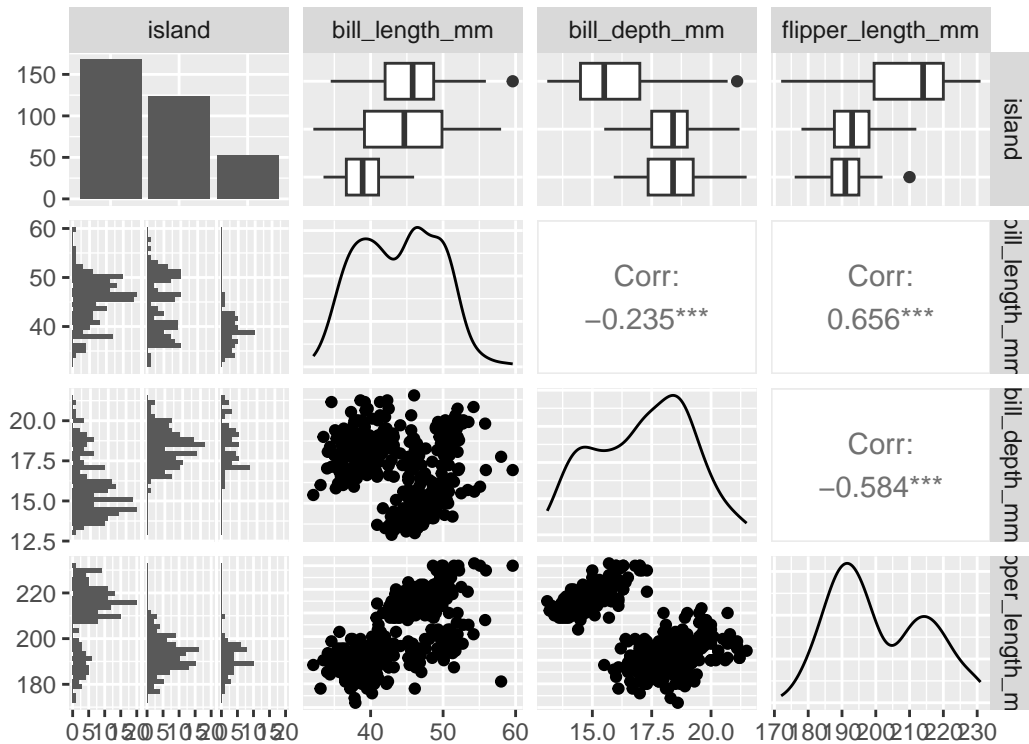
2.4 Summary

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, Wiernik, 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`)

3 Data manipulation with dplyr

Welcome to 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 Chapter 4 we used the `dplyr` package to manipulate data (Wickham, François, et al. 2022). In addition, we will learn how to systemically manipulate categorical variables with the `forcats` package (Wickham 2022a) in Chapter 5. Both packages help you to handle many common steps to manipulate data. This tutorial gives a `dplyr` recap and asks you to apply the introduced functions.

As the next output shows, we use the `gss2016` again to select variables, create a filter, generate new variables, and summarize the data. Ask R to provide a description of the data (`?data`) if you are not familiar with the `gss2016` data yet.

```r
# The setup of tutorial 4
library(dplyr)
library(PracticeR)
head(gss2016)[1:9]
```

```
#> # A tibble: 6 x 9
#>    year    id ballot      age childs sibs  degree race  sex
#>   <dbl> <dbl> <labell> <dbl>  <dbl> <lab> <fct>  <fct> <fct>
#> 1  2016     1 1           47      3 2     Bache~ White Male
#> 2  2016     2 2           61      0 3     High ~ White Male
#> 3  2016     3 3           72      2 3     Bache~ White Male
#> 4  2016     4 1           43      4 3     High ~ White Fema~
#> 5  2016     5 3           55      2 2     Gradu~ White Fema~
#> # i 1 more row
```

3.1 Select

Especially in case of large and cluttered data, we use `select()` to specify which variables we work with. For example, pick only one variable such as school `degree` from the `gss2016` data.

```
# Select a variable
select(gss2016, degree)
```

```
#> # A tibble: 2,867 x 1
#>    degree
#>    <fct>
#> 1 Bachelor
#> 2 High School
#> 3 Bachelor
#> 4 High School
#> 5 Graduate
#> # i 2,862 more rows
```

Select comes with handy functions and applies the same logic as base R. For example, select several columns by providing a start (e.g., id) and endpoint (e.g., degree).

```
# Select all variables from x to y
select(gss2016, id:degree) |> head()
```

```
#> # A tibble: 6 x 6
#>      id ballot        age childs sibs        degree
#>   <dbl> <labelled> <dbl>  <dbl> <labelled> <fct>
#> 1     1 1             47      3 2           Bachelor
#> 2     2 2             61      0 3           High School
#> 3     3 3             72      2 3           Bachelor
#> 4     4 1             43      4 3           High School
#> 5     5 3             55      2 2           Graduate
#> # i 1 more row
```

Maybe we need all columns except the variables shown in the last output. Ask for the opposite and insert parentheses and a minus signs to turn the selection around.

```
# Turn around the selection
select(gss2016, -(id:degree)) |> head()
```

```
#> # A tibble: 6 x 27
#>    year race  sex    region     income16 relig marital padeg
#>   <dbl> <fct> <fct>  <fct>      <fct>    <fct> <fct>   <fct>
#> 1  2016 White Male   New Engla~ $170000~ None  Married Grad~
#> 2  2016 White Male   New Engla~ $50000 ~ None  Never ~ Lt H~
```

```
#> 3  2016 White Male    New Engla~ $75000 ~ Cath~ Married High~
#> 4  2016 White Female New Engla~ $170000~ Cath~ Married <NA>
#> 5  2016 White Female New Engla~ $170000~ None  Married Bach~
#> # i 1 more row
#> # i 19 more variables: madeg <fct>, partyid <fct>,
#> #   polviews <fct>, happy <fct>, partners <fct>,
#> #   grass <fct>, zodiac <fct>, pres12 <labelled>,
#> #   wtssall <dbl>, income_rc <fct>, agegrp <fct>,
#> #   ageq <fct>, siblings <fct>, kids <fct>, religion <fct>,
#> #   bigregion <fct>, partners_rc <fct>, obama <dbl>, ...
```

The `gss2016` data does not contain variables with a running number nor other systematic variable names. However, `dplyr` helps to select such variables without much effort. Consider toy data with several measurements and running numbers to illustrate how we can select such variables efficiently.

```
# A new df to illustrate
df <- tibble(
  measurement_1 = 1:3,
  x1 = 1:3,
  measurement_2 = 1:3,
  x2 = 1:3,
  x3 = 1:3,
  other_variables = 1
)
```

Suppose we measured a variables several times and all start with an identical name (e.g., `measurement_`). Select all variables which start (or end) with a certain string. Thus, insert the `starts_with()` function and select all `measurement` variables.

```
# Select variables that start with a string
select(df, starts_with("measurement"))
```

```
#> # A tibble: 3 x 2
#>   measurement_1 measurement_2
#>           <int>         <int>
#> 1             1             1
#> 2             2             2
#> 3             3             3
```

Or pick variables with the running number. The `num_range` functions needs the name (`x`) and the running number.

```
# Select based on a running number
select(df, num_range("x", 1:3))
```

```
#> # A tibble: 3 x 3
#>      x1    x2    x3
#>   <int> <int> <int>
#> 1     1     1     1
#> 2     2     2     2
#> 3     3     3     3
```

The package offers more helpers to select variables than I can possibly outline. For example, `contains()` checks if a variable includes a certain word; `matches()` let us specify search patterns (regular expression, see Chapter 10); and we can also include other functions to select variables. For example, the `is.numeric` function checks if an input is numeric and we can combine it with `where()` to select columns only *where* the content is numeric.

```
# Insert a function to select variables
gss2016 |> select(where(is.numeric))
```

```
#> # A tibble: 2,867 x 10
#>     year    id ballot   age childs sibs  pres12 wtssall obama
#>    <dbl> <dbl> <labe> <dbl>  <dbl> <lab> <labe>   <dbl> <dbl>
#> 1   2016     1 1         47      3 2     3        0.957     0
#> 2   2016     2 2         61      0 3     1        0.478     1
#> 3   2016     3 3         72      2 3     2        0.957     0
#> 4   2016     4 1         43      4 3     2        1.91      0
#> 5   2016     5 3         55      2 2     1        1.44      1
#> # i 2,862 more rows
#> # i 1 more variable: income <dbl>
```

Next, we filter data but since all R outputs are large due to the `gss2016` data, let us first create a smaller subset to reduce the size of the output and the length of this document.

```
# Select a smaller subset for the rest of this tutorial
gss2016 <- select(PracticeR::gss2016, year:sex, income)
```

3.2 Filter

Use `filter()` to subset the data. The `dplyr` filters the data and returns a new data frame depending on the specified conditions. Use one or several relational or logical operators to select observations. For example, suppose you want to analyze persons who have a bachelor's `degree` only.

```
# Apply a filter
gss2016 |>
  filter(degree == "Bachelor") |>
  head()
```

```
#> # A tibble: 6 x 10
#>    year    id ballot    age childs sibs  degree race  sex
#>   <dbl> <dbl> <labell> <dbl>  <dbl> <lab> <fct>  <fct> <fct>
#> 1  2016     1 1          47      3 2     Bache~ White Male
#> 2  2016     3 3          72      2 3     Bache~ White Male
#> 3  2016    37 2          59      2 2     Bache~ White Male
#> 4  2016    38 1          43      2 6     Bache~ White Fema~
#> 5  2016    39 3          58      0 1     Bache~ White Fema~
#> # i 1 more row
#> # i 1 more variable: income <dbl>
```

Can you adjust the code so that two conditions have to be fulfilled simultaneously. For example, keep only observations from adults (18 years and older) with a bachelor's degree.

```
# Combine several conditions
gss2016 |>
  filter(degree == "Bachelor" & age > 17) |>
  head()
```

```
#> # A tibble: 6 x 10
#>    year    id ballot    age childs sibs  degree race  sex
#>   <dbl> <dbl> <labell> <dbl>  <dbl> <lab> <fct>  <fct> <fct>
#> 1  2016     1 1          47      3 2     Bache~ White Male
#> 2  2016     3 3          72      2 3     Bache~ White Male
#> 3  2016    37 2          59      2 2     Bache~ White Male
#> 4  2016    38 1          43      2 6     Bache~ White Fema~
#> 5  2016    39 3          58      0 1     Bache~ White Fema~
#> # i 1 more row
#> # i 1 more variable: income <dbl>
```

As outlined, keep your `base` R skills in mind when selecting or filtering data. For example, keep all degrees but exclude persons who have a `Bachelor`.

```
# All degrees, but not! Bachelors
gss2016 |>
  filter(degree != "Bachelor") |>
  head()
```

```
#> # A tibble: 6 x 10
#>    year    id ballot      age childs sibs  degree race  sex
#>   <dbl> <dbl> <labell> <dbl>  <dbl> <lab> <fct>  <fct> <fct>
#> 1  2016     2 2           61      0 3     High ~ White Male
#> 2  2016     4 1           43      4 3     High ~ White Fema~
#> 3  2016     5 3           55      2 2     Gradu~ White Fema~
#> 4  2016     6 2           53      2 2     Junio~ White Fema~
#> 5  2016     7 1           50      2 2     High ~ White Male
#> # i 1 more row
#> # i 1 more variable: income <dbl>
```

Use the `operators()` function from the `PracticeR` package when you have trouble to remember how `logical` and `relational` operators are implemented. The function inserts and runs examples via the console.

```
PracticeR::operators("logical")
#   Logical Operators
# > x <- TRUE
# > y <- FALSE
# > #Elementwise logical AND
# > x & y == TRUE
# [1] FALSE
# > #Elementwise logical OR
# > x | y == TRUE
# [1] TRUE
# > #Elementwise OR
# > xor(x, y)
# [1] TRUE
# > #Logical NOT
# > !x
# [1] FALSE
# > #In operator
# > 1:3 %in% rep(1:2)
# [1]  TRUE  TRUE FALSE
```

3.3 Mutate

In Chapter 4 I outline several ways to generate new variables based on observed ones. For example, raw data often contains a person's year of birth but not their age. With `mutate()` we can extend the data frame and estimate such a variable. Unfortunately, the `gss2016` has an `age` variable, but the variable does only reveal their age when the survey was conducted. To recap how `mutate()` works, recreate their birth year and a recent age variable, say for the year 2023.

```
# Create birth_year and a recent (year: 2023) age variable
gss2016 |>
  select(id, year, age) |>
  mutate(
    birth_year = year - age,
    age_2023 = 2023 - birth_year
  ) |>
  head()
```

```
#> # A tibble: 6 x 5
#>      id  year   age birth_year age_2023
#>   <dbl> <dbl> <dbl>      <dbl>    <dbl>
#> 1     1  2016    47       1969       54
#> 2     2  2016    61       1955       68
#> 3     3  2016    72       1944       79
#> 4     4  2016    43       1973       50
#> 5     5  2016    55       1961       62
#> # i 1 more row
```

Keep in mind that you can use relational and logical operators, as well other functions (e.g., log, rankings, etc.) to generate new variables. For example, generate a logical variable that indicates whether a person was an adult (older than 17) in the year 2016. The `if_else()` function helps you with this job.

```
# In theory: if_else(condition, true, false, missing = NULL)
gss2016 |>
  select(id, year, age) |>
  mutate(adult = if_else(age > 17, TRUE, FALSE)) |>
  head()
```

```
#> # A tibble: 6 x 4
#>      id  year   age adult
```

```
#>    <dbl> <dbl> <dbl> <lgl>
#> 1      1  2016    47 TRUE
#> 2      2  2016    61 TRUE
#> 3      3  2016    72 TRUE
#> 4      4  2016    43 TRUE
#> 5      5  2016    55 TRUE
#> # i 1 more row
```

In terms of generating new variables, also keep the `case_when()` function in mind, which provides a very flexible approach. Suppose we need to identify parents with a academic background. Parents educational background has many levels or attributes in the `gss2016` data, which makes a first attempt harder to apply (and we learn more about factor variables in Chapter 5). For this reason I created a smaller toy data set and I started to prepare the code. Can you complete it? The variable `academic_parents` is supposed to identify persons with a high educational background (`education`) with one or more `kids`. All other conditions are set to `FALSE`.

```
# Data to illustrate
df <- data.frame(
  kids = c(0, 1, 3, 0, NA),
  educ = c("high", "low", "high", "low", NA)
)

# In theory: case_when(condition ~ value)
df |>
  mutate(academic_parents = case_when(
    kids >= 1 & educ == "high" ~ "TRUE",
    TRUE ~ "FALSE"
  ))
```

```
#>   kids educ academic_parents
#> 1    0 high            FALSE
#> 2    1  low            FALSE
#> 3    3 high             TRUE
#> 4    0  low            FALSE
#> 5   NA <NA>            FALSE
```

3.4 Summarize

The `summarize()` function collapses several columns into a single row. By the way, the `dplyr` package understands both, British (e.g., `summarise`) and American English (e.g. `summarize`)

and it's up to you to decide which one you prefer.

Let's calculate the mean age of the survey participants. As outlined in Practice R, the variable has missing values which is why we need to drop them first. In Chapter 5 we will focus on this problem and we learn more about the consequences of such decisions. I already excluded missing values, can you `summarize()` the age?

```
# Exclude missing values but consider the consequences (see Chapter 5)
gss2016 <- gss2016 |>
  tidyr::drop_na(age, sex)

# Summarize age
gss2016 |> summarize(mean_age = mean(age))
```

```
#> # A tibble: 1 x 1
#>   mean_age
#>      <dbl>
#> 1     49.2
```

The `dplyr` package comes with several help functions to summarize data. For example, to count the number of observation per group (e.g., for `sex`), split the data by groups (`group_by`) and apply the `n()` function.

```
# County by (sex)
gss2016 |>
  group_by(sex) %>%
  summarize(count = n())
```

```
#> # A tibble: 2 x 2
#>   sex     count
#>   <fct>   <int>
#> 1 Male     1272
#> 2 Female   1585
```

Moreover, compare the groups by calculating the `median` age instead of the mean; add the standard deviation (`sd`); and count the number of distinct values (`n_distinct`) of the `degree` variable.

41

```
# Dplyr has more summary functions
gss2016 |>
  group_by(sex) |>
  summarise(
    median_age = median(age),
    sd_age = sd(age),
    distinct_degree = n_distinct(degree)
  )
```

```
#> # A tibble: 2 x 4
#>   sex    median_age sd_age distinct_degree
#>   <fct>       <dbl>  <dbl>           <int>
#> 1 Male           48   17.4               6
#> 2 Female         50   17.9               6
```

In the last examples we grouped the data and then collapsed it. The counterpart to **group** is **ungroup()** which we may add as a last step to disperse the data again. For example, we can estimate how old men or women are on average and add this information to the original data frame. Use **mutate()** instead of **summarise()** to see the logic behind **ungroup**.

```
# Mutate ungroups the data again
gss2016 |>
  select(id, sex, age) |>
  group_by(sex) |>
  mutate(count = round(mean(age), 2))
```

```
#> # A tibble: 2,857 x 4
#> # Groups:   sex [2]
#>      id sex      age count
#>   <dbl> <fct>  <dbl> <dbl>
#> 1     1 Male      47  48.3
#> 2     2 Male      61  48.3
#> 3     3 Male      72  48.3
#> 4     4 Female    43  49.8
#> 5     5 Female    55  49.8
#> # i 2,852 more rows
```

3.5 Arrange

Last but not least, keep the **arrange()** function in mind. It is easy to apply and I don't believe there is much to practice. However, it gives us the chance to repeat how **transmute()**

and the `between()` function works.

Consider the steps to build a restricted age sample to examine adults only. Use `mutate` to create a logical variable (`age_filter`) that indicates if a person is between 18 and 65. Furthermore, explore the difference between `mutate()` and `transmute()` if you can't remember it.

```
# Create a restricted analysis sample
# between: x >= left & x <= right
gss2016 |>
  transmute(age,
    age_filter = between(age, 18, 65)
  )
```

```
#> # A tibble: 2,857 x 2
#>     age age_filter
#>   <dbl> <lgl>
#> 1    47 TRUE
#> 2    61 TRUE
#> 3    72 FALSE
#> 4    43 TRUE
#> 5    55 TRUE
#> # i 2,852 more rows
```

Next, we need a `filter()` to restrict the sample, but how can we know that code worked? We can inspect the entire data frame with `View`, but we can also use `arrange()` to inspect if the filter was correctly applied. Sort in ascending and descending (`desc`) order.

```
# Filter and arrange the data
gss2016 |>
  transmute(age,
    age_filter = between(age, 18, 65)
  ) |>
  filter(age_filter == "TRUE") |>
  arrange(desc(age)) |>
  head()
```

```
#> # A tibble: 6 x 2
#>     age age_filter
#>   <dbl> <lgl>
#> 1    65 TRUE
#> 2    65 TRUE
#> 3    65 TRUE
```

```
#> 4    65 TRUE
#> 5    65 TRUE
#> # i 1 more row
```

The `dplyr` package offers many functions to manipulate data and this tutorial only summarizes the main functions. Consider the cheat sheet and the package website for more information.

```
# The dplyr website
PracticeR::show_link("dplyr", browse = FALSE)
#> [1] "https://dplyr.tidyverse.org/"
```

Keep in mind that data preparation steps may appear simple, but only as long as we are not supposed to prepare data on our own. In the latter case we will often need several attempts to come up with a solution that works. Thus, be patient with yourself when your first attempts will not work. Most of the time we all need more than one shot to come up with a workable solution. In addition, we will use the package one more time to combine data in Chapter 5 and other `dplyr` functions will appear through the Practice R book. Thus, there will be plenty of opportunities to apply and develop your `dplyr` skills.

There are often different approaches that lead to the same result. As the artwork by Jake Clark illustrates and the Practice R info box about *data manipulation approaches* underlines, the `subset()` function from `base` R does essentially the same as `dplyr::filter`. Base R provides the most stable solution, while `dplyr` is more verbose and often easier to learn. Don't perceive them as two different dialects that forces us to stick to one approach. Instead, embrace them both because you will come across different approaches if you use Google to solve a problem. Fortunately, many roads lead to Rome.

3.6 Summary

Keep the main `dplyr` functions in mind, among them:

- Keep rows that match a condition (`filter`)
- Order rows using column values (`arrange`)
- Keep or drop columns using their names and types (`select`)
- Create, modify, and delete columns (`mutate, transmute`)
- Summarize each group down to one row (`summarize`)
- Change column order (`relocate`)
- Vectorized if-else (`if_else`)
- A general vectorized if-else (`case_when`)
- Apply a function (or functions) across multiple columns (`across`)
- Select all variables or the last variable (e.g., `everything`)

And the following `base` functions:

- The names of an object (`names`)
- Sub-setting vectors, matrices and data frames (`subset`)
- Apply a function over a list or vector (`lapply, sapply`)
- Read R code from a file, a connection or expressions (`source`)

4 Prepare categorical variables

Welcome to the data preparation 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.

Chapter 5 was dedicated to support you to prepare data. We learned how to import, clean, and combine data. In addition, we got in touch with the `naniar` package which offers many functions to inspect missing values (Tierney et al. 2021); and I introduced the `forcats` package to prepare categorical variables for the analysis (Wickham 2022a).

What preparation steps you need to apply is dependent on the data at hand and the analysis intended, which is why Chapter 5 provided a detailed overview of what happens under the hood when we import data. Keep in mind that RStudio has many cool features (e.g., data preview) to import data and packages such as `readr` helps us with this task:

```
# Import a csv file
library(readr)
my_data <- read_csv("path_to_the_file/data.csv")
```

Since I have no idea what your data looks, this tutorial will not focus on how to import and clean data. Instead, let's focus systematically on the `forcats` package. Suppose we started to analyze whether participant's income has an effect on their happiness, but we need to control for participant's educational background, religious beliefs, and if other categorical variables affect our estimation results. I already introduced several functions of the `forcats` package, but this tutorial systematically focuses on the main tasks of the package, as is outlined in its cheat sheet (click on the hex sticker to download the cheat sheet from the website).

