First recap session

Claudius Gräbner-Radkowitsch

2025-04-11

Table of contents


```
library(DataScienceExercises)
```

1 Storing and presenting table-like data

> **Question:** After going through part 2 of the lab I am now a bit confused about the many ways to display data. Are there rules when to use the following options or is it just personal preference? `table()`, `tibble::as_tibble()`, `data.frame()`, `dplyr::glimpse()`, `head()`, and `View()`.

The classical way to store table-like data is the `data.frame`:

```
df_1 <- data.frame(
  "gender" = c(rep("male", 5), rep("female", 4)),
  "height" = c(189, 175, 180, 179, 201, 155, 166, 150, 172)
)
```

You can also use the `head()` function to get a quick overview of the first rows of your data set:

```
head(df_1)
```

```
  gender height
1   male    189
2   male    175
3   male    180
4   male    179
5   male    201
6 female    155
```

If you want to inspect the whole data set in an Excel-like fashion in R Studio, you can use `View()`:

```
View(df_1)
```

Often, it is a good idea to use a `tibble` instead of a `data.frame`. While it looks similar, there are a couple of practical advantages of using `tibbles`.

You transform a `data.frame` into a `tibble` using the `tibble::as_tibble()` function:

```
tb_1 <- tibble::as_tibble(df_1)
tb_1
```

```
# A tibble: 9 x 2
  gender height
  <chr>   <dbl>
1 male      189
2 male      175
3 male      180
4 male      179
5 male      201
6 female    155
7 female    166
8 female    150
9 female    172
```

Especially if you have large data sets, you might want to use the `dplyr::glimpse()` function to get an overview over the overall structure of the data set:

```
dplyr::glimpse(tb_1)
```

```
Rows: 9
Columns: 2
$ gender <chr> "male", "male", "male", "male", "male", "female", "female", "fe~
$ height <dbl> 189, 175, 180, 179, 201, 155, 166, 150, 172
```

A similar function is `str()`:

```
str(tb_1)
```

```
tibble [9 x 2] (S3: tbl_df/tbl/data.frame)
 $ gender: chr [1:9] "male" "male" "male" "male" ...
 $ height: num [1:9] 189 175 180 179 201 155 166 150 172
```

Beware: There is also a function `table()`, but it is used in a slightly different way. You can use it to create *frequency tables*:

```
table(df_1$gender)
```

```
female   male
     4      5
```

2 Extracting elements from lists

> **Question:** I had some difficulty with the coursework task on extracting from lists. It would be helpful to go over that again.

Lists are objects that can contain elements of different types. This distinguishes them from atomic vectors.

For instance, the following does not work:

```
atomic_vec <- c(1, 5, "Hey!", 99)
```

Or, at least it does not work as one might have intended. Because atomic vectors can only accommodate one type, all elements are coerced to the most basic type that occurs. In this case:

```
atomic_vec
```

```
[1] "1"    "5"    "Hey!" "99"
```

```
typeof(atomic_vec)
```

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

Lists, on the other hand, can accomodate vectors of different types and lengths:

```
l_1 <-  list(1, 5, "Hey!", 99)
l_1
```

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

[[2]]
[1] 5

[[3]]
[1] "Hey!"

[[4]]
[1] 99
```

Especially for more complex lists, the elements are often names:

```
l_2 <- list(
  "first_numbers" = c(1, 2, 3, 4),
  "second_characters" = c("a", "b", "c"),
  "third_logical" = c(TRUE, FALSE, TRUE, TRUE, TRUE)
)
l_2
```

```
$first_numbers
[1] 1 2 3 4

$second_characters
[1] "a" "b" "c"

$third_logical
[1]  TRUE FALSE  TRUE  TRUE  TRUE
```

There are three ways to extract elemnts from a list, two of which are synonymous:

1. Use `[]` to extract a list element by its name or index. This returns a **list**:

```
l_2["second_characters"] # same effect: l_2[2]
```

```
$second_characters
[1] "a" "b" "c"
```

2. Use `[[]]` to extract a list element by its name or index. This returns the **element**:

```
l_2[["second_characters"]] # same effect: l_2[[2]]
```

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

3. Use `$` to extract a list element by its name. This returns the **element** (so its equivalent to option 2):

```
l_2$second_characters # same effect: l_2[[2]] or l_2[["second_characters"]]
```

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

3 Creating and using functions

> **Question:** I've reviewed what we've covered so far, and I feel that I could benefit from additional practice with functions. I'm still not entirely comfortable with them, and I believe they will be crucial as we progress.

3.1 Using functions

Functions are algorithms that apply a certain routine on an input, thereby producing (almost always) an output. The function sqrt(), for instance, takes as input a number and returns as output another number, namely the square root of the input:

```
sqrt(2.5)
```

```
[1] 1.581139
```

To call a function we first write its name, followed by parentheses. Inside the parentheses we can specify the arguments of the function. Most of the time, the first argument(s) are the input(s).

Often, you can give more arguments to the function. These specify how the algorithm of the function works. If you are not sure which arguments a function accepts, you can always use `help()`.

Let us consider the example of `mean()`. This function takes a vector of numbers as input and returns the mean of these numbers:

```
test_1 <- c(1, 4, 2, 4, 9, NA, 44)
mean(test_1)
```

```
[1] NA
```

This result might be surprising. Lets inspect `mean()` further:

```
help(mean)
```

We see that the first argument of `mean()` is called `x`. We can, but do not have to specify this name for the first argument. Neither we must for the further argumens, but this is highly recommended.

There are two more arguments