Thus, we repeat and systematize our forcats skills: (1) We inspect factors; (2) change the order of levels; (3) change the value of levels; (4) and we add or drop levels. For this purpose, we use the `gss2016` data and I assigned a smaller subset as `df` with several categorical variables.

```
# Packages for Tutorial Nr. 5
library(naniar)
library(dplyr)
library(tidyr)
library(forcats)
```

```r
library(PracticeR)

# The gss2016 data
df <- PracticeR::gss2016 |>
  select(id, degree, relig, income16, happy, marital)
head(df)
```

```
#> # A tibble: 6 x 6
#>      id degree        relig   income16        happy        marital
#>   <dbl> <fct>         <fct>   <fct>           <fct>        <fct>
#> 1     1 Bachelor      None    $170000 or over Pretty Happy Married
#> 2     2 High School   None    $50000 to 59999 Pretty Happy Never Married
#> 3     3 Bachelor      Catholic $75000 to $89999 Very Happy  Married
#> 4     4 High School   Catholic $170000 or over Pretty Happy Married
#> 5     5 Graduate      None    $170000 or over Very Happy   Married
#> 6     6 Junior College None    $60000 to 74999 Very Happy   Married
```

Finally, we transform and combine data once more given that such steps are often necessary before we can start to prepare data. However, this time we examine how built-in data sets from the `tidyr` and the `dplyr` package make the first move a bit easier.

4.1 Inspect factors

Suppose we need to prepare several categorical variables, such as religion (`relig`) or marital status (`marital`), for an analysis. To inspect factors, count them with `fct_count()`.

```r
# Count factor variable
fct_count(df$marital)
```

```
#> # A tibble: 6 x 2
#>   f               n
#>   <fct>        <int>
#> 1 Married       1212
#> 2 Widowed        251
#> 3 Divorced       495
#> 4 Separated      102
#> 5 Never Married  806
#> 6 <NA>             1
```

Or examine the unique levels of a variable with the `fct_unique()` function:

```
# How many unique levels do we observe
fct_unique(df$marital)
```

```
#> [1] Married        Widowed        Divorced       Separated      Never Married
#> [6] <NA>
#> Levels: Married Widowed Divorced Separated Never Married
```

4.2 Change the order of levels

The variable religion (`relig`) has 13 different levels. Let's assume we want to control for the largest religious groups only in the analysis. Use the `fct_infreq()` function to identify how often each level appears.

```
# fct_infreq: Reorder factor levels by frequency
f <- fct_infreq(df$relig)
fct_count(f)
```

```
#> # A tibble: 14 x 2
#>    f                          n
#>    <fct>                  <int>
#>  1 Protestant              1371
#>  2 Catholic                 649
#>  3 None                     619
#>  4 Jewish                    51
#>  5 Other                     44
#>  6 Christian                 40
#>  7 Buddhism                  21
#>  8 Moslem/Islam              19
#>  9 Hinduism                  13
#> 10 Orthodox-Christian         7
#> 11 Inter-Nondenominational    7
#> 12 Other Eastern              4
#> 13 Native American            4
#> 14 <NA>                      18
```

The `fct_infreq()` sorts them in order of their frequency, but note we can also order the levels by first appearance (`fct_inorder`) or in a numeric order (`fct_inseq`). As the next console illustrates, R sorts levels alphabetically, which is clearly not always a desirable default behavior. Use the `fct_inorder()` to sort them by appearance.

```
# Example factor
f <- factor(c("b", "a", "c"))
levels(f)
```

```
#> [1] "a" "b" "c"
```

```
# fct_inorder: Reorder factor levels by first appearance
fct_inorder(f)
```

```
#> [1] b a c
#> Levels: b a c
```

Can you still remember how to manually relevel? Use the `fct_relevel()` and sort the level `Never Married` at the second position. You can provide a vector with level names or use the `after` option to change the position of the level.

```
# Relevel manually
# f <- fct_relevel(df$marital, c("Married", "Never Married"))
f <- fct_relevel(df$marital, "Never Married", after = 1)
fct_count(f)
```

```
#> # A tibble: 6 x 2
#>   f                 n
#>   <fct>         <int>
#> 1 Married        1212
#> 2 Never Married   806
#> 3 Widowed         251
#> 4 Divorced        495
#> 5 Separated       102
#> 6 <NA>              1
```

Sometimes we need to turn the order around. Reverse the order of the levels with `fct_rev()`.

```
# fct_rev: Reverse order of factor levels
f <- fct_rev(df$marital)
fct_count(f)
```

```
#> # A tibble: 6 x 2
#>   f                  n
#>   <fct>          <int>
#> 1 Never Married    806
#> 2 Separated        102
#> 3 Divorced         495
#> 4 Widowed          251
#> 5 Married         1212
#> 6 <NA>               1
```

4.3 Change the value of levels

The `relig` variable has many levels and even has a category named `other`, since there are so many religious groups. The same logic applies the `fct_other()` function which collapses all levels but the one we actually need. Create a variable that includes the five largest groups only. Use the `fct_other()` function and tell R which variables to `keep`.

```r
# Create a variable with the five largest, rest are others
df$relig5 <- fct_other(df$relig,
  keep = c("Protestant", "Catholic", "None", "Jewish")
)

fct_count(df$relig5)
```

```
#> # A tibble: 6 x 2
#>   f              n
#>   <fct>      <int>
#> 1 Protestant  1371
#> 2 Catholic     649
#> 3 Jewish        51
#> 4 None         619
#> 5 Other        159
#> 6 <NA>          18
```

The `fct_other()` function includes in the code the used levels. If we are unconcerned about this information, you can use one of the `fct_lump()` functions. The function picks between different methods to lump together factor levels. Nowadays the authors recommend to use one of the specific `fct_lump_*` functions (`fct_lump_min`, `fct_lump_prop`, `fct_lump_lowfreq`) as outlined in the help file. In our case, use the `fct_lump_n()` function to lump together the most frequent (`n`) ones.

```
# Lump uncommon factor together levels into "other"
f <- fct_lump_n(df$relig, n = 5, other_level = "Further groups")
fct_count(f)
```

```
#> # A tibble: 7 x 2
#>   f                   n
#>   <fct>           <int>
#> 1 Protestant       1371
#> 2 Catholic          649
#> 3 Jewish             51
#> 4 None              619
#> 5 Other              44
#> 6 Further groups    115
#> 7 <NA>               18
```

Next, we are going to prepare the educational background. The variable `degree` includes several levels, as the console shows.

```
# Count degrees
fct_count(df$degree)
```

```
#> # A tibble: 6 x 2
#>   f                   n
#>   <fct>           <int>
#> 1 Lt High School    328
#> 2 High School      1461
#> 3 Junior College    216
#> 4 Bachelor          536
#> 5 Graduate          318
#> 6 <NA>                8
```

We already used the `fct_recode()` function to change factor levels by hand. The lowest category of `degree` is called *less than high school* but the text label is confusing. Recode the variable, insert the new label in back ticks to replace the old label (`Lt High School`).

```
# fct_recode: Change factor levels by hand
f <- fct_recode(df$degree, `Less than high school` = "Lt High School")
fct_count(f)
```

```
#> # A tibble: 6 x 2
#>   f                      n
#>   <fct>              <int>
#> 1 Less than high school  328
#> 2 High School           1461
#> 3 Junior College         216
#> 4 Bachelor               536
#> 5 Graduate               318
#> 6 <NA>                     8
```

Suppose we want to control only if participants have a high educational background. Use the `fct_collapse()` function to create a binary dummy variable. The variable should indicate if a person's educational background is low (`Lt High School`; `High School`, and `Junior College`) or high (`Bachelor` and `Graduate`).

```r
# Collapse factor variable
df$edu_dummy <- fct_collapse(df$degree,
  low = c(
    "Lt High School",
    "High School",
    "Junior College"
  ),
  high = c("Bachelor", "Graduate")
)

fct_count(df$edu_dummy)
```

```
#> # A tibble: 3 x 2
#>   f        n
#>   <fct> <int>
#> 1 low    2005
#> 2 high    854
#> 3 <NA>      8
```

4.4 Add or drop levels

As always, the `forcats` package has more to offer than I can outline. For example, suppose we observed the following `religion` variable.

```r
# New religion variable
religion <- factor(
  x = c("Protestant", "Jewish", NA, NA),
  levels = c("Protestant", "Jewish", "Catholic")
)

religion
```

```
#> [1] Protestant Jewish     <NA>       <NA>
#> Levels: Protestant Jewish Catholic
```

Did you notice that the variable has a level for `Catholic` even though we do not observe it. The `fct_expand()` can be used to expand levels, while the `fct_drop()` function helps us to get rid of unused levels.

```r
# Drop unused levels
fct_drop(religion)
```

```
#> [1] Protestant Jewish     <NA>       <NA>
#> Levels: Protestant Jewish
```

Furthermore, I included missing values on purpose and the latter may have an impact on our analysis. We can make them explicit and include them as a level with `fct_na_value_to_level()`.

```r
# Make NAs explicit
fct_na_value_to_level(religion, level = "Missing")
```

```
#> [1] Protestant Jewish     Missing     Missing
#> Levels: Protestant Jewish Catholic Missing
```

4.5 Further steps

Chapter 5 discussed many steps to prepare data, but of course this was not an all-encompassing list. I introduced data formats and we learned how to combine data given that many official data sets are split into several files. Unfortunately, transforming and combining data can be tricky and we may introduce mistakes if we neglected to prepare and clean the data properly. Thus, it is up to you to assure that the data can be transformed (combined) and further cleaning steps might be necessary.

Instead of re-running these steps with the `gss2016` data, let us explore how the `tidyr` package can help with the task (Wickham and Girlich 2022). As other packages, `tidyr` has a cheat sheet and provides a tiny data set that lets us repeat how the functions work. For example, the `table4a` data is a wide data set with observations of three countries and two years.

```
# Example wide table
head(table4a)
```

```
#> # A tibble: 3 x 3
#>   country      `1999` `2000`
#>   <chr>         <dbl>  <dbl>
#> 1 Afghanistan     745   2666
#> 2 Brazil        37737  80488
#> 3 China        212258 213766
```

Use the `pivot_longer()` function to transform the data. The long data should have a new variable for the `year` (via `names_to`) and you can give the values (`values_to`) to a variable named `cases`.

```
# Make em longer
pivot_longer(table4a,
  cols = 2:3, names_to = "year",
  values_to = "cases"
)
```

```
#> # A tibble: 6 x 3
#>   country      year  cases
#>   <chr>        <chr> <dbl>
#> 1 Afghanistan  1999    745
#> 2 Afghanistan  2000   2666
#> 3 Brazil       1999  37737
#> 4 Brazil       2000  80488
#> 5 China        1999 212258
#> 6 China        2000 213766
```

Or consider the `table2` data, the variable `type` has two outcome types (`cases` and `population`) which underline why we should transform the data into the wide format.

```
# Example long table
head(table2)
```

```
#> # A tibble: 6 x 4
#>   country      year type          count
#>   <chr>       <dbl> <chr>         <dbl>
#> 1 Afghanistan  1999 cases           745
#> 2 Afghanistan  1999 population  19987071
#> 3 Afghanistan  2000 cases          2666
#> 4 Afghanistan  2000 population  20595360
#> 5 Brazil       1999 cases         37737
#> 6 Brazil       1999 population 172006362
```

Keep in mind that we need to provide *where* the names (`names_from`) and the values (`values_from`) are coming from to transform the data.

```
# Make it wider
pivot_wider(table2,
  names_from = type,
  values_from = count
)
```

```
#> # A tibble: 6 x 4
#>   country      year  cases population
#>   <chr>       <dbl>  <dbl>      <dbl>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
#> 3 Brazil       1999  37737  172006362
#> 4 Brazil       2000  80488  174504898
#> 5 China        1999 212258 1272915272
#> 6 China        2000 213766 1280428583
```

I introduced these data sets because `tidyr` offers such simple examples in the cheat sheet that demonstrates how we can transform data. In addition, the `copycat` package has the code snippets from the `tidyverse` cheat sheets included. As the animation shows, it only takes a few seconds to insert these examples via the add-in. Start with such a simple example if you do not transform and combine data on a regular basis. After you made sure that the code works, adjust it for your purpose, but be careful how the data is transformed.

The same applies if you need to combine data. The `dplyr` also offers a small data set to practice mutating joins (Wickham, François, et al. 2022). The `band_members` data includes `names` from members of two different music bands; and the `band_instruments` data includes their instruments.

```r
# Small data to recapture the join_* functions
band_members
```

```
#> # A tibble: 3 x 2
#>   name  band
#>   <chr> <chr>
#> 1 Mick  Stones
#> 2 John  Beatles
#> 3 Paul  Beatles
```

```r
band_instruments
```

```
#> # A tibble: 3 x 2
#>   name  plays
#>   <chr> <chr>
#> 1 John  guitar
#> 2 Paul  bass
#> 3 Keith guitar
```

Use one of the `join` function (e.g., `inner_join`, `full_join`) to combine the data.

```r
# Mutating joins
band_members |> inner_join(band_instruments, by = "name")
```

```
#> # A tibble: 2 x 3
#>   name  band    plays
#>   <chr> <chr>   <chr>
#> 1 John  Beatles guitar
#> 2 Paul  Beatles bass
```

```r
band_members |> full_join(band_instruments, by = "name")
```

```
#> # A tibble: 4 x 3
#>   name  band    plays
#>   <chr> <chr>   <chr>
#> 1 Mick  Stones  <NA>
#> 2 John  Beatles guitar
#> 3 Paul  Beatles bass
#> 4 Keith <NA>    guitar
```

```
# Further joins:
# band_members |> left_join(band_instruments)
# band_members |> right_join(band_instruments)
```

Finally, one last word about missing values: make sure you explore the data before you run an analysis, but don't neglect to inspect missing and implausible values as well. The `naniar` package has a lot to offer for this task and of course I did not introduce everything it is capable of in Chapter 5. For example, we used the `vis_miss()` function to visualize missing values, but not the amount of missing values. Give the `gg_miss_var()` function a try. It returns a lollipop chart to visualize the amount of missing values. To display percentages, set the `show_pct` option to `TRUE`.

```
# Visualize the amount of missing values
library(naniar)
gg_miss_var(df, show_pct = TRUE)
```
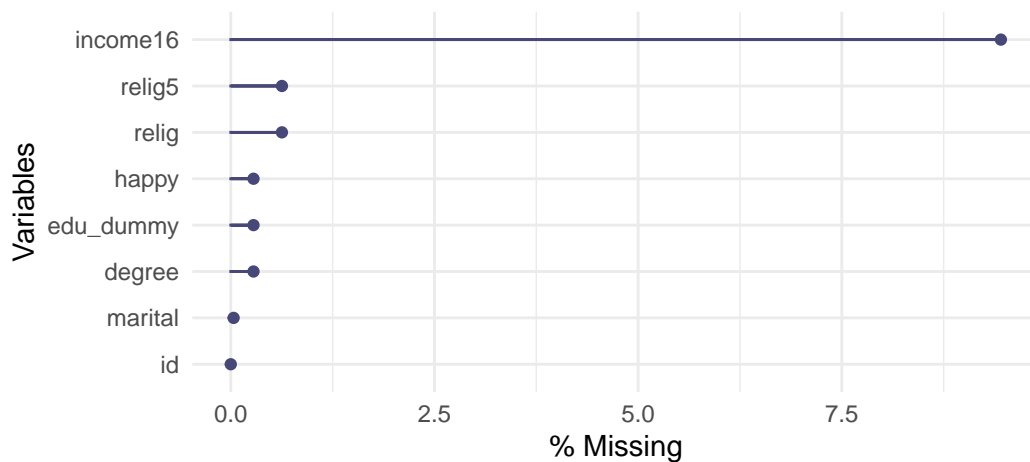

4.6 Summary

In addition to the discussed content, keep the following R functions and packages in mind:

- Import data with different packages. For example:
  - CSV with the `readr` package (Wickham, Hester, and Bryan 2022)
  - Excel with the `readxl` package (Wickham and Bryan 2022)
  - SPSS or Stata with the `haven` package (Wickham, Miller, and Smith 2022)

- Convert objects into numeric (character) vectors (`as.numeric, as.character`)

- Rename columns (`dplyr::rename`)

- Cleans names of an object (`janitor::clean_names`: Firke 2021)

- Combine data:

  - Pivot data from long to wide (`tidyr::pivot_wider`)
  - Pivot data from wide to long (`tidyr::pivot_longer`)
  - Mutating joins (`dplyr::inner_join, left_join, right_join, full_join`)
  - Filtering joins (`dplyr::semi_join, anti_join`)
  - Set pperations (`base::union, intersect, setdiff, setequal`)

- Missing (and implausible) values:

  - The `naniar` package and its function to explore missing values (e.g., `n_miss, n_complete, vis_miss`)
  - Check if something is not available (e.g., `base::is.na`)
  - Convert values to NA (`dplyr::na_if`)
  - Drop rows containing missing values (`tidyr::drop_na`)
  - Replace NAs with specified values (`tidyr::replace_na`)

5 Analyze data

Welcome to the tutorial six 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.

We explored how a linear regression analysis works in Chapter 6. I introduced the corresponding `lm()` function and many packages that help us to develop a linear regression model. This tutorial summarizes the discussed steps and asks you to apply them by running an example analysis. We examine whether life satisfaction, participant's sex, or age has an effect on people's income.

In order to focus on data analysis steps, I have already prepared the `gss2016` data. Keep its limitations in mind. It is a cross-sectional survey and some of the variables such as `income` and `happy` are not a measured on a numerical scale, as the raw data shows. The same applies to `degree`, the educational background is measured as a categorical variable, which is why I transformed it (`degree_num`). We will nonetheless use the `gss2016` data to summarize how a linear regression analysis is implemented in R and which packages help us to develop a model.

```r
# Select variables
library(dplyr)
library(PracticeR)
varlist <- c("income", "age", "sex", "happy", "degree")

# And mutate to create a numerical variable for degree
df <- PracticeR::gss2016 |>
  select(all_of(varlist)) |>
  mutate(degree_num = case_when(
    degree == "Lt High School" ~ 8,
    degree == "High School" ~ 9,
    degree == "Junior College" ~ 12,
    degree == "Bachelor" ~ 15,
    degree == "Graduate" ~ 17,
    degree == NA ~ NA
  ))
```

Furthermore, we implicitly assume that an independent variable `x` influences a dependent variable `y`, although the research design and the data may not allow such a wide-ranging assumptions. For this reason, Chapter 6 introduced the main idea of causality and elaborated which variables we need to control in a linear regression analysis. The *correlation vs. causation* comic strip underlines this point once more.

This tutorial focus on the coding skills to run a linear regression and not on the underlying causal structure between the examined variables. First, we repeat the basics to estimate a linear regression analysis. Next, I ask you to develop your model by examining non-linear effects, interaction effects, and by comparing the performance of such adjustment steps. Finally, I briefly summarize several package that help us with the model specification and assumptions.

```
# Setup of tutorial 6
library(effectsize)
library(estimatr)
library(dotwhisker)
library(huxtable)
library(interactions)
library(jtools)
library(lmtest)
library(performance)
```
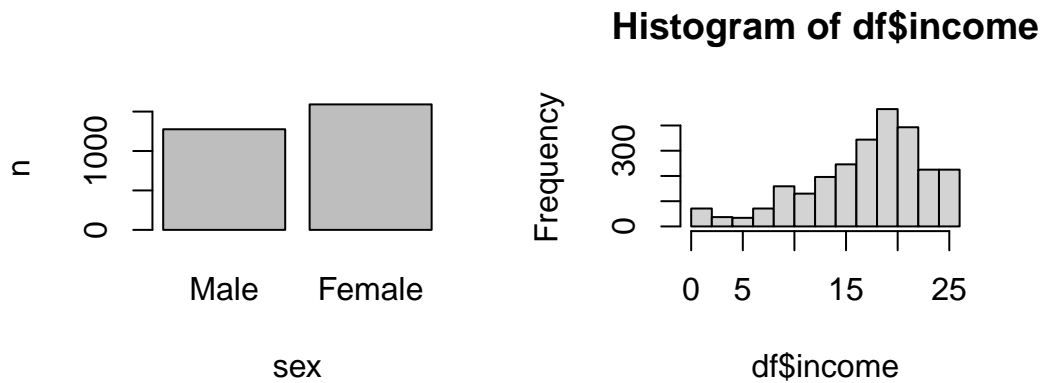
5.1 Estimate a linear regression analysis

I used data for teaching purposes to introduce a linear regression analysis in Practice R. This made it possible to focus on the code and its implementation; we did not explore the data, there was no need to clean the data, prepare variables, or deal with missing values. Such steps are necessary to analyze data and the process is not linear: We start to explore the data, we prepare variables, and run a first analysis. However, often we need to circle back to improve the model due to different reasons (e.g. to include control variables, check on implausible values, etc.). Thus, the first estimation results are preliminary and may substantially change during the course of the model development.

So, we need to explore the variable first. Suppose we examine the gender wage gap: how large is the effect of `sex` on `income`? Explore the distribution of each variable. This gives us an overview how many men and women we observe and whether we may transform the outcome variable in a later step. I already adjusted the graphical parameters (`par`) to put the two graphs next to each other (`mfrow` creates one row and two columns). Create a bar plot and a histogram to examine the variables.

```
# Count sex
count_sex <- dplyr::count(df, sex)

# Plot two graphs
par(mfrow = c(1, 2))
barplot(n ~ sex, data = count_sex)
hist(df$income)
```


We may run a first analysis after we have explored the data, cleaned, and prepared the variables. Use the `lm()` function to estimate a linear regression analysis. The function needs the data and a formula (y ~ x1) to estimate the effect of sex on income.

```
# The lm function
lm(income ~ sex, data = df)
```

```
#>
#> Call:
#> lm(formula = income ~ sex, data = df)
#>
#> Coefficients:
#> (Intercept)     sexFemale
#>     17.7642       -0.7323
```

Since income is not measured on a numeric scale, this coefficient is hard to interpret, but in accordance with theoretical expectations, females have a lower income. The `summary()` function helps us with the interpretation of the model. It returns the estimated coefficients,

R², and further information about the model. In addition, add a second variable with a plus sign (`+`) and examine whether the educational background (`degree_num`) mediates the effect.

```
# The summary function
summary(lm(income ~ sex + degree_num, data = df))
```

```
#>
#> Call:
#> lm(formula = income ~ sex + degree_num, data = df)
#>
#> Residuals:
#>      Min      1Q   Median      3Q      Max
#> -20.8416  -2.7738   0.9904   3.7014  11.2262
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 10.06283    0.39773  25.300  < 2e-16 ***
#> sexFemale   -0.83192    0.21199  -3.924 8.92e-05 ***
#> degree_num   0.69287    0.03285  21.090  < 2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 5.375 on 2590 degrees of freedom
#>   (274 observations deleted due to missingness)
#> Multiple R-squared:   0.15,  Adjusted R-squared:  0.1493
#> F-statistic: 228.5 on 2 and 2590 DF,  p-value: < 2.2e-16
```

Apparently, the wage gap can't be explained by the educational background of the participants since sex has a significant effect. Use the `predict()` function to apply the model. I have already saved the `model` and I created a new data frame with example values. Apply the model and predict how income changes if `degree_num` increases; or examine how predicted values differ between male and female participants. Predicting values give you a better intuition about the model.

```
# The model
model <- lm(income ~ sex + degree_num, data = df)

# Generate example data
new_data <- data.frame(
  sex = c("Female", "Male"),
  degree_num = c(10, 10)
)
```

```
# Apply the model with predict(model, data)
predict(model, new_data)
```

```
#>        1        2
#> 16.15957 16.99149
```

Finally, keep in mind that the **effectsize** package helps us to interpret model parameters such as R^2 (Ben-Shachar et al. 2022). I have saved the summary as **sum_model**. Can you extract R^2 (**r.squared**) from the latter and interpret it with **interpret_r2()** function. As default, it uses the Cohen's **rules** to interpret the effect size.

```
# Assign summary of the model
sum_model <- summary(model)
```

```
# Interpret R2
effectsize::interpret_r2(sum_model$r.squared, rules = "cohen1988")
```

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

5.2 Develop the model

As outlined in Chapter 6, we develop models step by step. We start simple with a bivariate model. We include control variables to inspect how our estimation results change; we examine whether interaction effects mediate the effect; and to which extent an effect is linear. This is not an all-encompassing list, but developing a model step by step implies that we need to compare models to see how the estimation results change between steps. For this purpose we need tables and visualization to compare the estimated models.

We already started to develop a model as we included a second independent variable, but our approach made it hard to comprehend how the estimations results change if we add (drop) a variable. Use the **huxreg()** function from the **huxtable** package to compare models (Hugh-Jones 2022).

```
# Models
m1 <- lm(income ~ sex, data = df)
m2 <- lm(income ~ sex + degree_num, data = df)
```

```
# Develop models step by step
huxtable::huxreg(m1, m2)
```

	(1)	(2)
(Intercept)	17.764 ***	10.063 ***
	(0.169)	(0.398)
sexFemale	-0.732 **	-0.832 ***
	(0.230)	(0.212)
degree_num		0.693 ***
		(0.033)
N	2596	2593
R2	0.004	0.150
logLik	-8256.549	-8038.689
AIC	16519.099	16085.377

*** $p < 0.001$; ** $p < 0.01$; * $p < 0.05$.

In addition, use dot-and-whisker plots to compare model graphically. The `plot_summs()` function from the `jtools` package only needs the model names (Long 2022).

```
# jtools returns a dot-and-whisker plot
jtools::plot_summs(m1, m2)
```
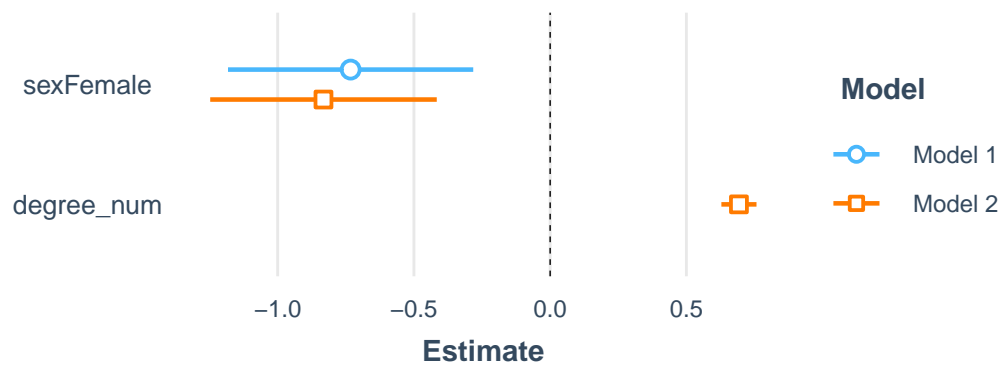

Now that we have established the framework to develop models, let us inspect how we can examine non-linear effects, transform variables, and include interaction effects. Finally we need to check how model changes affect the performance of the model.

We have already applied to `lm()` function when we first created a scatter plot. Since we assume a linear relationship, we should start to examine the effect with a scatter plot in the case of a numerical independent variable. As outlined in Chapter 3, we may insert a regression line with `abline` and the `lm()` function. For example, consider the scatter plot for the effect of age on income.

```
# Create a scatter plot
plot(y = df$income, x = df$age)
abline(lm(income ~ age, data = df), col = "red")
```
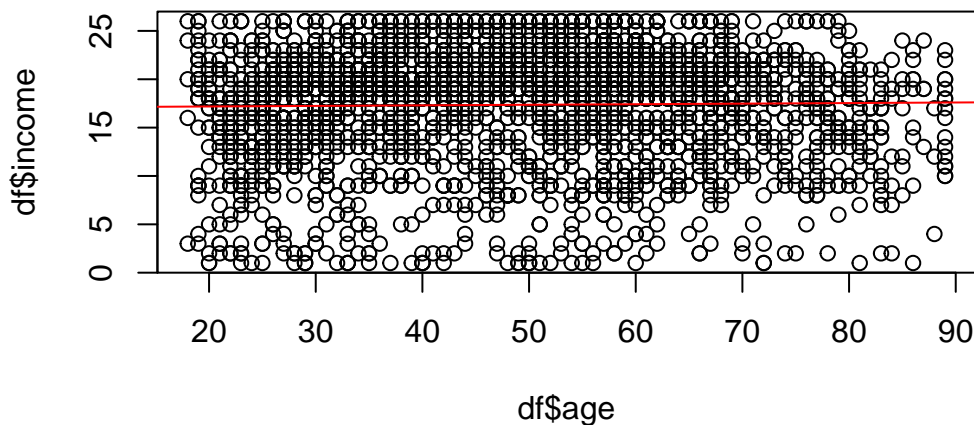

It seems though that both variables are not or only weakly related. Does this mean that we are supposed to stop here since there is no (large) effect? A linear regression assumes a linear effect, but the effect of age on income might not be linear. For example, create a squared age variable and including it in second model to examine if age has a non-linear effect. By including a squared variable for age, we can estimate if the effect increases (decreases) for older people.

```
# Make a squared age variable
df$age_sqr <- df$age^2

# Compare models
m1 <- lm(income ~ age, data = df)
m2 <- lm(income ~ age + age_sqr, data = df)
huxtable::huxreg(m1, m2)
```

65

	(1)	(2)
(Intercept)	17.066 ***	10.421 ***
	(0.343)	(0.880)
age	0.006	0.305 ***
	(0.007)	(0.037)
age_sqr		-0.003 ***
		(0.000)
N	2589	2589
R2	0.000	0.026
logLik	-8238.575	-8205.475
AIC	16483.151	16418.951

*** p < 0.001; ** p < 0.01; * p < 0.05.

We may transform the outcome variable to increase the model fit as well. In the case of income, we often observe many people with little or average income while the amount of people with of a very high income is low. In such a case a logarithm of the income may help to increase the model fit. Keep in mind that the interpretation of the coefficient will change if we transform the variables. Regardless of the interpretation, the `transformer()` function shows what the distribution of a numerical variable would look like (e.g. `log`) if you transform it.

```
# Transform a numerical variable
PracticeR::transformer(df$income)
```

income transformed
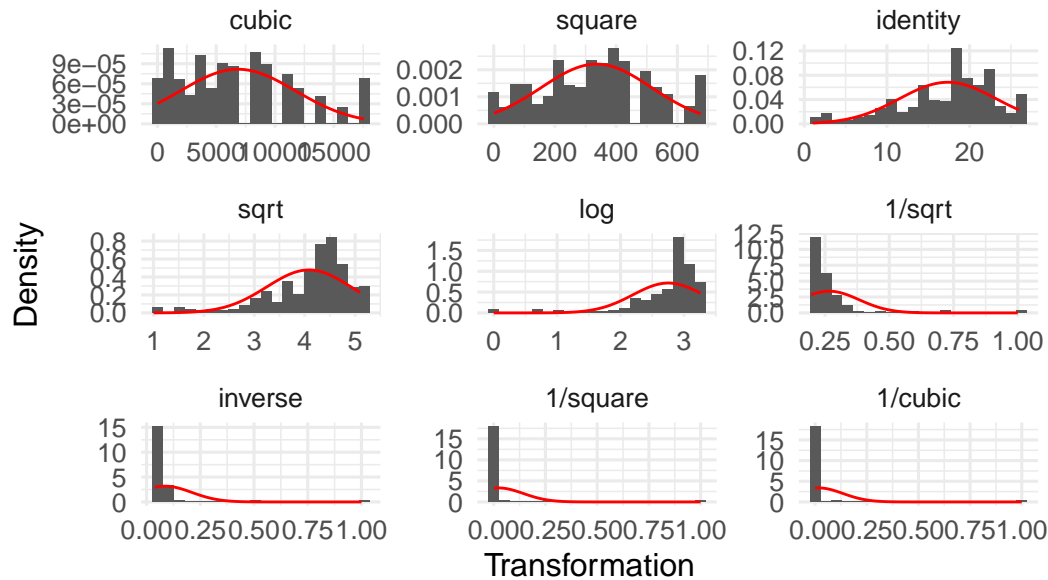
Next, we examine interaction effects: I estimated a model with an interaction effect between `happy` and `sex`. Certainly, I only included it to repeat how this works, but it implies that the effect of happiness on income is moderated by sex. Regardless of my ad-hoc hypothesis, visualize the effect with `cat_plot()` function from the **interactions** package (Long 2021); it needs the model, the name of the predictor (`pred`) and the moderator variable (`modx`). As the plots shows, there is no significant interaction effect.

```
# Interaction of two categorical variables
library(interactions)
m3 <- lm(income ~ happy * sex, data = df)

# cat_plot for categorical predictors
cat_plot(m3, pred = happy, modx = sex)
```
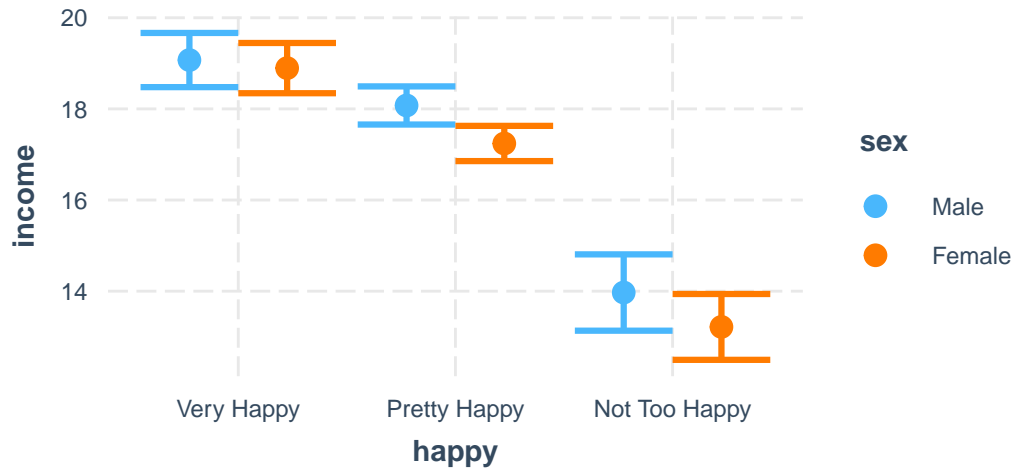
Suppose we believe there is an interaction between sex and education. We may expect that male participants gain much more advantages from education than female participants. Use the `interact_plot()` function with the predictor variable (`pred`) and the moderator (`modx`) variable. The `interval` option shows the confidence interval and we can see the overlap.

```
# Interaction model
m3 <- lm(income ~ sex * degree_num, data = df)

# Interaction between sex*degree_num
interact_plot(m3,
  pred = degree_num, modx = sex,
  interval = TRUE, plot.points = FALSE
)
```
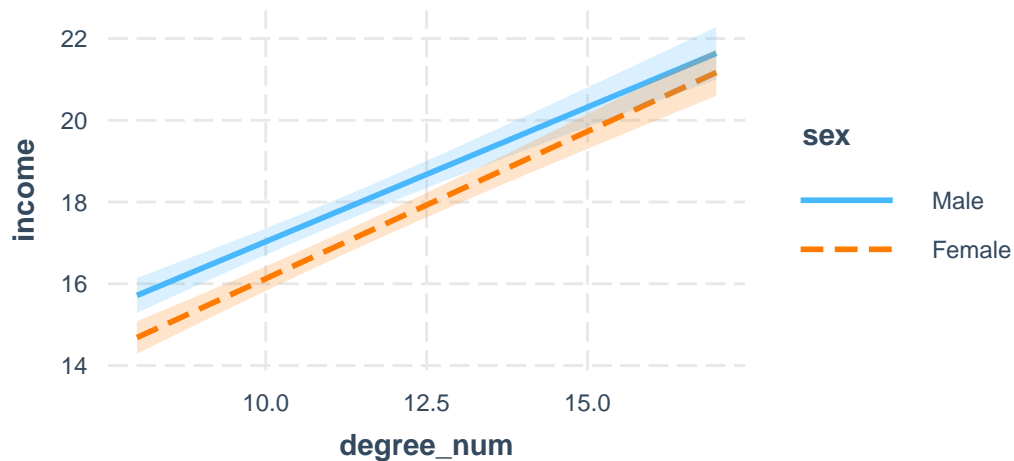
Finally, keep the `performance` package in mind when developing models (Lüdecke, Makowski, Ben-Shachar, et al. 2022). Check how the performance changes if you insert a non-linear parameter, include interaction terms or if you compare different model specifications. The `compare_performance()` function returns several performance indicators and it even creates a radar plot if we assign and plot the results.

```
# Compare performance
library(performance)
performance_models <- compare_performance(m1, m2,
  metrics = c("AIC", "BIC", "R2_adj")
)

# Compare performance
performance_models

#> Comparison of Model Performance Indices

#>Name | Model |   AIC (weights) |   BIC (weights) |  R2 (adj.)
#>----------------------------------------------------------------
#>m1   |    lm | 16483.2 (<.001) | 16500.7 (<.001) | -8.117e-05
#>m2   |    lm | 16419.0 (>.999) | 16442.4 (>.999) |      0.025

# Plot results
plot(performance_models)
```


Comparison of Model Indices

69
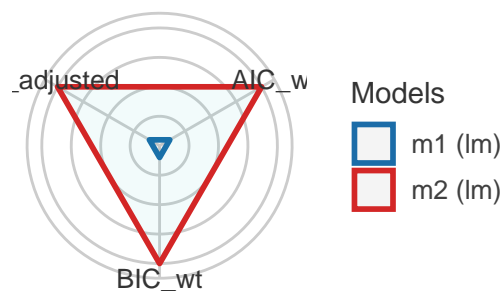
5.3 Improve the model

There are more steps to develop and improve the model. Up to this point we developed the model from a theoretical point of view: we checked if variables interact with each other or in case of a non-linear effect. There is still much room for improvement after we worked off theoretical points. At least we should be aware about the assumptions of a linear regression analysis and the packages that can help us to address such concerns. So, what shall we do if we finalized the first model(s)?

```
# Final model(s)
m_all <- lm(income ~ sex + degree_num, data = df)
```

I introduce the performance package because it gives you a quick overview about potential violations. First, the `check_model()` returns an overview with several plots to check the model assumptions.

```
# Get a quick overview
check_model(m_all)
```

Second, the package has several `check_*` functions to examine assumptions individually. For example, what about multicollinearity and heteroscedasticity?

```
# multicollinearity
check_collinearity(m_all)

#> Check for Multicollinearity

#> Low Correlation

#>       Term  VIF  VIF 95% CI Increased SE Tolerance
#>        sex 1.00 [1.00, Inf]         1.00      1.00
#> degree_num 1.00 [1.00, Inf]         1.00      1.00
#>
#> Tolerance 95% CI
#>    [0.00, 1.00]
#>    [0.00, 1.00]

# check_heteroscedasticity
check_heteroscedasticity(m_all)

#> Warning: Heteroscedasticity (non-constant error variance) detected (p < .001).
```

The last function runs a statistical test to check on the assumptions; in the case of heteroscedasticity we can apply the Breusch & Pagan (1979) test (`bptest`), which runs in the background. The `lmtest` package gives you access to such statistical tests (Hothorn et al. 2022).

```
# Breusch & Pagan test (1979)
lmtest::bptest(m1)
```

```
#>
#>  studentized Breusch-Pagan test
#>
#> data:  m1
#> BP = 3.2195, df = 1, p-value = 0.07277
```

The error of our model is heteroscedastic and the `estimatr` package runs a regression with (cluster) robust standard errors to address this point (Blair et al. 2022). Run a regression with the `lm_robust()` function and adjust the type of standard errors with the `se_type` option.

```
# Robust standard errors
library(estimatr)
robust_model <- lm_robust(income ~ age,
  data = df,
  se_type = "stata"
)

summary(robust_model)
```

```
#>
#> Call:
#> lm_robust(formula = income ~ age, data = df, se_type = "stata")
#>
#> Standard error type:  HC1
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)  CI Lower CI Upper   DF
#> (Intercept) 17.066368   0.348788 48.9305   0.0000 16.382436 17.75030 2587
#> age          0.005891   0.006638  0.8875   0.3749 -0.007125  0.01891 2587
#>
#> Multiple R-squared:  0.0003053 , Adjusted R-squared:  -8.117e-05
#> F-statistic: 0.7876 on 1 and 2587 DF,  p-value: 0.3749
```

Finally, one last word about the visualization of regression results. The `jtools` package provides convenient solutions to create dot-and-whisker plots; and the `dotwhisker` package lets us customize the graph (Solt and Hu 2021). For this purpose I introduce the package, but this does not mean that we have to build a long and complicated code from the ground up each time we need an individual dot-and-whisker plot. In the next chapter we learn more about `ggplot2` which will boost your visualization skills and in a later step we will create functions to create plots efficiently (Wickham, Chang, et al. 2022).

The same applies to the `dotwhisker` package. Once you have built a graph, you can build your own function to create such plots. Don't let complicated code scare you off, we'll soon work on strategies how to create plots without the trouble of memorizing complex code. For example, I created a function called `visualize_model()` which rebuilds the complicated code to create a dot-and-whisker plot from Chapter 6. However, it only needs the models and the names for each predictor to create the plot.

```
# visualize_model() runs dotwhisker in the background
visualize_model(m_all, p1 = "Sex", p2 = "Education")
```
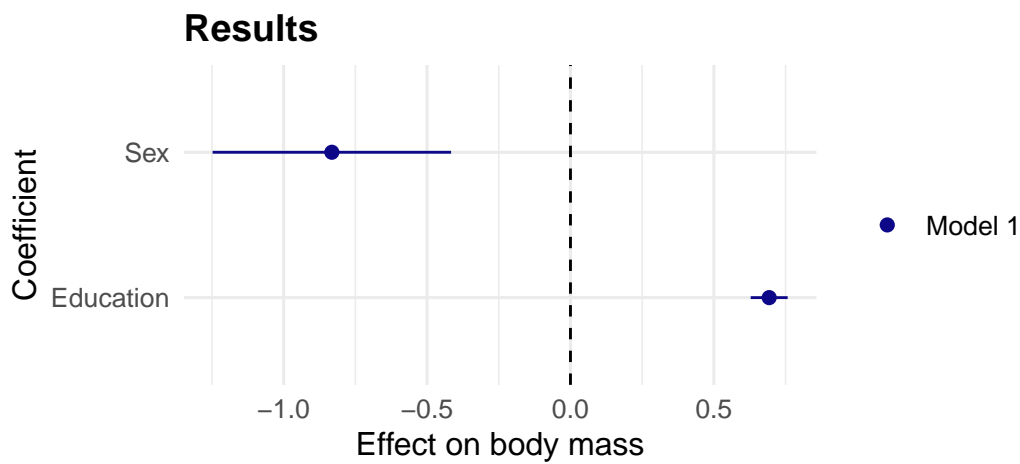

5.4 Summary

Keep the following R functions and packages in mind:

- Fitting linear models (`lm`)
- Model predictions (`predict`)
- Interpret coefficient of determination (`effectsize::interpret_r2`)

- Reorder levels of factor (`relevel`)

- Create a huxtable to display model output (`huxtable::huxreg`)

- Plot regression summaries (`jtools::plot_summs`)

- Plot interaction effects in regression models (e.g., `interactions::interact_plot`)

- The `performance` package and its functions to examine the performance of a model.

  - Compute the model's R2 (`r2`)
  - Compare performance of different models (`compare_performance`)
  - Visual check of model assumptions (e.g.,`check_model, check_outliers, check_heteroscedasticity`)

- Transform a numerical input (`PracticeR::transformer`)

- Export regression summaries to tables (`jtools::export_summs`)

- OLS with robust standard errors (`estimatr::lm_robust`)

- Create fine tuned dot-and-whisker plots API with the `dotwhisker` package

6 Visualize data

Welcome to 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.

Chapter 7 introduced `ggplot2` which gives us plenty of opportunities to visualize data (Wickham, Chang, et al. 2022). We got in touch with the `ggplot()` function, we applied themes, add color, changed fonts, and we learned many more details about the package. Against this background there are at least two options for a `ggplot2` tutorial: I could ask you to apply steps to create a similar plot. It is my personal believe that it needs a lot of time and experience to get fluent in `ggplot2`, which is why we do not pursue such a heroic aim. Some people document the long process to generate a graph as the artwork by Cédric Scherer underlines. It is an animation that shows the steps to create a graph. Thus, it needs a lot of time and effort to develop a customized plot.

Besides the technical skills, the *guiding principles of visualization* will help you to create insightful visualizations. Cairo (2016) summarizes five qualities of a graph as:"

1. *It is truthful*, as it's based thorough and honest research.
2. *It's functional*, as it constitutes an accurate depiction of the data, and it's build in a way that lets do people meaningful operations based on it (seeing change in time).
3. *It's beautiful*, in the sense of being attractive, intriguing, and even aesthetically pleasing for its intended audience – scientists, in the first place, but the general public, too.
4. *It is insightful*, as it reveals evidence that we would have a hard time seeing otherwise.
5. *It is enlightening* because if we grasp and accept the evidence it depicts, it will change our minds for the better." (Cairo 2016: 45).

These principles give us a guidance, but some of them seem complex and depend on the creator and viewer. For example, we probably all agree on the first quality of being truthful, but who says that a graph is (not) beautiful? And what does that even mean? Or a graph might be less insightful if the topic is not novel for the audience. We may argue for a long time whether a principle is fulfilled, but hopefully we agree in the case of obvious flaws that could be improved.

For this reason, this tutorial is dedicated providing first insights about well-known visualization pitfalls and we increase our `ggplot2` skills by learning how to fix them. First, we learn why it is important *to order data*. Then, we inspect why a lot of people perceive *box plots* with

suspicion. Next, we get in touch with a *spaghetti plot*. Finally, we will see what it means *to cut the clutter*.

```
# Tutorial 7 needs the following packages in addition:
library(babynames)
library(forcats)
library(patchwork)
library(dplyr)
library(viridis)
library(ggthemes)
library(ggplot2)
library(showtext)
```

6.1 Order the data

Suppose you created a bar graph to examine cars and their manufacturer (`mpg$manufacturer`). The data is not important, but we need to learn how to order the levels of a categorical variable. As the next console illustrates, the displayed information is difficult to perceive because the bars are all mixed up. Adjust the levels of a factor variable manually or use the `fct_infreq()` function from the **forcats** package to order the data by frequency (Wickham 2022a).

```
#Simple bar graph
p1 <- ggplot(data=mpg, aes(x=manufacturer)) +
  geom_bar()

#Order the data
p2 <- ggplot(data=mpg, aes(x=fct_infreq(manufacturer))) +
  geom_bar()

p1 + p2
```
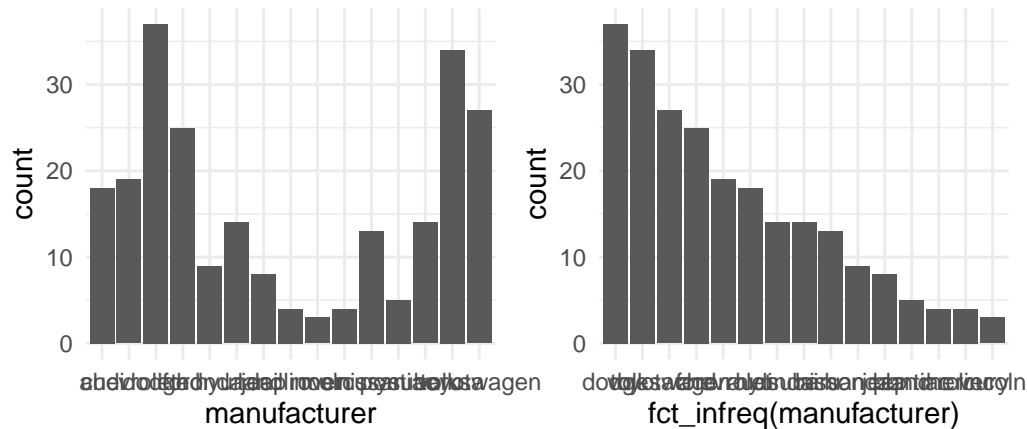
It is our job is to make the graph and its insights accessible. The example underlines that we need to structure and present the data in a way that leverages the message. The last graph also illustrates that there are many group levels making it difficult to depict them all in one graph even if we ordered the data. Moreover, look at the labels, they are not vertically aligned which makes it hard to read. Remember, the `forcats` package offers many functions to manipulate factor variables. For example, display only the five largest groups with the `fct_lump` function and use the `coord_flip()` function to turn around the axes to align the labels vertically.

```
# Lump levels with fct_lump
mpg$manufacturer_max <- fct_lump(mpg$manufacturer, n = 5)

# Left: Plot less levels
p1 <- ggplot(data = mpg, aes(x = fct_infreq(manufacturer_max))) +
  geom_bar()

# Right: Flip axes
p2 <- ggplot(data = mpg, aes(x = fct_infreq(manufacturer_max))) +
  geom_bar() +
  coord_flip()

p1 + p2
```
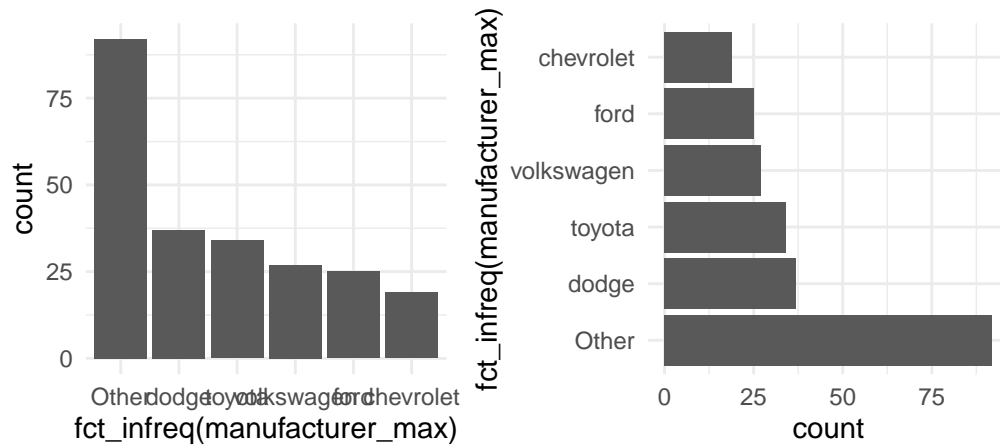
To order the data is important, regardless of the graph created. For example, suppose you examine car consumption (`mpg$hwy`) for different classes of cars (`mpg$class`) with a box plot. Look at the unsorted plot, can you tell me which level has the highest mean? It is complicated to compare groups without a useful order. Try to apply the `fct_reorder()` function, because it lets us reorder the `class` variable by its consumption (`hwy`).

```
# A basic plot
p1 <- ggplot(mpg, aes(hwy, class)) +
  geom_boxplot()

# Use fct_reorder to sort class by their consumption
p2 <- ggplot(mpg, aes(hwy, fct_reorder(class, hwy))) +
  geom_boxplot()

p1 + p2
```
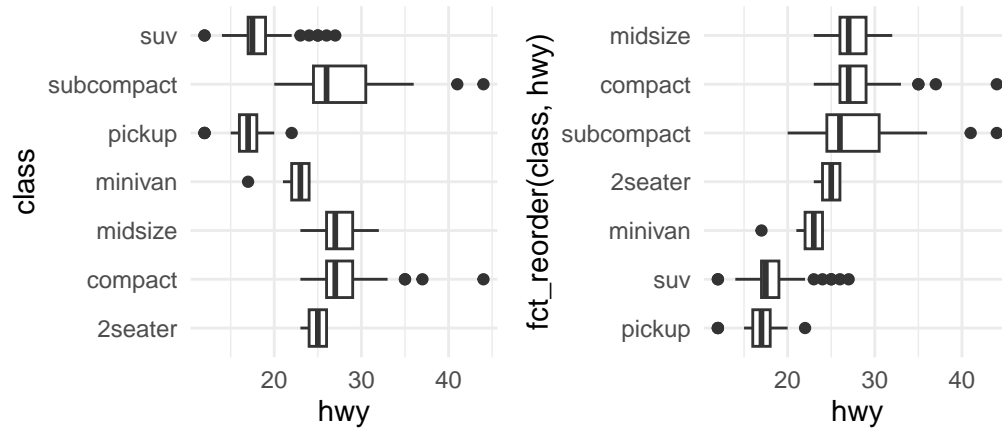
We therefore are supposed to order the data and communicate in a coherent way, otherwise the audience may get confused. There are however additional pitfalls when it comes to box plots.

6.2 Boxplot pitfalls

I generated fake `data` with a `group` and an `outcome` variable to illustrate the main concerns against box plots.

```
# Some fake data
glimpse(data)
```

```
#> Rows: 615
#> Columns: 2
#> $ group   <chr> "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A~
#> $ outcome <dbl> 11.641745, 13.593761, 12.573567, 11.908736, 11.157175, 11.5344~
```

Say you estimated a box plot to examine the differences between the groups. At first glance there seems to be a large differences between groups as the first box plot reveals, but are we comparing on fair grounds? See what happens if you add a `geom_jitter()`. It displays observations with points, but compared to a `geom_point` it adds a small amount of random variation to reduce over plotting.

```
#A Basic geom_boxplot
p1 <- ggplot(data, aes(x = group, y = outcome)) +
  geom_boxplot()

#Add a geom_jitter(color, size, alpha)
p2 <- ggplot(data, aes(x = group, y = outcome)) +
  geom_boxplot() +
  geom_jitter(color = "#d62828",
              size = 0.5,
              alpha = 0.6)

p1 + p2
```
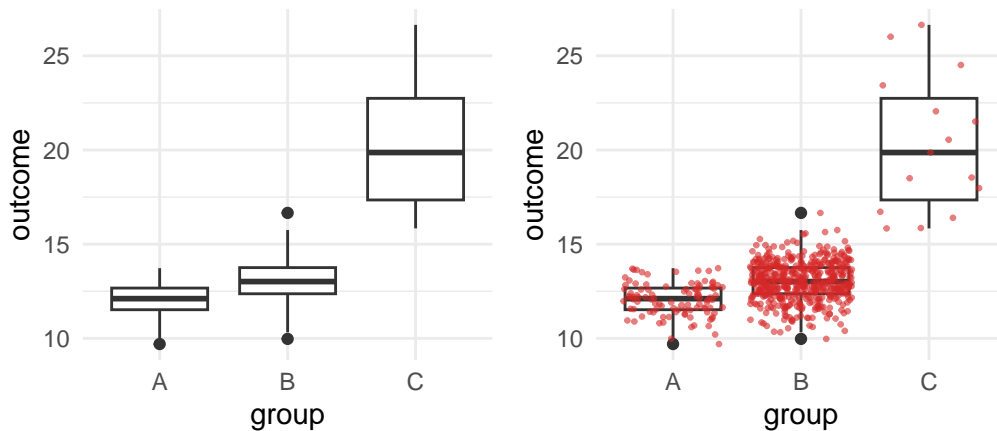

We are comparing three different groups, but the amount of observations are unevenly distributed between the groups and we hardly observe any from group C. This becomes visible when using `geom_jitter()` to add observations, compared to `geom_boxplot()` which does not display the data. A box plot disguises such problems which is obviously a serious concern.

The `geom_jitter` already improved the graph, what else can we do to fulfill the guiding principles of visualization. For example, include the sample size in the graph to make our reader conscious about the problem. The next steps are a bit trickier to apply: Estimate the sample size per group and assign the results. Use the `dplyr::n()` function to count observations, but you will need to group the data first (Wickham, François, et al. 2022).

```
# Estimate sample_size (n) per group
sample_size <- data |>
  dplyr::group_by(group) |>
```

```
  dplyr::summarize(num = dplyr::n())

sample_size


#> # A tibble: 3 x 2
#>   group    num
#>   <chr> <int>
#> 1 A       100
#> 2 B       500
#> 3 C        15
```

To include the sample size in the graph, we need to combine the **group** label and the sample size. We can paste text strings together with the **paste** (and **paste0**) function, as the next console illustrates. It returns text strings which we can include in the graph.

```
# Concatenate Strings with paste (and paste0)
paste(sample_size$group, "has N =", sample_size$num, " observations.")


#> [1] "A has N = 100  observations." "B has N = 500  observations."
#> [3] "C has N = 15  observations."
```

First, combine both data sets with a **left_join()**. Second, create a new variable to add the text label. Use the **paste** function to paste the text label, but also add a new line (**\n**) to separate the group name and the text to display the sample size (**num**).

```
# Join data and mutate with text labels for group_N
data <- data |>
  dplyr::left_join(sample_size) |>
  dplyr::mutate(group_N = paste0(group, "\n", "N=", num))

head(data)


#>   group  outcome num  group_N
#> 1     A 11.64174 100 A\nN=100
#> 2     A 13.59376 100 A\nN=100
#> 3     A 12.57357 100 A\nN=100
#> 4     A 11.90874 100 A\nN=100
#> 5     A 11.15717 100 A\nN=100
#> 6     A 11.53444 100 A\nN=100
```

Now we can use the new variable (`group_N`) as `x` and include the sample size. It goes without saying that there are more ways to improve a box plot (and to include text). For example, we can use a `geom_violin()` to examine the distribution, as the second plot on the right side shows.

```
#Use the new variable group_N as x
p1 <- ggplot(data, aes(x = group_N, y = outcome)) +
  geom_boxplot()+
  geom_jitter(color = "#d62828",
              size = 0.5,
              alpha = 0.6)

#A violin plot and stat_summary
p2 <- ggplot(data, aes(x = group, y = outcome)) +
  geom_violin(width=0.6, alpha=0.8)+
  stat_summary(fun = "median", color = "red",
               size = 1.5, geom = "point")+
  stat_summary(fun.data = return_stats,
               geom = "text",
               size = 2, fontface = "bold",
               hjust = 0.5, vjust = 0.9)

p1 + p2
```
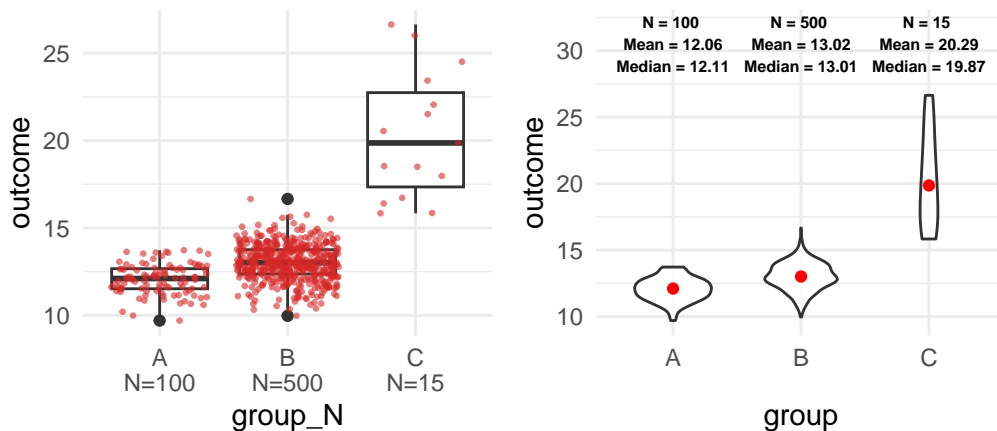

As the right plot shows, I used the `stat_summary()` function twice to include further statistics. First, I used the function to display the median of each group. Second, I used a function (`return_stats`) that returns the statistics and finally the `stat_summary()` function which includes them as text in the plot. The latter approach is more flexible but also more complicated

than the first approach. The next console shows how the function works and we will learn more about the `geom_text` at the end of this tutorial.

```r
# The return_stats function
return_stats <- function(y) {
  return(data.frame(
    value = max(y) * 1.2,
    label = paste(
      "N =", length(y), "\n",
      "Mean =", round(mean(y), 2), "\n",
      "Median =", round(median(y), 2), "\n"
    )
  ))
}


return_stats(data$outcome)
```

```
#>      value                                         label
#> 1 31.97093 N = 615 \n Mean = 13.04 \n Median = 12.88 \n
```

Regardless of the approach, keep in mind that a box plot does not show the data nor does it display the distribution. Compared to that, the `geom_jitter()` displays the data and the violin plot reveals the underlying distribution.

To calculate the sample size or other statistics seems a bit awkward if you are not used to customized plots. Fortunately, there are further **ggplot2** extension package that help us with this task. For example, the **see** package has a `geom_violindot()` function, which combines a violin with a dot plot. The latter makes it convenient to inspect the sample size and the distribution (Lüdecke, Makowski, Patil, et al. 2022). Add the geom, fill the dots black (via `fill_dots`); and find a reasonable size for the dots via `size_dots` option.

```r
# The see package adds a geom_violindot
library(see)
ggplot(data, aes(x = group, y = outcome, fill = group)) +
  geom_violindot(fill_dots = "black", size_dots = 5) +
  scale_fill_material_d(palette = "contrast")
```
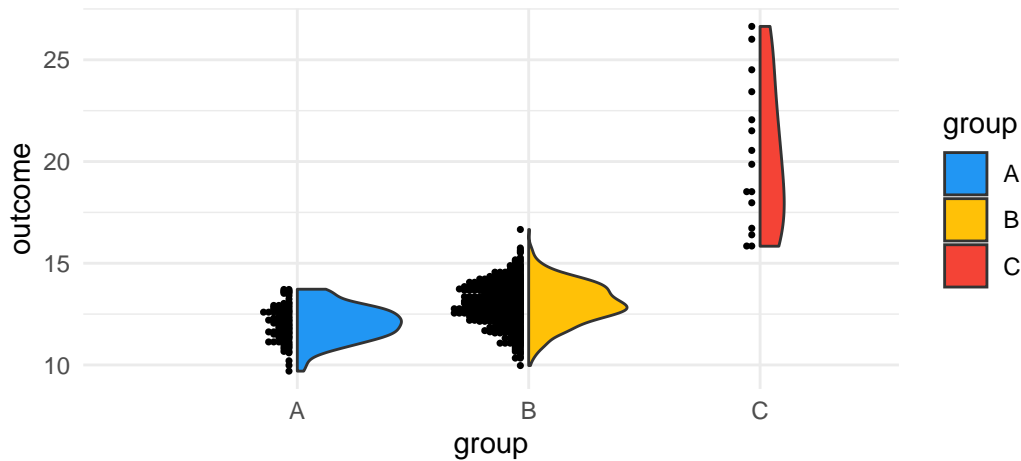
Or consider the `ggstatsplot` package: As the result from the `ggbetweenstats()` function shows, the package automatically adds statistical details to the graph. In our case, it combines box and violin plots to compare the outcome *between* the subjects (Patil 2023).

```r
# The ggstatsplot package
library(ggstatsplot)
ggbetweenstats(data, group, outcome) +
  theme_minimal(base_size = 10)
```
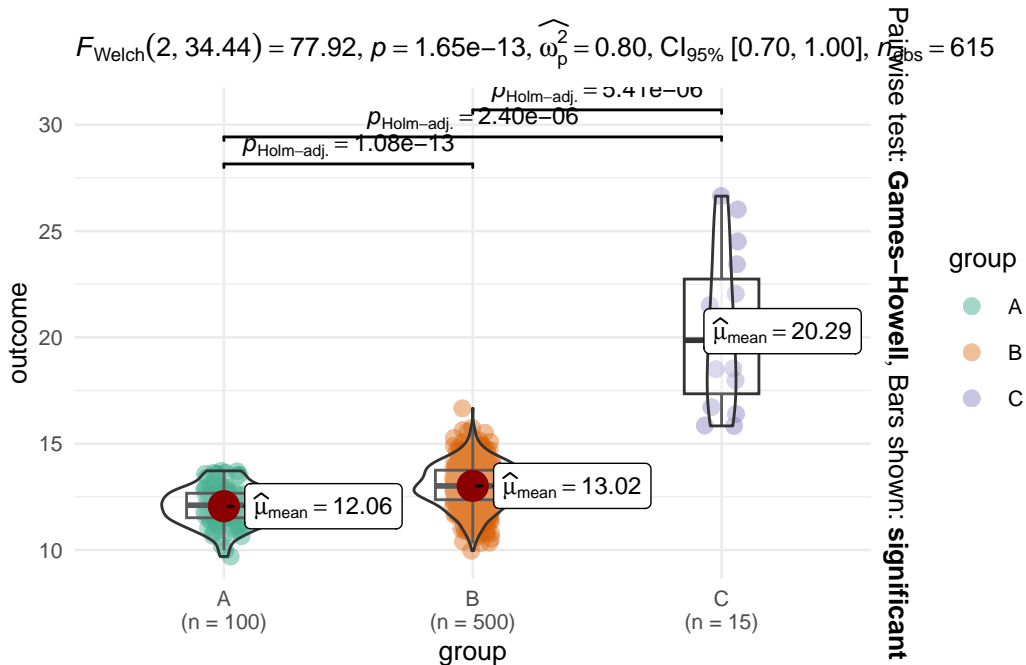
6.3 The spaghetti plot

Another classic visualization pitfall is the spaghetti plot. Essentially it is a line graph with too many lines and colors which is why we cannot see what is going on. We can create a spaghetti plot with the `babynames` package and the corresponding data (Wickham 2021). The package contains names of newborn babies in the US and includes proportion for a long period (1880-2017). Suppose we examine how the most popular male names have been developed over time. I have already prepared the data to identify the most popular male names (Top 10: `name_pop`).

```
# The Top 10 male names
name_pop
```

```
#>  [1] "James"      "Michael"      "Robert"    "John"    "David"
#>  [6] "William"    "Christopher"  "Richard"   "Mark"    "Jason"
```

To visualize how often these names appear, we need to apply a filter to get only male `babynames` and to filter the data for the Top 10 names.

```
# Get male baby names for the Top 10 names
babynames_df <- babynames %>%
  filter(sex == "M" & name %in% name_pop)
```

Next, visualize the data with a line plot (`geom_line`). Use `year` as x, `n` as y, and `name` as group and color aesthetic.

```
# Plot
babynames_df %>%
  ggplot(aes(x = year, y = n, group = name, color = name)) +
  geom_line() +
  scale_color_viridis(discrete = TRUE) +
  ggtitle("A spaghetti chart example")
```

A spaghetti chart example
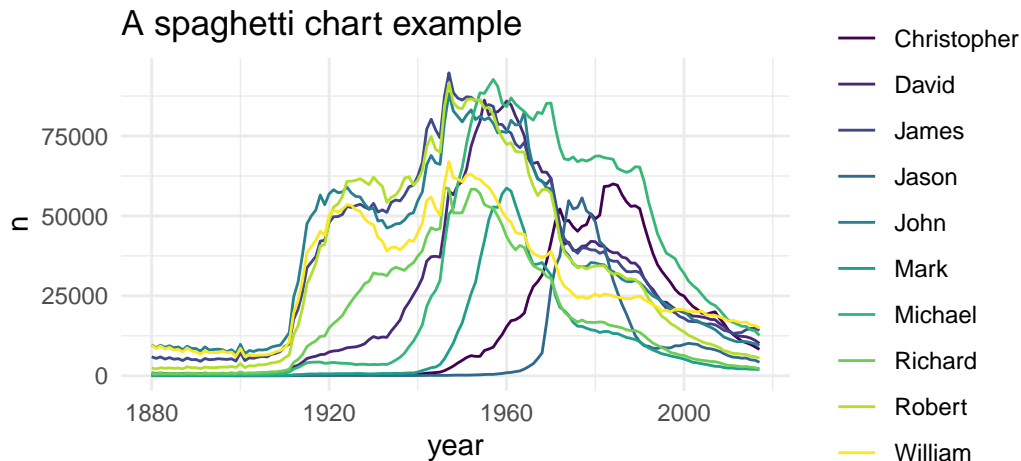
What a confusing graph: single lines look like spaghettis and we can't see how often each name was used over the time. How can we improve the spaghetti plot? You are already familiar with a simple, but powerful solution. Apply a `facet_wrap()` and split the graph in subplots.

```
# Split with facet_wrap
ggplot(babynames_df, aes(x = year, y = n, group = name)) +
  geom_line() +
  facet_wrap(name ~ ., nrow = 2)
```
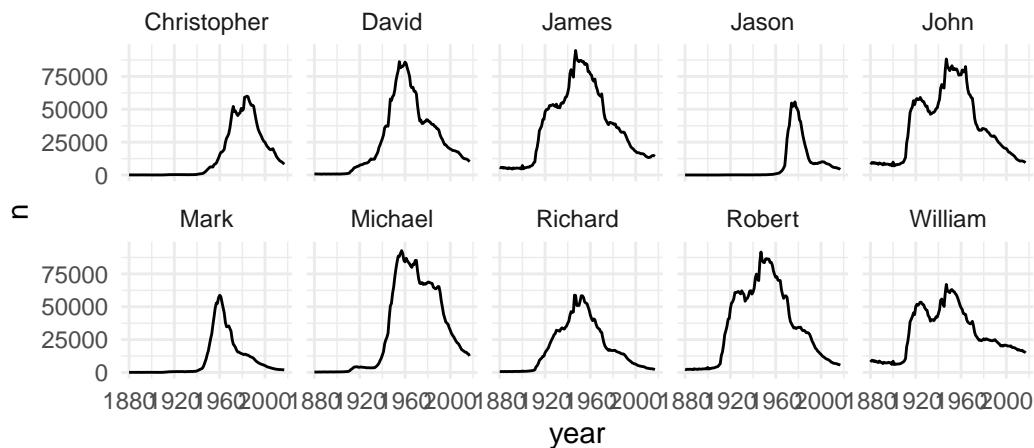

There is still room for further improvement: We could - for example - to draw all lines in gray and highlight for each facet the corresponding line in a different color. First, we need to create a copy of the `name` variable (`facet_names`), which we will use to facet the graph.

```
# Copy the names column
babynames_df$facet_names <- babynames_df$name
```

Next, I prepared the `geom_line()` to create a spaghetti plot one more time with gray lines only, as the first plot on the left side shows. However, see what happens if you add the `facet_wrap()` function and the `facet_names` variable to split the graph.

```
p1 <- ggplot(babynames_df, aes(x=year, y=n)) +
  geom_line(data = babynames_df %>% select(-facet_names),
            aes(group=name),
            color="grey",
            linewidth=0.5,
            alpha=0.5)

p2 <- ggplot(babynames_df, aes(x=year, y=n)) +
  geom_line(data = babynames_df %>% select(-facet_names),
            aes(group=name),
            color="grey",
            linewidth=0.5,
            alpha=0.5)+
  theme_minimal(base_size = 8)+
  facet_wrap(facet_names ~ ., nrow = 2)

p1 + p2
```
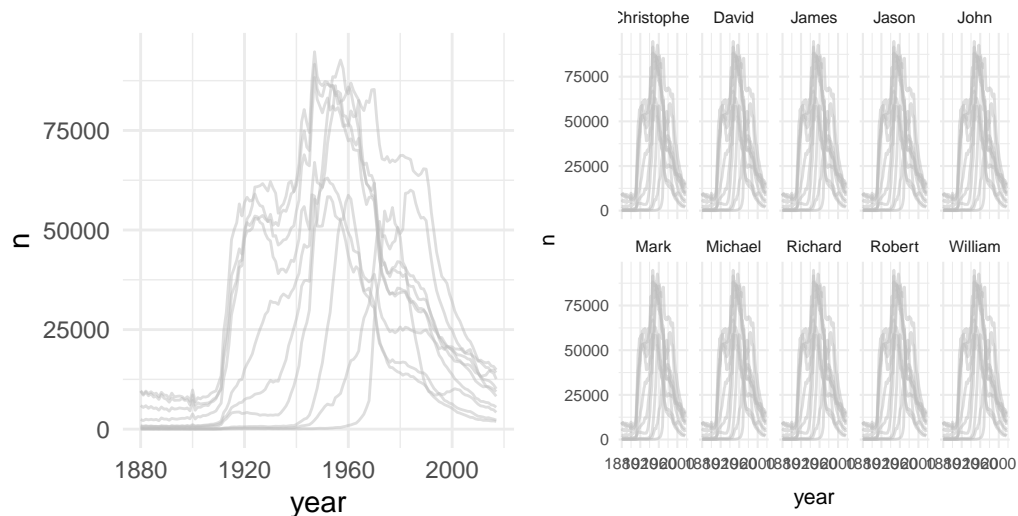
As the second plot show, the new variable gives us the chance to create subplot for each name, but all lines are still included if we use the copy. Next, use a second `geom_line()` for the overlay. Insert the `name` as a color aesthetic, which will make a comparison easier. Moreover, give the overlaying line a distinct `color` and adjust its size with `linewidth`.

```
#add another geom_line as overlay
final_plot <- ggplot(babynames_df, aes(x=year, y=n)) +
  geom_line(data = babynames_df %>% select(-facet_names),
            aes(group=name),
            color="grey",
            linewidth=0.5,
            alpha=0.5) +
  theme_minimal(base_size = 10)+
  facet_wrap(facet_names ~ ., nrow = 2)+
  geom_line(aes(color=name), color="darkred", linewidth=0.75)

final_plot
```
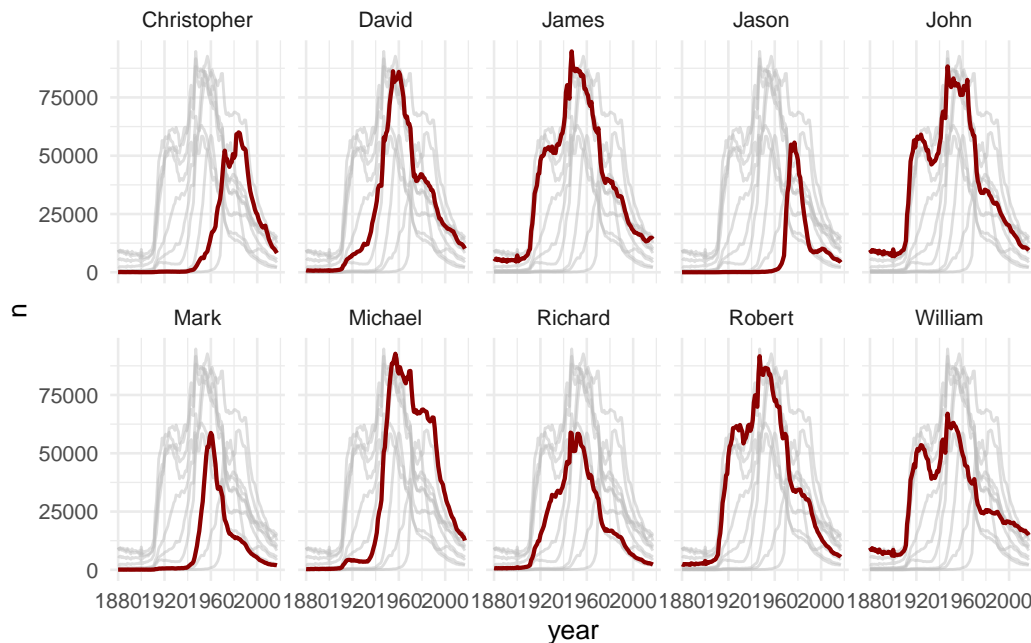

We focused on `ggplot2`, because we need a print version to visualize data in applied empirical research. However, we could also make the last plot interactive to untangle the spaghetti plot. For example, *Highcharts* is a JavaScript software library to create interactive charts and I used the `highcharter` package to create a responsive HTML version of the spaghetti plot

(Kunst 2022). The next console shows the code for an improved version of the graph with the `highcharter` package.

An interactice spaghetti chart example
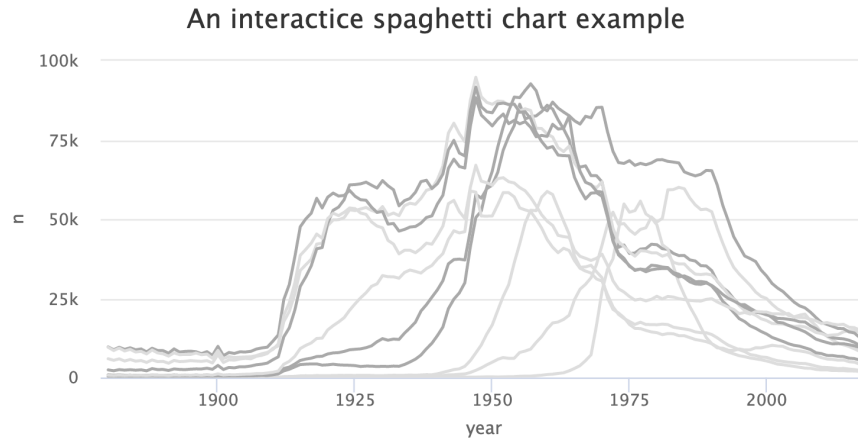
Figure 6.1: The highcharter package

Learning a new package and creating interactive graphs might be too far reaching in the beginning, just keep in mind that such possibilities exits. And in this case it is not even necessary to learn a new package to make the graph interactive, because the `plotly` package can create an interactive version for many standard graphs that are made with `ggplot2` (Sievert et al. 2022). Plotly is a JavaScript library to visualize data and can convert a `ggplot2` object into a plotly chart.

Figure 6.2: The plotly package

Consider reading *Interactive web-based data visualization with R, plotly, and shiny* by Carson Sievert if you want to improve your interactive visualization skills

```
# Interactive web-based data visualization with R, plotly, and shiny
PracticeR::show_link("plotly")
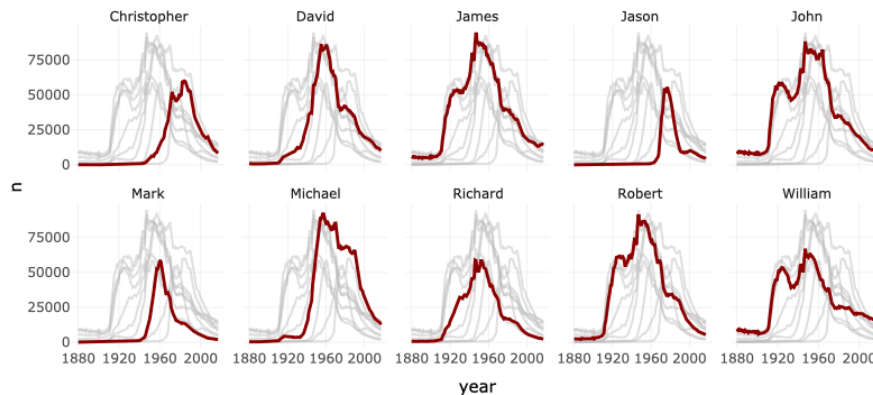```

Instead of learning more about interactive visualization techniques, the last pitfall is not a flaw, it is a principle and an important advice.

## 6.4 Clutter

Edward Tufte underlines: "Clutter and confusion are failures of design, not attributes of information". He highlightes that we are supposed to cut the clutter and get rid of everything that is not necessary to visualize the data.

Consider the next two graphs. I made two bar graphs with a toy data frame and a binary outcome to keep it as simple as possible. I took my quite some time to create a graph that outlines the idea. As the plot on the left side shows, I created a theme that is supposed to look like the old Excel theme with a lot of clutter: The background is gray, I colored the bars even though the color and the legend do not transport any information, and I picked thick, black grid lines for a finishing touch. To compare this ugly beast, the right side shows the `ggplot2` default version. Unfold the code if you want to create a ugly, cluttered graph on your own.
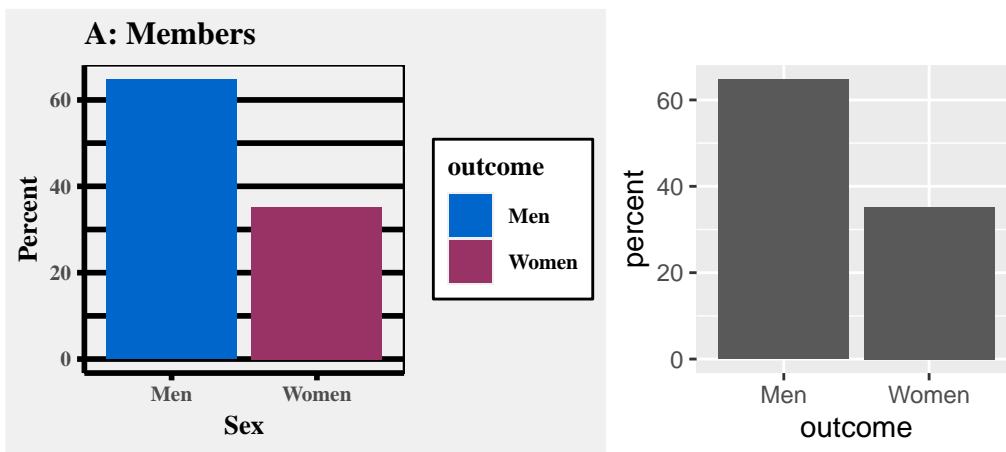


The reinvention of the old Excel theme seems a bit drastic, but even the default `ggplot2` theme has some clutter that we could get rid of. This might not be necessary, but it highlights that there is always room to improve a graph, especially when it comes to clutter. For example, we could use a different `theme` to get rid of the gray background, there is no need to color each bar since they do not represent information, and we could integrate a label for each bar to communicate clearly.

So, fill the bars white and make the border of the bars black. In addition, use a theme without background colors and provide a descriptive title.

```
# De-color de bars
ggplot(df_clutter, aes(x = outcome, y = count)) +
  geom_col(color = "black", fill = "white") +
  theme_minimal(base_size = 12) +
  labs(title = "Members by Sex")
```
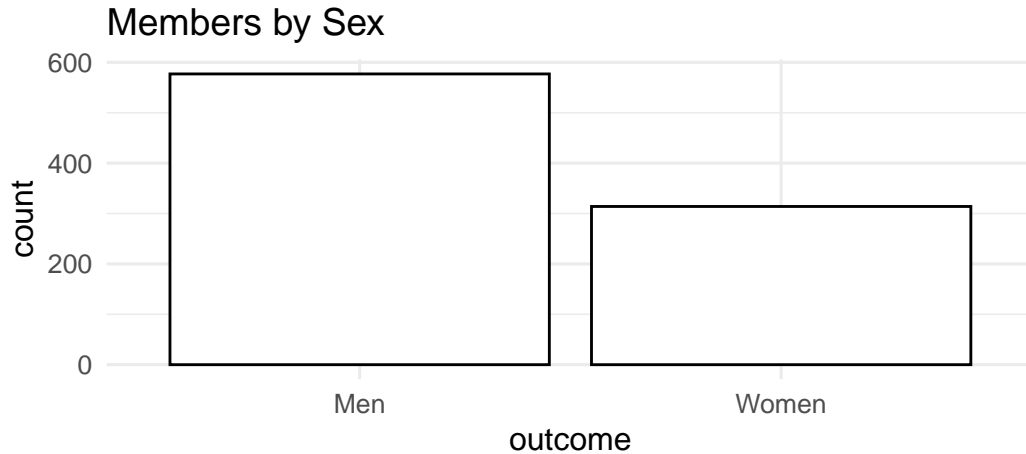

Next, the `geom_text()` helps us to integrate text labels. Essentially, the function displays texts as a geometrical object which is why the main logic is not different compared to other geoms. I added a simple data frame (`df_text`) to illustrate how the geom works. It contains coordinates for `x` and `y` and an example `text` to visualize.

```
df_text <- tibble::tribble(
  ~x, ~y, ~text, ~group,
  -1, -1, "bottom left", "B",
  -1, 1, "top right", "A",
  1, 1, "top left", "A",
  1, -1, "bottom right", "B",
  0, 0, "center", "C"
)
```

As the next console highlights, the function depicts the `text` in accordance with the `x` and `y` coordinate, as the plot on the right side shows. The geom understands supplementary aesthetics and options (e.g., `size`, `fontface`) to display text. To give you an idea how it works, add the color aesthetics for each `group` and adjust the alignment of the text with the

90

vjust (vertical adjustment) and the `hjust` (horizontal adjustment) option. If you set them to `inward`, the text will be aligned towards the center, but there are more alignment options available (e.g., `left`, `right`, `center`) should you prefer those.

```r
# geom_text example
p1 <- ggplot(df_text, aes(x, y)) +
  geom_text(aes(label = text),
    size = 3
  )

# insert color aesthetic and adjust options (e.g., size, fontface)
p2 <- ggplot(df_text, aes(x, y, color = group)) +
  geom_text(aes(label = text),
    vjust = "inward",
    hjust = "inward",
    size = 3,
    fontface = "bold"
  ) +
  scale_color_brewer(palette = "Set1")

p1 + p2
```
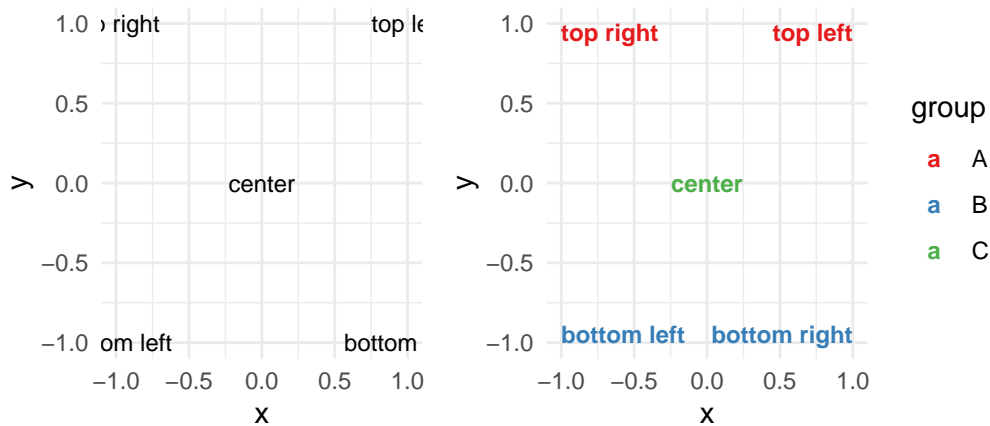

Since the data does not contain text to improve the bar graph, we may use the `paste()` function to create a `label`. It contains the group level, a new line (`\n`), and the percentages.

```r
# Paste a label
paste0(df_clutter$outcome, "\n", df_clutter$percent, "%")
```

```
#> [1] "Men\n64.8%"    "Women\n35.2%"
```

Include the latter as a label and adjust the position via the `y` parameter Use the `count` and adjust it by increasing (decreasing) it manually. In addition, pick a text `color` and a reasonable text `size`.

```
#Include text labels inside the bars
ggplot(df_clutter, aes(x=outcome, y=count)) +
  geom_col(color = "black", fill = "white")+
  geom_text(aes(label = paste0(outcome, "\n", percent, "%"),
                y = count - 100),
            color="black",
            size = 3.5)+
  theme_minimal(base_size = 12)+
  labs(title= "Members by Sex")
```

Members by Sex
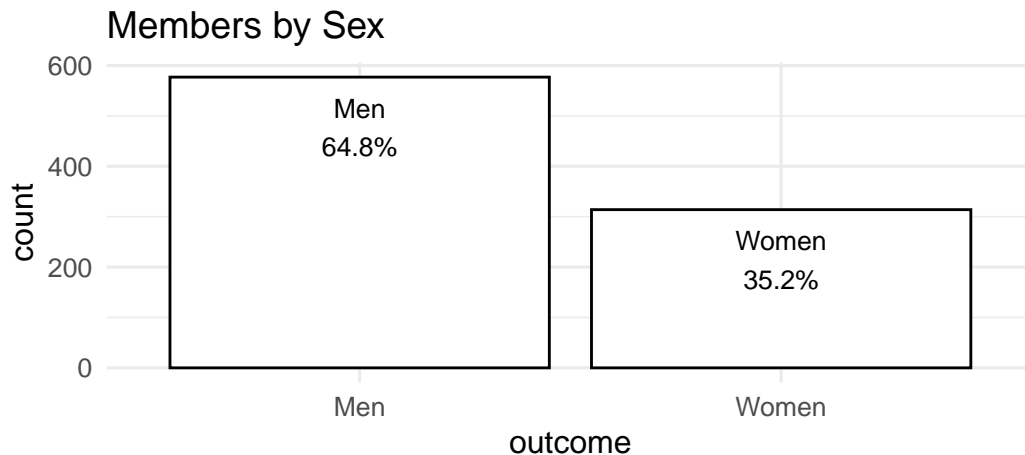
6.5 Summary

I highlighted several books to improve your `ggplot2` and data visualizations skills, but at the end of the day your skills will improve faster, if you start to visualize data on your own and accept that trial and error are not necessarily a wrong approach. To this end, the `ggplot2` cheat sheet (from the package website) will support you as well.

In addition, keep the following functions and packages from Chapter 7 in mind:

- Create a new ggplot (`ggplot`), aesthetic mappings (`aes`), and add a geom_* (e.g., `geom_bar, geom_point, geom_smooth`)

- Add a layer with `+`, start each new function on a new line, don't forget to delete the plus sign if you delete the last line of code

- There are several predefined theme functions (e.g., `theme_bw`, `theme_light`).

- Modify axis, legend, and plot labels (e.g., with `labs`)

- Lay out panels in a grid (e.g., `facet_grid`)

- Discard (or adjust) the legend (e.g., `theme(legend.position = "none")`)

- Adjust the coordinate system (e.g., `coord_cartesian`)

- Further packages:

  - Themes: `ggthemes` (Arnold 2021)
  - Font types: `showtext` (Qiu 2022)
  - Color: The `RColorBrewer` (Neuwirth 2022) and the `viridis` package (Garnier 2021)
  - Many color palettes: `paletteer` (Hvitfeldt 2021)
  - Combine graphs: `patchwork` (Pedersen 2022b)
  - Zoom in: `ggforce` (Pedersen 2022a)

7 Create tables

Welcome to the 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 Chapter 8, we learned how to create documents with `rmarkdown`, but in this tutorial we will focus on tables (Allaire et al. 2022). I tried to convince you that R is an excellent companion to create documents, but we only discovered the tip of the iceberg when it comes to tables. We focused on the `flextable` (Gohel and Skintzos 2022), `huxtable` (Hugh-Jones 2022), and the `stargazer` package (Hlavac 2022) because they make it comfortable to create tables to report the results of an analysis.

There are many more packages to create tables, each specialized for their specific output format (e.g., HTML, PDF) and they each rely on a different approach to create tables. It is up to you to decide which one suits you best. You can stick to the introduced packages if you are happy with the tables we made in Chapter 8. However, this tutorial gives a glimpse of other approaches and potential next steps.

For example, consider the `DT` packages to create interactive HTML tables (Xie, Cheng, and Tan 2022). The next console shows an illustration with output from the `penguins` data which we will use in this tutorial. The `datatable` function returns the HTML output, the user can even sort or filter the data.

```r
library(palmerpenguins)
library(DT)
datatable(data = penguins, class = "cell-border stripe", filter = "top")
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
	All	All	All	All	All	All	All	All
1	Adelie	Torgersen	39.1	18.7	181	3750	male	2007
2	Adelie	Torgersen	39.5	17.4	186	3800	female	2007
3	Adelie	Torgersen	40.3	18	195	3250	female	2007
4	Adelie	Torgersen						2007
5	Adelie	Torgersen	36.7	19.3	193	3450	female	2007
6	Adelie	Torgersen	39.3	20.6	190	3650	male	2007
7	Adelie	Torgersen	38.9	17.8	181	3625	female	2007
8	Adelie	Torgersen	39.2	19.6	195	4675	male	2007

Showing 1 to 10 of 344 entries   Previous 1 2 3 4 5 … 35 Next

Instead of creating HTML tables, we will explore different packages to create tables for static documents such as PDF files. First, I introduce the `gt` package because it makes elegant tables and we use its framework to underline why different packages rely on different frameworks. Next, I highlight the `kableExtra` package (Zhu 2021) because it provides many cool features for PDF (and HTML) documents. Finally, we pick up where we left and practice. We repeat the main functions of the `huxtable` package, but this time we reduce our work effort by developing our own table functions.

```
# Makes sure the following packages have been installed:
library(DT)
library(dplyr)
library(gt)
library(huxtable)
library(kableExtra)
library(modelsummary)
library(tidyr)
```

7.1 The gt package

The R community has developed many cool packages to create tables. They rely on different approaches, have a different aim, or are specialized for different output formats. For example, the `gt` package creates elegant tables for PDF and HTML files and it outlines its approach to create tables graphically (Iannone et al. 2022). The next Figure shows the parts to create a `gt` table.

Approaches to create customized tables can quickly become complex, since even a simple table includes many parts (e.g., header, labels, body, etc.) that need to be defined, formatted, and
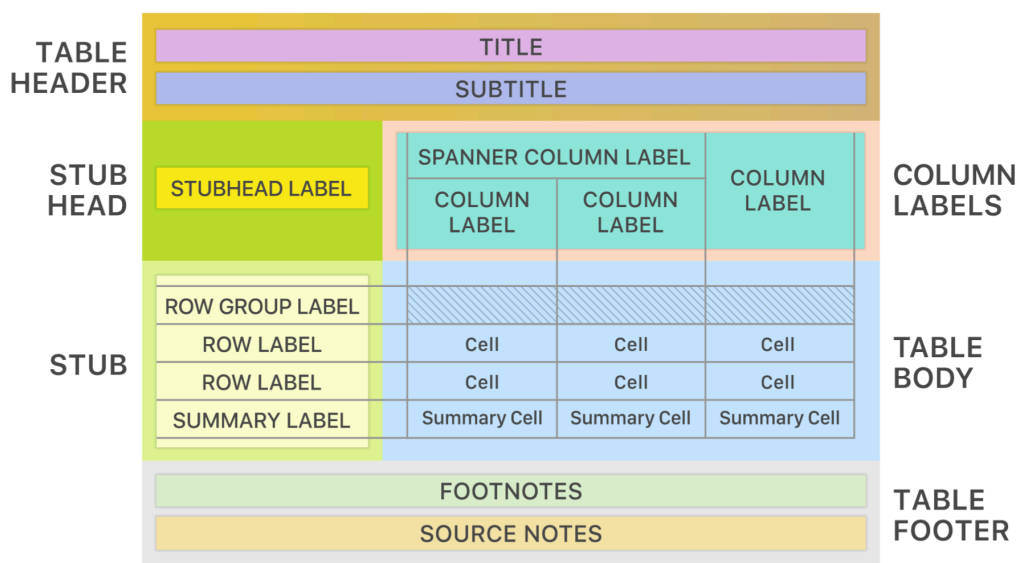
The Parts of a gt Table

TABLE HEADER	TITLE			
	SUBTITLE			
STUB HEAD	STUBHEAD LABEL	SPANNER COLUMN LABEL		COLUMN LABEL
		COLUMN LABEL	COLUMN LABEL	
STUB	ROW GROUP LABEL			
	ROW LABEL	Cell	Cell	Cell
	ROW LABEL	Cell	Cell	Cell
	SUMMARY LABEL	Summary Cell	Summary Cell	Summary Cell
	FOOTNOTES			
	SOURCE NOTES			

COLUMN LABELS · TABLE BODY · TABLE FOOTER

Figure 7.1: The gt package: Artwork by Iannone et al. (2022)

generated for a certain output. Irrespective of the package, the first step to create a table are often similar and not complicated. We thus need to prepare the output and give it to the package. As the next console shows, I estimated the mean of several variables for each `species` of the `penguins` data which we will use as an example input for the table. The corresponding `gt()` function returns the input as a simple, but elegant table.

```r
# Create table output
penguins_table <- penguins |>
  group_by(species) |>
  drop_na() |>
  summarise(across(bill_length_mm:flipper_length_mm, mean))

# Create a gt table
library(gt)
gt_tbl <- gt(penguins_table)
gt_tbl
```

The package has functions and options to improve the default result. For example, `fmt_number()` rounds numerical columns; we can format the table header (`tab_header`) and the column labels (`cols_label`) with the `md()` function which interprets the input as Markdown. By the way, the `html()` does essentially the same for HTML code. Never mind if

species	bill_length_mm	bill_depth_mm	flipper_length_mm
Adelie	38.82397	18.34726	190.1027
Chinstrap	48.83382	18.42059	195.8235
Gentoo	47.56807	14.99664	217.2353

The Palmerpenguins

Species	Bill Length (mm)	Bill Depth (mm)	Flipper Length (mm)
Adelie	38.82	18.35	190.10
Chinstrap	48.83	18.42	195.82
Gentoo	47.57	15.00	217.24

are not yet familiar with HTML, Chapter 11 gives you a hands on and the next console shows the discussed code and table.

```
# Improve the table
gt_tbl |>
  fmt_number(
    columns = c(bill_length_mm, bill_depth_mm, flipper_length_mm),
    decimals = 2
  ) %>%
  tab_header(
    title = md("**The Palmerpenguins**")
  ) |>
  cols_label(
    species = "Species",
    bill_length_mm = md("Bill Length (mm)"),
    bill_depth_mm = md("Bill Depth (mm)"),
    flipper_length_mm = md("Flipper Length (mm)")
  )
```

Like this, there are many cool packages to create tables, but depending on our aim, it may become quite complicated. Let me underline this point with the `kableExtra` package.

7.2 The kableExtra package

You can create awesome HTML and LaTeX tables with `knitr::kable()` and the `kableExtra` package. As we have seen before, the first step is not complicated, we need an input and the `kbl()` function returns a basic table.

```
# https://haozhu233.github.io/kableExtra/awesome_table_in_html.html
# booktabs = TRUE
penguins_table %>%
  kbl()
```

species	bill_length_mm	bill_depth_mm	flipper_length_mm
Adelie	38.82397	18.34726	190.1027
Chinstrap	48.83382	18.42059	195.8235
Gentoo	47.56807	14.99664	217.2353

The package provides many features to adjust the table. For example, there are predefined rules (based on CSS, see Chapter 11) to style the appearance of a HTML table. See what happens if you add `kable_styling()`. It returns the table in minimal style.

```
# Styles for HTML tables
penguins_table %>%
  kbl(booktabs = T) %>%
  kable_styling()
```

In a similar vein, there many options to style an HTML table. The `bootstrap_options` returns `striped` cells in white and light gray, we can adjust the width of the table (e.g., `full_width`); and define the `position` of the table.

```
# Further options of an HTML table
kbl(penguins_table) %>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

Unfortunately, all flexible approaches to create tables become complex if we want to customize a table in more detail. The next console shows a complicated illustration. I adjusted the header, I used colors to highlight values, and I inserted an inline histogram of depict `flipper_length_mm`. Please ignore the code details and inspect the package vignette for more information. I will not outline how it works, but it underlines how complex the code can become.

```
# Customize the table in LATEX
peng_split <- split(penguins$flipper_length_mm, penguins$species)
penguins_table$flipper_length_mm2 <- ""
penguins_table$species <- ""

penguins_table %>%
  kbl(booktabs = TRUE, col.names = c(
```

Species	Bill length	Bill Depth	Flipper Length	Histogram
	38.82397	18.34726	190.1027	
	48.83382	18.42059	195.8235	
	47.56807	14.99664	217.2353	

```r
    "Species",
    "Bill length",
    "Bill Depth",
    "Flipper Length",
    "Histogram"
)) |>
kable_paper(full_width = FALSE) |>
kable_styling(font_size = 10) |>
row_spec(0, bold = T, font_size = 12) |>
column_spec(1,
  image = spec_image(c(
    "images/ch_08/p1.png",
    "images/ch_08/p2.png",
    "images/ch_08/p3.png"
  ), 200, 100)
) |>
column_spec(2,
  bold = T,
  color = spec_color(penguins_table$bill_length_mm[1:3])
) |>
column_spec(5,
  bold = T,
  image = spec_hist(peng_split, width = 200, height = 100)
)
```

Is it worth to create such customized tables? It depends on your goal and the possibilities to recycle the code. Overall, all those different packages rely on different approaches to generate tables. There is not one packages for all purposes and it depends on your taste and needs or which approach you prefer. For this reason this tutorial tried to raise awareness that several excellent packages to create tables exist. I did not even discuss them all, but I have another one for the road.

Consider the `modelsummary` package because it provides many features to create tables for

	Unique	Missing Pct.	Mean	SD	Min	Median	Max
bill_length_mm	165	1	43.9	5.5	32.1	44.5	59.6
bill_depth_mm	81	1	17.2	2.0	13.1	17.3	21.5
flipper_length_mm	56	1	200.9	14.1	172.0	197.0	231.0
body_mass_g	95	1	4201.8	802.0	2700.0	4050.0	6300.0
year	3	0	2008.0	0.8	2007.0	2008.0	2009.0

models (Arel-Bundock 2023). Furthermore, the `datasummary_skim()` creates a nice summary table. It even let you determine the `output` style and change its overall appearance. Pick a output style, for example, `flextable`, `gt`, or `kableExtra`.

```
# output style: gt, kableExtra, flextable, huxtable
library(modelsummary)
datasummary_skim(penguins, output = "gt", histogram = FALSE)
```

To report research findings, customized tables are definitely worth the trouble, but we can reduce our effort to create tables. Let us switch back to the `huxtable` package and improve our skills to create tables for research findings.

7.3 The huxtable package

Let us revise what we learned in Chapter 8. First, we created a table with the `huxreg` function and in a second step I introduced some options to improve the table. I already estimated three example linear regression models (`m1`, etc.). Can you create a regression table with the `huxtable` package?

```
# The models
m1 <- lm(body_mass_g ~ bill_length_mm,
  data = penguins
)
m2 <- lm(body_mass_g ~ bill_length_mm + flipper_length_mm,
  data = penguins
)
m3 <- lm(body_mass_g ~ bill_length_mm + flipper_length_mm + sex,
  data = penguins
)

# The minimal code
library(huxtable)
huxreg(m1, m2, m3)
```

	(1)	(2)	(3)
(Intercept)	362.307	-5736.897 ***	-5433.534 ***
	(283.345)	(307.959)	(286.558)
bill_length_mm	87.415 ***	6.047	-5.201
	(6.402)	(5.180)	(4.860)
flipper_length_mm		48.145 ***	48.209 ***
		(2.011)	(1.841)
sexmale			358.631 ***
			(41.572)
N	342	342	333
R2	0.354	0.760	0.807
logLik	-2696.987	-2527.741	-2426.664
AIC	5399.975	5063.482	4863.327

*** $p < 0.001$; ** $p < 0.01$; * $p < 0.05$.

In the second step, I outlined that we can omit coefficients(`omit_coefs`), adjust the reported `statistics`, and add a `note` to inform the reader about the model. These options are not a comprehensive list, but they illustrated some of the typical steps to create a table for a publication. Thus, omit the model's intercept (`(Intercept)`), pick some `statistics` (e.g., `nobs` for N; `r.squared`), and add a `note`. In addition, format the returned numbers of the table with `number_format`.

```
# Show my models via huxreg()
huxreg(m1, m2, m3,
  omit_coefs = "(Intercept)",
  statistics = c(`N` = "nobs", `R²` = "r.squared"),
  number_format = 2,
  note = "Note: Some important notes."
)
```

We can recycle a lot of code the next time we need to report a similar table, but there are still a lot of steps involved to create such a table. And who can remember all those options? So, we can define what the table should look like, without the need to rebuild a table from scratch

	(1)	(2)	(3)
bill_length_mm	87.42 ***	6.05	-5.20
	(6.40)	(5.18)	(4.86)
flipper_length_mm		48.14 ***	48.21 ***
		(2.01)	(1.84)
sexmale			358.63 ***
			(41.57)
N	342	342	333
R^2	0.35	0.76	0.81

Note: Some important notes.

every time: We improve our coding skills by learning how to create our own table functions. We already defined the most important options to create the table. The next time we create a similar table, we need to update the estimated models, text labels, or the note.

Create a new function: Give the function a name (e.g., `my_huxreg`) and insert the code from the last step into the body of the function. As a first step, the function should only update the included models. Instead of the function parameters, put three points (...) inside the `function()` and the `huxreg()` function instead of the model names. Such a dot-dot-dot argument allows us to send uncounted numbers of arguments (here models) to the `huxreg()` function. Moreover, create a list with model names (`modelfits`) and test the approach by running the `my_huxreg()` function with the estimated models.

```r
# Create your own huxreg function
my_huxreg <- function(...) {
  huxreg(...,
    omit_coefs = "(Intercept)",
    statistics = c(`N` = "nobs", `R²` = "r.squared"),
    number_format = 2,
    note = "Important note"
  )
}

# Create a list of models
modelfits <- list(
  "Model A" = m1,
```

```
  "Model B" = m2,
  "Model C" = m3
)

my_huxreg(modelfits)
```

	Model A	Model B	Model C
bill_length_mm	87.42 ***	6.05	-5.20
	(6.40)	(5.18)	(4.86)
flipper_length_mm		48.14 ***	48.21 ***
		(2.01)	(1.84)
sexmale			358.63 ***
			(41.57)
N	342	342	333
R²	0.35	0.76	0.81

Important note

We only passed the models via the function, but we can integrate further function parameters to improve the approach. For example, each time we create a new table, text labels for the variables names (`coefs_names`) are needed; we may omit different variables (`drop`) from the models; and - depending on the outcome and the reported models - we should adjust the `message` of the note. Instead of providing these options inside the function, include them as parameters and insert their objects names in the `my_huxreg` function.

```
# Include option parameters
my_huxreg <- function(..., coefs_names, drop, message) {
  huxreg(...,
    coefs = coefs_names,
    omit_coefs = drop,
    statistics = c(`N` = "nobs", `R²` = "r.squared"),
    number_format = 2,
    note = message
  ) |>
    set_bold(row = 1, col = everywhere) |>
    set_align(1, everywhere, "center")
```

```
}
```

Moreover, the `huxtable` is not made for regression tables only, but for tables in general. For this reason the package has much more to offer than the discussed options. Consider the last two lines of code of the solution: I added them to illustrate this point. The `set_bold` function prints the first row of all columns in bold; and I align numbers (`set_align`) in the `center`.

Regardless of the discussed steps, we can now recycle the code by creating a function. We only need to hand over the estimated models and the new information about the models. I already started to create text labels for the coefficients (`coefs`). Adjust which variables you will drop (`droped_coefs`); the `message` option, and insert those objects into the `my_huxreg` function.

```
# Option input
coefs <- c(
  "Bill length" = "bill_length_mm",
  "Flipper length" = "flipper_length_mm",
  "Male" = "sexmale"
)


droped_coefs <- c("(Intercept)")
message <- "Note: Some important notes that can change."

# Create table (my_huxreg is based on the solution of the last console)
mytable <- my_huxreg(modelfits,
  coefs_names = coefs,
  drop = droped_coefs,
  message = message
)
mytable
```

We will learn more about automation and text reporting in Chapter 10, but keep in mind that such steps to create your own function are often worth considering, as it makes the code less clunky and reproducible. If you create a document, save your function in a separate R script, and include it in your document via the `souce()` function in the setup R chunk. It runs the code if you render the document.

Finally, let me briefly introduce Quarto (`.qmd`), a publishing system to create many different document types. In a similar sense as `rmarkdown`, it uses `knitr` and Pandoc to create documents with R. As the illustration from Allison Horst underlines, Quarto is a flexible system since it is not tied to R. As outlined on the homepage: "Quarto was developed to be multilingual, beginning with R, Python, Javascript, and Julia, with the idea that it will work even for languages that don't yet exist".

	Model A	Model B	Model C
Bill length	87.42 ***	6.05	-5.20
	(6.40)	(5.18)	(4.86)
Flipper length		48.14 ***	48.21 ***
		(2.01)	(1.84)
Male			358.63 ***
			(41.57)
N	342	342	333
R²	0.35	0.76	0.81

Note: Some important notes that can change.

I introduced `rmarkdown` because it is the classic approach and both procedures are very identical in terms of creating documents. The main difference stems from the YAML header because Quarto standardize the YAML code between different documents types. For example, consider the next console, it compare the YAML header of a default `rmarkdown` document with a default Quarto document. The latter has a `format` field instead of the `output` field, but will create an HTML document with both ways. Certainly there are more differences, but the main steps – code is evaluated via code-chunks, you need to format text via Markdown, etc. – are almost identical.

```
#rmarkdown YAML
---
title: "Untitled"
output: html_document
---
#Quarto YAML
title: "Untitled"
format: html
```

Thus, also consider Quarto if you start to create documents on a regular basis, because it provides many templates and an excellent documentation. You can create documents from RStudio after you installed Quarto. Click on the Quarto logo to visit the website.

7.4 Summary

Keep the following R packages, functions, and key insights from Chapter 8 in mind:

- Create many different documents with `rmarkdown` and adjust the document in the meta section (YAML header)

- Pandoc runs in the background, converts the file and uses, for example, Latex in case of a PDF file

- Adjust code via chunk-options (e.g., `eval`, `echo`, `warning`, `message`)

- Format text with Markdown and RStudio's visual markdown editing mode

- Start using a citation manager (e.g., Zotero)

- Create table with, for example, the `flextable`, the `modelsummary`, and the `stargazer` package

- Create a huxtable to display model output (`huxtable::huxreg`)

- Read R code from a file, a connection or expressions (`source`)

8 Automate work

Welcome to the 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.

Chapter 10 emphasized that we are not supposed to repeat our-self in terms of writing code and I highlighted that R can be used to do the boring stuff. To this end, we made dynamic reports with `rmarkdown` and we focused on the process of creating and sending them automatically (Allaire et al. 2022).

Unfortunately, we cannot build a document in a tutorial, but we can level up your skills to automate work. We made only a graph to illustrate the principle of dynamic reports in Chapter 10, but we can further improve the automation process by writing our own functions for this task. Thus, we will learn to automate the process of creating graphs and explore `purrr` once more (Henry and Wickham 2022). In addition, I briefly introduce packages that let us automate other boring tasks.

```r
# Tutorial 8 needs the following packages:
library(dplyr)
library(ggplot2)
library(magick)
library(palmerpenguins)
library(purrr)
library(tesseract)
```

8.1 Automate graphs

Suppose someone asked you to create a report with many descriptive graphs. If the report contains many graphs - say a histogram for all numerical variables - the chance is high that we start to repeat ourselves in terms of code. You may create a customized plot with `ggplot2` (Wickham, Chang, et al. 2022), but for each variable we still need to copy the code and then change the plot slightly. Such a *copy and paste* approach makes the code messy and we repeatedly need to adjust each graph manually if we find an error.

For this reason, I introduced the advantages of dynamic reports, but we can also make functions that help us to create graphs more efficiently. With functions, we minimize the urge to repeat ourselves. They make our work less error prone because if the function includes a mistake, the error would appear each time we call the function, making it easier to spot them. Therefore, our work becomes more flexible, because we have to change the code only in one place and repeat it over and over for an entire document.

Suppose you need to visualize all numerical variables from the `penguins` data with a histogram (Horst, Hill, and Gorman 2022). We will use the next code as an example to build functions. It is not important what the plots actually looks like, as we can still adjust all kind of graphical appearances and include such steps in the created function. Let's keep it simple to illustrate how the approach works. As a bare minimum let's pick a numerical variable (e.g., `bill_length_mm`) and give the plot a title.

```
# Pick a variable and give a title
ggplot(penguins) +
  geom_histogram(aes(x = bill_length_mm)) +
  ggtitle("Bill length")
```
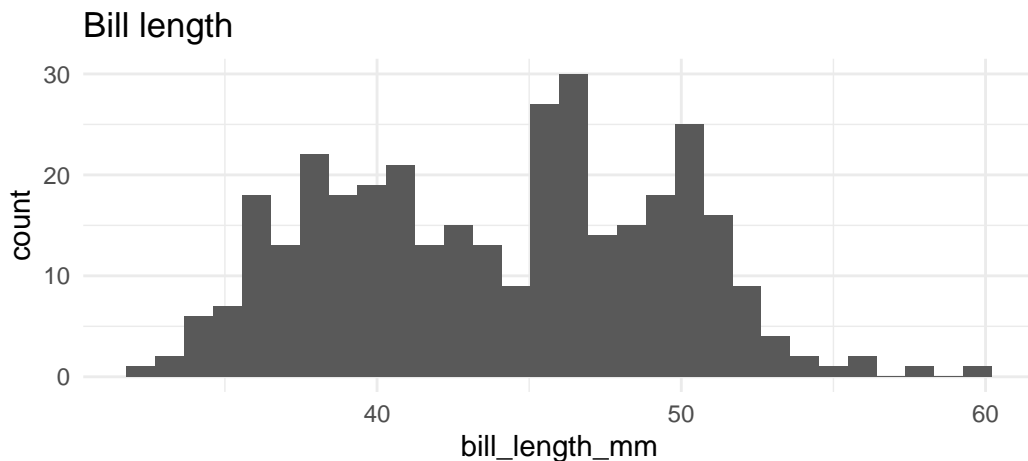

I already created the body of a function to build such a histogram (`hist_fun`) and you need to add the code to create the graph. First, insert the code from the last console. Next, we need to adjust the input variable. There are different ways to use `ggplot2` in functions. Adjust the code and fill in `.data[[x]]` instead of `x`, it will hand over the column vector from the function call. As always, give it a try before you consider the solution and check if the function is working.

```
# A function to create histograms
hist_fun <- function(data, x) {
  ggplot(data) +
    geom_histogram(aes(x = .data[[x]])) +
    ggtitle(paste("Var:", x))
}

# Did it work?
hist_fun(penguins, x = "bill_length_mm")
```
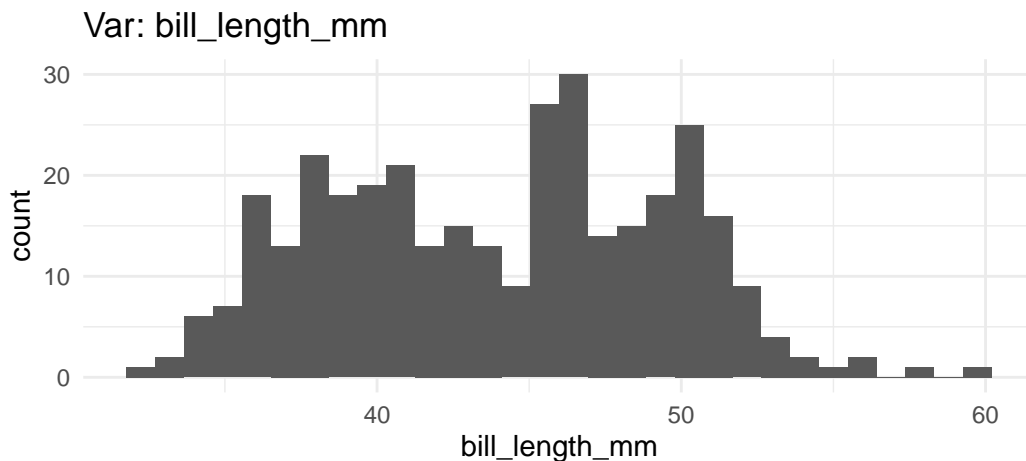

Functions reduce our workload, but the approach is quite repetitive, especially if we display each numerical variable with a histogram. Why don't we create one plot with a histogram for each numerical variable? We used the `DataExplorer` package for this task in Chapter 3, but with your own function you will be able to create a plot in the exact same way as you want them.

First we need to identify which variables of a data frame are numerical. The `select_if()` function from `dplyr` lets us pick variables under a specified condition (Wickham, François, et al. 2022). As the next console shows, it returns variables which are double (numerical, factor) if we insert the `is.double` (`is.numeric`; `is.factor`) function.

```
# dplyr::select_if
dplyr::select_if(penguins, is.double) |>
  head()


#> # A tibble: 6 x 2
#>   bill_length_mm bill_depth_mm
```

```
#>              <dbl>        <dbl>
#> 1            39.1         18.7
#> 2            39.5         17.4
#> 3            40.3         18
#> 4            NA           NA
#> 5            36.7         19.3
#> 6            39.3         20.6
```

However, we need a vector with the corresponding variable names to create the plot. Insert the `select_if` function into the `names()` function to get the variable names.

```
# Only numerical input
names(dplyr::select_if(penguins, is.double))
```

```
#> [1] "bill_length_mm" "bill_depth_mm"
```

Now that we have a vector with variable names and a function to create histograms, enables us to use `purrr` (Henry and Wickham 2022). The `map` function returns a list with the histogram for each input variable. First, assign the results of the last console as `numerical_variables`. Next, we apply the `hist_fun()` for each input of `numerical_variables`.

```
# Iterate with purrr
numerical_variables <- names(dplyr::select_if(penguins, is.double))
plots_list <- purrr::map(numerical_variables, ~ hist_fun(penguins, x = .x))
```

Finally, use the `plot_grid` function from the `cowplot` package to combine them all in one graph (Wilke 2020). The function only needs a `plotlist` in order to combine all the plots from the created list (`plots_list`) in one graphical output.

```
# Insert your plotlist
cowplot::plot_grid(plotlist = plots_list)
```
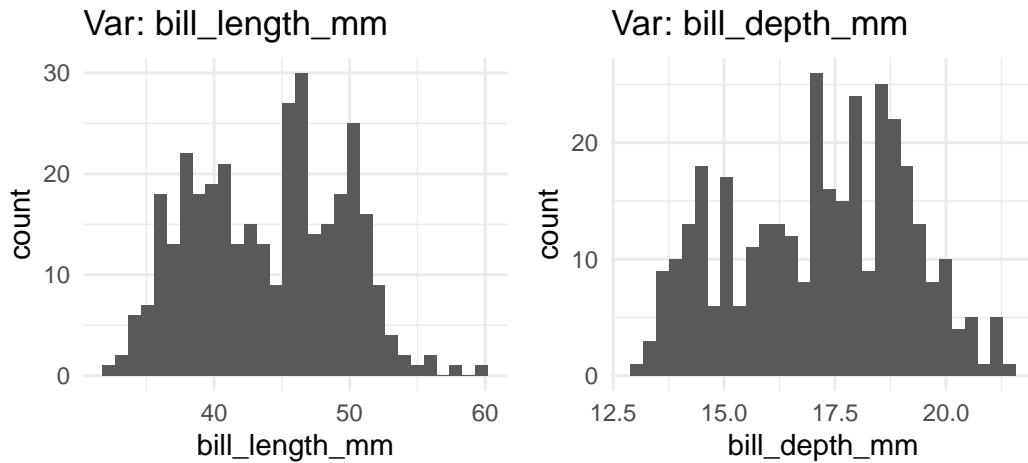
Now it's up to you to combine these steps. Create a function (`all_hist`) that takes all numerical variables from a data frame, then create a histogram for each variable with `purrr` and combine all into a single graph with `cowplot`. Finally, check if the function works.

```r
# Create a function that returns several histograms
all_hist <- function(data) {
  numerical_variables <- names(dplyr::select_if(data, is.double))
  plots_list <- purrr::map(numerical_variables, ~ hist_fun(data, x = .x))
  plot <- cowplot::plot_grid(plotlist = plots_list)
  return(plot)
}

# Did it work?
all_hist(penguins)
```
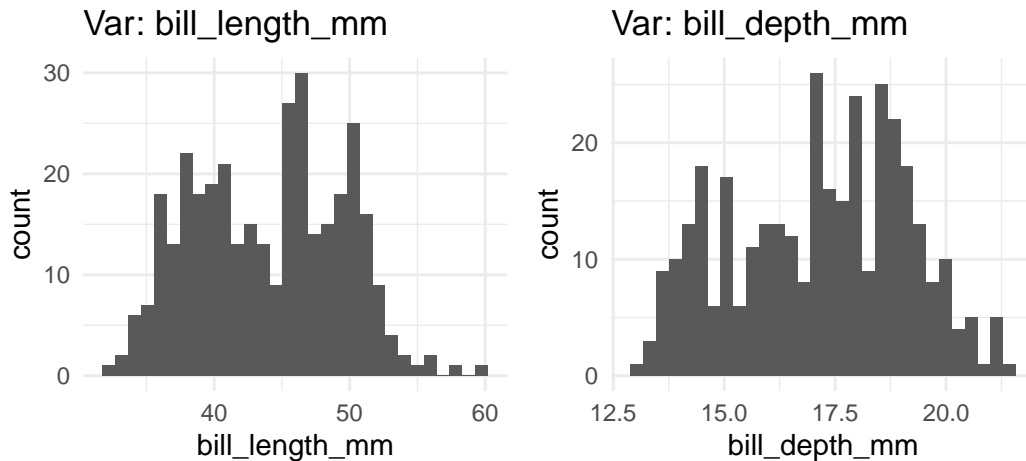
Let us create an error message and adjust the options to illustrate how to improve the approach. So far our function works only with numerical variables, but what happens if the data does not contain one? The first two variables of the penguins data are categorical which gives us the opportunity to inspect which error our function returns if there is no numerical input.

```r
# What happens if there is an error?
all_hist(penguins[1:2])
```

```
#> Error in grobs[[i]]: subscript out of bounds
```

The error message is pretty obscure, but we can improve it: It should warn us in a reasonable manner and abort the function. Consider the `numerical_variables` vector. If there are no numerical variables, it has a length of zero and the function is supposed to abort.

```r
# names_num is zero if the data has no numerical input
numerical_variables <- names(dplyr::select_if(penguins[1:2], is.double))
numerical_variables
```

```
#> character(0)
```

The **cli** package provides helpers for developing *command line interfaces* and we can use the `cli_abort` function from it (Csárdi 2022). It aborts the function call and returns a warning message as the next console illustrates.

```r
# CLI provides helpers for developing Command Line Interfaces
cli::cli_abort("What is the problem?")
```

```
#> Error:
#> ! What is the problem?
```

Insert the `cli_abort()` function together with an `if` condition in the `all_hist()` function. Only if the `names_num` vector has a length of zero, the `cli_abort` function should abort the function call and returns a warning.

```r
# Add if and cli::cli_abort
all_hist <- function(data) {
  numerical_variables <- names(dplyr::select_if(data, is.double))
  if (length(numerical_variables) == 0) {
    cli::cli_abort("Input must be a double-precision vector.")
  }
  plots_list <- purrr::map(numerical_variables, ~ hist_fun(data, x = .x))
  plot <- cowplot::plot_grid(plotlist = plots_list)
  return(plot)
}
# Do we get an error?
all_hist(penguins[1:2])
```

```
#> Error in `all_hist()`:
#> ! Input must be a double-precision vector.
```

It goes without saying that we can further improve the function by adding and adjust the options that generates the plot.

For example, add the `ncol` option in the `plot_grid()` function. It lets us determine the number of columns used to plot the graphs. Don't forget to insert the option also in the `function()` and give a default value that suits your purpose.

```r
# Adjust the plot_grid function
all_hist <- function(data, ncol = 3) {
  numerical_variables <- names(dplyr::select_if(data, is.double))
  if (length(numerical_variables) == 0) {
    cli::cli_abort("Input must be a double-precision vector.")
  }
  plots_list <- purrr::map(numerical_variables, ~ hist_fun(data, x = .x))
  cowplot::plot_grid(plotlist = plots_list, ncol = ncol)
```

```
}

# The all_hist with ncol
all_hist(iris, ncol = 4)
```
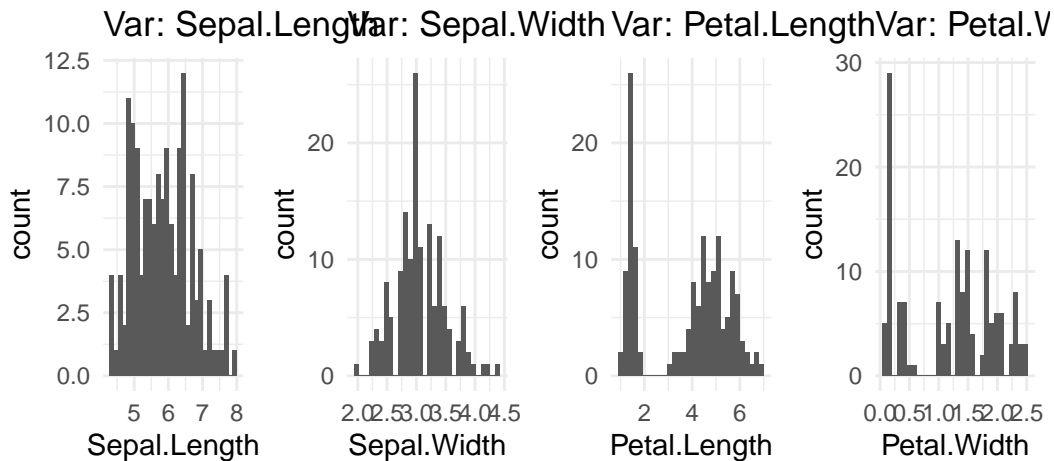

Finally, one last thought about the themes. If we adjust the `theme()` inside the function, the same styling rules are applied. However, we can also create our own `theme` function which increases the flexibility, since we can also apply the theme also to other graphs.

Creating a theme is not complicated. We best start by using a predefined theme and only adjust it where the theme does not fit for our purpose. As the next console shows, the `my_theme()` function relies on a `ggthemes` theme (Arnold 2021); to give you an idea how it works I only adjusted the text size of the title and the caption. After we created the theme, we can integrate it in the function or call it after we created the graph, as the next console shows.

```
# Create a customized theme
my_theme <- function() {
  ggthemes::theme_gdocs() +
    theme(plot.title = element_text(size = 16)) +
    theme(plot.caption = element_text(size = 10))
}

# Apply it where needed
plot <- all_hist(iris, ncol = 2)
plot + my_theme()
```
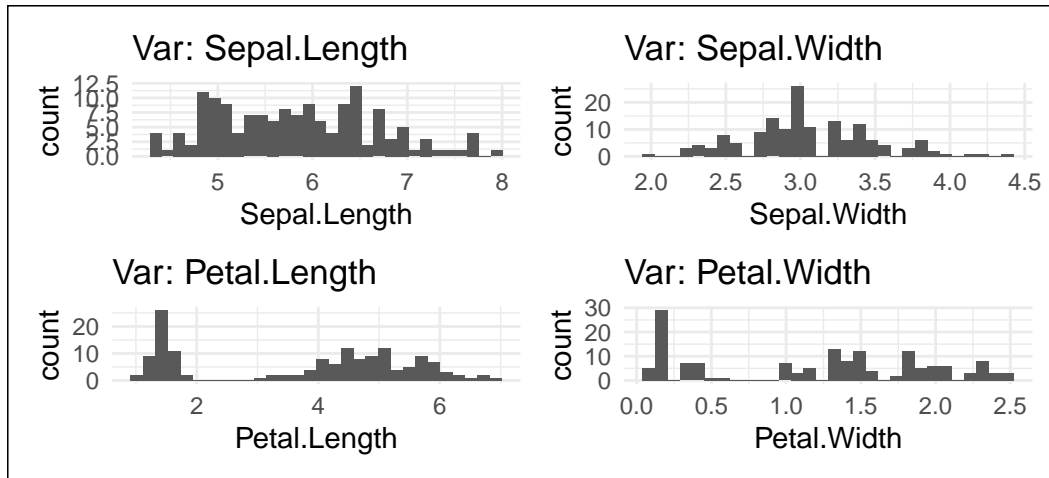
8.2 Automate the boring stuff

Students in the social sciences learn R to apply statistics, but R helps us also to automate repetitive tasks. This last section does not necessarily have applied empirical research in mind, but it tries to raise awareness about how we can get the boring stuff done (with R and different packages).

8.2.1 The officer package

Till now we focused on PDF reports, but the `officeverse` package encompasses several packages and functions to work with Microsoft Office files. For example, the `officer` package will help you to create and change all sorts of MS Office documents.

Suppose that a data set gets an update and we are supposed to do the same with the corresponding Word reports. There is one Word document for a long list of countries and we certainly do not want to make the update manually. For the sake of simplicity, suppose we only need to add a new page, insert a new headline, and provide an updated plot for each country. Therefore, we first need a new plot that we are supposed to add:

```
# New plot
plot <- ggplot(penguins) +
  geom_histogram(aes(x = bill_length_mm)) +
  ggtitle("Bill length")
```

The next console shows how such a minimal but effective update may look like based on a simple workaround. First, we read the document with `read_docx`; next we add a new page

(`body_add_break`), a new header (`body_add_par`), and the plot (`body_add_gg`). Finally, we are able to export the new document with the `print()` function and the `target` option.

```r
library(officer)
# Read and update my_doc
doc_updated <- read_docx(path = "my_doc.docx") |>
  body_add_break(pos = "after") |>
  body_add_par("Updated results", style = "heading 1") |>
  body_add_gg(value = plot, style = "normal")

# Save updated doc
print(doc_updated, target = "doc_updated.docx")
```

Of course, this was a simple workaround for illustration purposes. Consider the package website for more information about the `officeverse`.

8.2.2 The pdftools package

Say you have a bunch of PDF files and you need to combine them in a single file. Or the other way around, you have a large PDF file but for some reasons you need to split them. Certainly there are different software solutions available, but your time is too precious to add each page manually or apply another repetitive task to the file(s). The `pdftools` and the `qpdf` package has corresponding functions to combine, split, and perform further functions when working with PDF files (Ooms 2022a, 2022b).

```r
# Join several pdf files into one
qpdf::pdf_combine()

# Split a single pdf into separate files, one for each page
qpdf::pdf_split()
```

In the case of image files, consider the `magick` package (Ooms 2021).

8.2.3 The magick package

Suppose you have many images that are saved a PDF file and you need to convert them into *png* files. As before, you can open all files in a graphic software and export each of them manually. Or you can use the `magick` package to read the files and convert them into a certain format, as the next console highlights. The package provides many features to read, adjust, and convert image files. Consider the package documentation because it outlines more features that I can introduce.

```
# Convert Images
img <- image_read_pdf("figure.pdf")

# Write as PNG
image_write(img, "output.png")
```

Chapter 10 introduced packages and features to automate work. Certainly, we cannot automate all kinds of work with R, but keep in mind that there are often solutions are available to get recurring tasks done, before you start to repeat yourself several times. I did not introduce the `magick` package only because of it lets us convert images. I did so because it has many features to work with images and sometimes you will be surprised about the numerous possibilities of R and its landscape.

Suppose you want to analyze texts, but they are only available as an image files. I made a screen shot of an info box from the Practice R book. The image shows the first lines of the `cronR` info box.

The cronR package

Cron lets you execute processes (or code) at a certain time on Unix-based operating systems (or alternatively use a task scheduler for Windows). The cronR package comes with a convenient addin, as Figure 10.5 shows. Select the R script and pick a time for the Cron job to run. The cronR package waits until the launch time and runs the script automatically. Keep in mind that this approach is only working if your computer is not switched off, but a cloud-based implementation (e.g., via GitHub Actions) of the Cron job lifts that restriction.

The `magick` package helps us to extract this text from the image files. As the next console shows, we need to read the image first before we can extract the text.

```
# Read image
img <- image_read("https://raw.githubusercontent.com/edgar-treischl/...
.../Scripts_PracticeR/main/Images/text.png")
```

The `image_ocr()` function (optical character recognition) scans the file and extract the data. The functions relies on `tesseract`, which is an open OCR engine that supports over 100 languages. To make the output a bit easier to read in the console, I used the `stringr` package, which I will introduce in Chapter 11 in detail.

```
# Extract text via OCR
library(tesseract)
library(stringr)
txt <- image_ocr(img)
```

```
text <- str_split(txt, "\n")
text <- as_vector(text)
head(text)


#> [1] "The cronR package"
#> [2] ""
#> [3] "Cron lets you execute processes (or code) at a certain time on Unix..."
#> [4] "alternatively use a task scheduler for Windows). The cronR package ..."
#> [5] "Figure 10.5 shows. Select the R script and pick a time for the Cron..."
#> [6] "until the launch time and runs the script automatically. Keep in mi..."
```

Instead of repeating yourself, use your time wiser and learn something new about R and its environment. How about D3 (Data-Driven Documents), which is a JavaScript library to create interactive visualizations for the web. The **r2d3** package let you integrate D3 into R (Strayer, Luraschi, and Allaire 2022). The next console illustrates that the package runs a JavaScript file (`voronoi.js`) and returns an example made by Mike Bostock.

```
library(r2d3)
# voronoi.js is based on: https://bl.ocks.org/mbostock/4060366
r2d3(d3_version = 4, script = "d3/voronoi.js")
```
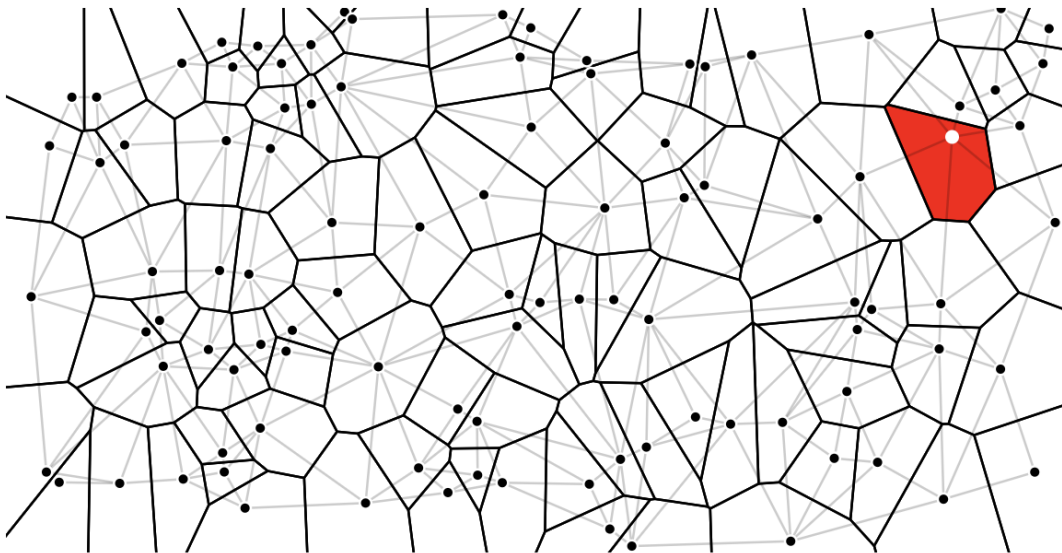

If you are not in the right mood to learn something new, how about playing an old school video game? The `Rcade` package lives on GitHub only, but thanks to Romain Lesur you can play Tetris, Mario, or Pacman directly from R.

```
# devtools::install_github('RLesur/Rcade')
# library(Rcade)
Rcade::games$Pacman
```

Thus, be curious and don't miss the opportunity to automate work that does not need your full attention, because it will also improve your skills. In a similar sense, consider to write your own R package if you have invested a lot of time and effort in your work. Depending on your goal, this might not be necessary but all the functions would be available if you combine them in a package. Moreover, I couldn't resist to highlight the possibility one more time, especially regarding the *American Chopper* meme.

Don't let fancy packages such as `ggplot2` discourage you when considering to create your own package for the first time: you and your ideas are worth the time and effort. Keep that in mind if you start fooling around with the idea.

8.3 Summary

Keep the following R functions and packages from Chapter 10 in mind:

- Keep distinct/unique rows (`dplyr::distinct`)
- Concatenate strings (`paste`)
- Format and interpolate a string (`glue`: Hester and Bryan 2022)
- Render R Markdown (`rmarkdown::render`)
- Find your files (`here::` Müller 2020)
- Control flow (e.g., `if, for loop`)
- Play a short sound (`beepr`: Bååth 2018)
- Automatic reporting of R objects (`report::` Makowski, Lüdecke, et al. 2022)
- Send an email (`blastula`: Iannone and Cheng 2020)
- Interpret input text as Markdown-formatted text (`blastula::md`)
- List the Files in a directory/folder (`list.files`)
- Apply a function to each element of a vector (`purrr::map`)

9 Collect data

Welcome to the collect data 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.

We extracted data from a PDF, I outlined the basics about web scraping, and we got in touch with APIs in Chapter 11. As outlined, to collect data offers unique opportunities for applied empirical research, but can be very tricky, especially web scraping becomes quickly complicated.

Regardless of the approach to collect data, I introduced the **stringr** package and its main functions before we extracted information from a PDF file and worked with unstructured data from HTML files (Wickham 2022d). To give you a compact overview about the many **str_*** functions, this tutorial is dedicated to the **stringr** package: We recapture the introduced functions and explore further possibilities how we can handle strings. The next console shows data and fictive email addresses from persons you may know from the Netflix series *Stranger Things*. Never mind if you are not familiar with the series, we will use the character variables such as the email addresses to work with **stringr**.

```r
# Libs for Tutorial 11
library(purrr)
library(stringi)
library(stringr)

# The stranger things example data
head(sf_data)
```

```
#> # A tibble: 6 x 5
#>   character        firstname     lastname   year email
#>   <chr>            <chr>         <chr>     <dbl> <chr>
#> 1 Eleven           Millie Bobby  Brown      2004 eleven@HawkinsLab.com
#> 2 Dustin Henderson Gaten         Matarazzo  2002 Dustin.Henderson@gmx.com
#> 3 Will Byers       Noah          Schnapp    2004 byers-castle@gmx.com
#> 4 Erica Sinclair   Priah         Ferguson   2006 Erica-Sinclair1@aol.com
#> 5 Martin Brenner   Matthew       Modine     1959 MBrenner@HawkinsLab.com2
#> 6 Jim Hopper       David         Harbour    1975 jim.hopper@hawkinspd.com
```

The `stringr` package increases your string powers tremendously, but we need to keep up with many `str_*` functions and names. All you have to do is pick the "right" function in this tutorial. For the compact overview, we focus on the sections of the package cheat sheet: (1) We detect matches; (2) we mutate strings; (3) we subset strings; (4) we join and split strings; and (5) we order strings and manage their length.

9.1 Detect matches

Suppose we want to create an online survey which is why we scraped `emails` of our participants such as in the fictive email addresses from the Stranger Things data. Unfortunately, the strings contain some minor mistakes that need to be fixed:

```
# Email examples
emails <- sf_data$email
emails
```

```
#> [1] "eleven@HawkinsLab.com"     "Dustin.Henderson@gmx.com"
#> [3] "byers-castle@gmx.com"      "Erica-Sinclair1@aol.com"
#> [5] "MBrenner@HawkinsLab.com2" "jim.hopper@hawkinspd.com"
#> [7] "Joyce-B@gmx.com"           "Mike@TheWheelers.com"
#> [9] "1nancy-wheeler92@gmx.com"
```

Notice, some email addresses start (end) with a number instead of letters. Those signs are not a part of the email address but refer to footnotes on the webpage where we scraped the data. Suppose we do not know how virulent this problem is, can you detect which one does not *start* (`str_starts`) or *end* (`str_ends`) with a letter?

```
# Does the string start with ...?
str_starts(emails, "[:alpha:]")
```

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

```
# Does the string end with ...?
str_ends(emails, "[:alpha:]")
```

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

Some of the email addresses are private, while others are from a company (e.g., HawkinsLab.com). If you need to know how many, use the str_count() function and build the sum. How many email addresses are from HawkinsLab.com?

```
# Count them
sum(str_count(emails, "HawkinsLab.com"))
```

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

Use the str_detect() function to detect all strings from the HawkinsLab.

```
# Detect strings
str_detect(emails, "@HawkinsLab.com")
```

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

The str_which() is also handy, it returns *at which position* we observe the search pattern.

```
# And at which position?
str_which(emails, "@HawkinsLab.com")
```

```
#> [1] 1 5
```

Suppose we need to extract the user names because we want to include them in the email invitation for the survey. In order to extract the names, *locate* the position of a string. Use the str_locate() to locate where the @ sign appears, because it splits the string into the user and the provider name.

```
# Locate a start and an end point (here @)
str_locate(emails, "@")
```

```
#>      start end
#> [1,]     7   7
#> [2,]    17  17
#> [3,]    13  13
#> [4,]    16  16
#> [5,]     9   9
#> [6,]    11  11
#> [7,]     8   8
#> [8,]     5   5
#> [9,]    17  17
```

In the next step we will use the position of the `@` sign to mutate the strings and to extract their user names.

9.2 Mutate strings

Let us first clean the email addresses. *Remove* strings that do not start or end with a letter but with a number, which is clearly an error. Very similar to the `str_replace()` function, the `str_remove()` searches the string, but it removes a match instead of performing a replacement. Can you still remember how to remove the digits from the beginning (`^`) and the end (`$`) of a string? Replace the `emails` vector and check if it worked.

```
# Remove strings
emails <- str_remove(emails, "^[:digit:]")
emails <- str_remove(emails, "[:digit:]$")

# Did it work?
emails
```

```
#> [1] "eleven@HawkinsLab.com"     "Dustin.Henderson@gmx.com"
#> [3] "byers-castle@gmx.com"      "Erica-Sinclair1@aol.com"
#> [5] "MBrenner@HawkinsLab.com"   "jim.hopper@hawkinspd.com"
#> [7] "Joyce-B@gmx.com"           "Mike@TheWheelers.com"
#> [9] "nancy-wheeler92@gmx.com"
```

We could use the `str_extract()` function and our regex knowledge to extract the user names, but regex are hard to build even in the case of a supposedly simple strings. The email addresses make this point clear: Each user name consist of one or several words; some have a separator between the first and the last name, some contains digits (or not), and the user name ends before the `@` sign. There is a much simpler solution to extract the user names, but nevertheless keep the `str_view_all()` function in mind if you are building a regex because it displays the strings in the viewer pane and highlights matched characters.

Instead of building a regex, we can use the `str_sub()` function to create a vector with the user names only. The function needs the strings, a start, and an endpoint to create the subset. For this purpose we already located the positions of the `@` sign with the `str_locate()` function. Thus, all user names start at the first position until the `@` sign appears in the string. I copied the code to locate the `@` sign and saved the results as `x`. Subset `x` to get a vector with the end position of the user name, then subset the `emails`.

```r
# Get and set substrings using their positions
x <- str_locate(emails, "@")
end <- x[, 1]
names <- str_sub(emails, 1, end - 1)
names
```

```
#> [1] "eleven"          "Dustin.Henderson" "byers-castle"     "Erica-Sinclair1"
#> [5] "MBrenner"        "jim.hopper"       "Joyce-B"          "Mike"
#> [9] "nancy-wheeler92"
```

Further steps to manipulate the strings might be easier to apply if all the user would have used the same style regarding their user names. Use the `str_replace()` function and replace the dashes with points.

```r
# Replace
str_replace(names, "-", ".")
```

```
#> [1] "eleven"          "Dustin.Henderson" "byers.castle"     "Erica.Sinclair1"
#> [5] "MBrenner"        "jim.hopper"       "Joyce.B"          "Mike"
#> [9] "nancy.wheeler92"
```

Depending on the purpose, it might also be useful to create a uniform formatting of the strings. Use one of the `str_to_*()` functions to make them *lower*, *upper*, or *title* case.

```r
# str_to_* (lower, upper, title)
str_to_lower(names)
```

```
#> [1] "eleven"          "dustin.henderson" "byers-castle"     "erica-sinclair1"
#> [5] "mbrenner"        "jim.hopper"       "joyce-b"          "mike"
#> [9] "nancy-wheeler92"
```

9.3 Subset strings

We used the `str_sub()` to split strings by their position, but the `str_subset()` function lets us create a subset for a search pattern. For example, consider all participants with an specific email account (e.g., `gmx`):

```
# Find matching elements
str_subset(emails, pattern = "gmx")
```

```
#> [1] "Dustin.Henderson@gmx.com" "byers-castle@gmx.com"
#> [3] "Joyce-B@gmx.com"          "nancy-wheeler92@gmx.com"
```

Furthermore, most of the time we use the `str_detect()` function to detect a pattern. For example, the functions shows us which input has a specific pattern and we can detect if an string has no `@` sign at all.

```
strings <- c(
  "Dustin Henderson",
  "hop@gmx.com jim.hopper@hawkinspd.com",
  "Erica-Sinclair@aol.com",
  "nancy-wheeler92@gmx.com"
)

is_email <- "@"

# Detect a pattern
str_detect(strings, is_email)
```

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

We used the function to illustrate the first few things about regular expressions. However, we do not need to filter the data and first detect the email addresses if we want to extract this information. Consider how the `str_extract()` and the `str_extract_all` function work. The function needs `strings` and a pattern (such as `is_email`). It shows us which string does (not) include the given pattern.

```
# Extract the complete match
str_extract(strings, is_email)
```

```
#> [1] NA  "@" "@" "@"
```

```
str_extract_all(strings, is_email)
```

```
#> [[1]]
#> character(0)
#>
#> [[2]]
#> [1] "@" "@"
#>
#> [[3]]
#> [1] "@"
#>
#> [[4]]
#> [1] "@"
```

Finally, the `str_match()` (and `str_match_all`) does essentially the same as `str_extract()`, but returns matches as matrix.

```
# Extract components (capturing groups) from a match
str_match(strings, is_email)
```

```
#>      [,1]
#> [1,] NA
#> [2,] "@"
#> [3,] "@"
#> [4,] "@"
```

9.4 Join and splits

The `stringr` package has join and split functions. Suppose we scraped the first and the last name of a person separately, but for the survey invitation we need to combine them. Use `str_c()` for this job and assign them as `names`. Combine the `firstname` with the `lastname` from the `sf_data`. Use a blank space as a separator (`sep`).

```
# Use str_c to combine strings
names <- str_c(sf_data$firstname, sf_data$lastname, sep = " ")
names
```

```
#> [1] "Millie Bobby Brown" "Gaten Matarazzo"    "Noah Schnapp"
#> [4] "Priah Ferguson"     "Matthew Modine"     "David Harbour"
#> [7] "Winona Ryder"       "Finn Wolfhard"      "Natalia Dyer"
```

Use the `str_split_fixed()` in the opposite scenario. Split the `names` vector from the last task: Use the blank space as a `pattern` and each name consist of two text chunks we want to split (`n`).

```
# Split strings
str_split_fixed(names, pattern = " ", n = 2)
```

```
#>        [,1]      [,2]
#>  [1,] "Millie"  "Bobby Brown"
#>  [2,] "Gaten"   "Matarazzo"
#>  [3,] "Noah"    "Schnapp"
#>  [4,] "Priah"   "Ferguson"
#>  [5,] "Matthew" "Modine"
#>  [6,] "David"   "Harbour"
#>  [7,] "Winona"  "Ryder"
#>  [8,] "Finn"    "Wolfhard"
#>  [9,] "Natalia" "Dyer"
```

We used the `str_sub()` function to extract the user names, but we could also use the `str_split()` function to split the strings before and after the `@` sign. Say we want to extract unique provider names this time. The `str_split()` function returns a list as the next console shows. Use the pipe and the `map_chr()` functions from `purrr` to get the first or second element of each list (Henry and Wickham 2022). Furthermore, apply the `stri_unique()` function from `stringi` to examine unique provider names only (Gagolewski et al. 2022).

```
# Split email, get provider names, but only unique ones
str_split(emails, pattern = "@") |>
  purrr::map_chr(2) |>
  stringi::stri_unique()
```

```
#> [1] "HawkinsLab.com"  "gmx.com"        "aol.com"        "hawkinspd.com"
#> [5] "TheWheelers.com"
```

The `glue` package offers some useful features to work with strings, especially if we create texts and documents. Suppose we want to create a sentence that describe how old a person like *Jim Hopper* is. I already calculated his age (`hopper_age`); use the `paste` function to create a sentences that describes how old he is.

```
# Traditional approach
hopper_age <- lubridate::year(Sys.time()) - sf_data$year[6]
paste("Jim Hopper is", hopper_age, "years old.")
```

```
#> [1] "Jim Hopper is 49 years old."
```

Did you realize that we need a lot of quotation marks and that we need to be careful not to introduce any error. The `str_glue()` tries to improve this case. We can refer to objects with curved braces without further ado.

```
# Glue strings
str_glue("Hop is {hopper_age} years.")
```

```
#> Hop is 49 years.
```

One step further goes the `str_glue_data()` function. It returns strings for each observation of a data set. For example, build a sentence that outlines the `firstname`, `lastname` and the birth `year` of the Stranger Things actors.

```
# Glue strings from data
str_glue_data(sf_data, "- {firstname} {lastname} is born in {year}.")
```

```
#> - Millie Bobby Brown is born in 2004.
#> - Gaten Matarazzo is born in 2002.
#> - Noah Schnapp is born in 2004.
#> - Priah Ferguson is born in 2006.
#> - Matthew Modine is born in 1959.
#> - David Harbour is born in 1975.
#> - Winona Ryder is born in 1971.
#> - Finn Wolfhard is born in 2002.
#> - Natalia Dyer is born in 1995.
```

Finally, the package offers functions to order strings and manage their length.

9.5 Length and order

Do not forget that `stringr` comes with example strings (`fruit`, `sentences`) that lets you test the functions before you run them in the wild, but of course we can also build our own `fruits`. So, do you remember how we can estimate the length of strings?

```
# Length of a string
fruits <- c("banana", "apricot", "apple", "pear    ")
str_length(fruits)
```

```
#> [1] 6 7 5 9
```

Unfortunately, the `fruits` vector includes an mistake. There is a lot of white space around the last fruit. Do you know how to get rid of such noise.

```
# Trim your strings
fruits <- str_trim(fruits)
fruits
```

```
#> [1] "banana"  "apricot" "apple"    "pear"
```

Finally, order (`str_order`) and sort (`str_sort`) the fruits.

```
# Order strings
str_order(fruits)
```

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

```
# Sort strings
str_sort(fruits, decreasing = F)
```

```
#> [1] "apple"    "apricot" "banana"  "pear"
```

9.6 Summary

Keep also the following functions and packages from Chapter 11 in mind:

- PDF utilities (e.g., `pdf_text`, Ooms 2022a)
- Coerce a list to a vector (`purrr::as_vector`)
- The Names of an object (`names`)
- Subset rows using their positions (`dplyr::slice_*`, Wickham, François, et al. 2022)
- Bind multiple data frames by row (`dplyr::bind_rows`)
- The `rvest` package and its functions for web scraping (e.g., `read_html, html_table`, Wickham 2022c)
- The `httr` package and its functions for receive information from a website (`GET, content`, Wickham 2022b)
- Create an API with the `plumber` package (Schloerke and Allen 2022)

References

Allaire, JJ, Yihui Xie, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, Hadley Wickham, Joe Cheng, Winston Chang, and Richard Iannone. 2022. *rmarkdown: Dynamic Documents for R.* https://CRAN.R-project.org/package=rmarkdown.

Arel-Bundock, Vincent. 2023. *modelsummary: Summary Tables and Plots for Statistical Models and Data: Beautiful, Customizable, and Publication-Ready.* https://CRAN.R-project.org/package=modelsummary.

Arnold, Jeffrey B. 2021. *ggthemes: Extra Themes, Scales and Geoms for ggplot2.* https://CRAN.R-project.org/package=ggthemes.

Bååth, Rasmus. 2018. *Beepr: Easily Play Notification Sounds on any Platform.* https://CRAN.R-project.org/package=beepr.

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.

Blair, Graeme, Jasper Cooper, Alexander Coppock, Macartan Humphreys, and Luke Sonnet. 2022. *estimatr: Fast Estimators for Design-Based Inference.* https://CRAN.R-project.org/package=estimatr.

Cairo, Alberto. 2016. *The Truthful Art: Data, Charts, and Maps for Communication.* New Riders.

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

Csárdi, Gábor. 2022. *cli: Helpers for Developing Command Line Interfaces.* https://CRAN.R-project.org/package=cli.

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

Firke, Sam. 2021. *janitor: Simple Tools for Examining and Cleaning Dirty Data.* https://CRAN.R-project.org/package=janitor.

Gagolewski, Marek, Bartek Tartanus, others; Unicode, Inc., et al. 2022. *stringi: Fast and Portable Character String Processing Facilities.* https://CRAN.R-project.org/package=stringi.

Garnier, Simon. 2021. *viridis: Colorblind-Friendly Color Maps for R.* https://CRAN.R-project.org/package=viridis.

Gohel, David, and Panagiotis Skintzos. 2022. *flextable: Functions for Tabular Reporting.* https://CRAN.R-project.org/package=flextable.

Henry, Lionel, and Hadley Wickham. 2022. *purrr: Functional Programming Tools.* https://CRAN.R-project.org/package=purrr.

Hester, Jim, and Jennifer Bryan. 2022. *glue: Interpreted String Literals.* https://CRAN.R-project.org/package=glue.

Hlavac, Marek. 2022. *stargazer: Well-Formatted Regression and Summary Statistics Tables.* https://CRAN.R-project.org/package=stargazer.

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

Hothorn, Torsten, Achim Zeileis, Richard W. Farebrother, and Clint Cummins. 2022. *lmtest: Testing Linear Regression Models.* https://CRAN.R-project.org/package=lmtest.

Hugh-Jones, David. 2022. *huxtable: Easily Create and Style Tables for LaTeX, HTML and Other Formats.* https://CRAN.R-project.org/package=huxtable/.

Hvitfeldt, Emil. 2021. *paletteer: Comprehensive Collection of Color Palettes.* https://CRAN.R-project.org/package=paletteer.

Iannone, Richard, and Joe Cheng. 2020. *blastula: Easily Send HTML Email Messages.* https://CRAN.R-project.org/package=blastula.

Iannone, Richard, Joe Cheng, Barret Schloerke, Ellis Hughes, and JooYoung Seo. 2022. *gt: Easily Create Presentation-Ready Display Tables.* https://CRAN.R-project.org/package=gt.

Kunst, Joshua. 2022. *highcharter: A Wrapper for the Highcharts Library.* https://CRAN.R-project.org/package=highcharter.

Long, Jacob A. 2021. *interactions: Comprehensive, User-Friendly Toolkit for Probing Interactions.* https://CRAN.R-project.org/package=interactions.

———. 2022. *jtools: Analysis and Presentation of Social Scientific Data.* https://CRAN.R-project.org/package=jtools.

Lüdecke, Daniel, Dominique Makowski, Mattan S. Ben-Shachar, Indrajeet Patil, Philip Waggoner, and Brenton M. Wiernik. 2022. *performance: Assessment of Regression Models Performance.* https://CRAN.R-project.org/package=performance.

Lüdecke, Daniel, Dominique Makowski, Indrajeet Patil, Mattan S. Ben-Shachar, Brenton M. Wiernik, and Philip Waggoner. 2022. *see: Model Visualisation Toolbox for easystats and ggplot2.* https://CRAN.R-project.org/package=see.

Makowski, Dominique, Daniel Lüdecke, Mattan S. Ben-Shachar, Indrajeet Patil, and Brenton M. Wiernik. 2022. *report: Automated Reporting of Results and Statistical Models.* https://CRAN.R-project.org/package=report.

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. 2020. *here: A Simpler Way to Find Your Files.* https://CRAN.R-project.org/package=here.

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.

Neuwirth, Erich. 2022. *RColorBrewer: ColorBrewer Palettes.* https://CRAN.R-project.org/package=RColorBrewer.

Ooms, Jeroen. 2021. *magick: Advanced Graphics and Image-Processing in R.* https://CRAN.

R-project.org/package=magick.

———. 2022a. *pdftools: Text Extraction, Rendering and Converting of PDF Documents.* https://CRAN.R-project.org/package=pdftools.

———. 2022b. *qpdf: Split, Combine and Compress PDF Files.* https://CRAN.R-project.org/package=qpdf.

Patil, Indrajeet. 2023. *ggstatsplot: ggplot2 Based Plots with Statistical Details.* https://CRAN.R-project.org/package=ggstatsplot.

Pedersen, Thomas Lin. 2022a. *ggforce: Accelerating ggplot2.* https://CRAN.R-project.org/package=ggforce.

———. 2022b. *patchwork: The Composer of Plots.* https://CRAN.R-project.org/package=patchwork.

Qiu, Yixuan. 2022. *showtext: Using Fonts More Easily in R Graphs.* https://CRAN.R-project.org/package=showtext.

Schloerke, Barret, and Jeff Allen. 2022. *plumber: An API Generator for R.* https://CRAN.R-project.org/package=plumber.

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.

Sievert, Carson, Chris Parmer, Toby Hocking, Scott Chamberlain, Karthik Ram, Marianne Corvellec, and Pedro Despouy. 2022. *plotly: Create Interactive Web Graphics via plotly.js.* https://CRAN.R-project.org/package=plotly.

Solt, Frederick, and Yue Hu. 2021. *dotwhisker: Dot-and-Whisker Plots of Regression Results.* https://CRAN.R-project.org/package=dotwhisker.

Strayer, Nick, Javier Luraschi, and JJ Allaire. 2022. *R2d3: Interface to D3 Visualizations.* https://CRAN.R-project.org/package=r2d3.

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. 2021. *babynames: US Baby Names 1880-2017.* https://github.com/hadley/babynames.

———. 2022a. *forcats: Tools for Working with Categorical Variables (Factors).* https://CRAN.R-project.org/package=forcats.

———. 2022b. *httr: Tools for Working with URLs and HTTP.* https://CRAN.R-project.org/package=httr.

———. 2022c. *rvest: Easily Harvest (Scrape) Web Pages.* https://CRAN.R-project.org/package=rvest.

———. 2022d. *stringr: Simple, Consistent Wrappers for Common String Operations.* https://CRAN.R-project.org/package=stringr.

Wickham, Hadley, and Jennifer Bryan. 2022. *readxl: Read Excel Files.* https://CRAN.R-

project.org/package=readxl.

Wickham, Hadley, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, and Dewey Dunnington. 2022. *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics.* https://CRAN.R-project.org/package=ggplot2.

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

Wickham, Hadley, and Maximilian Girlich. 2022. *tidyr: Tidy Messy Data.* https://CRAN.R-project.org/package=tidyr.

Wickham, Hadley, Jim Hester, and Jennifer Bryan. 2022. *readr: Read Rectangular Text Data.* https://CRAN.R-project.org/package=readr.

Wickham, Hadley, Evan Miller, and Danny Smith. 2022. *haven: Import and Export SPSS, Stata and SAS Files.* https://CRAN.R-project.org/package=haven.

Wilke, Claus O. 2020. *cowplot: Streamlined Plot Theme and Plot Annotations for ggplot2.* https://CRAN.R-project.org/package=cowplot.

Xie, Yihui, Joe Cheng, and Xianying Tan. 2022. *DT: A Wrapper of the JavaScript Library DataTables.* https://CRAN.R-project.org/package=DT.

Zhu, Hao. 2021. *kableExtra: Construct Complex Table with kable and Pipe Syntax.* https://CRAN.R-project.org/package=kableExtra.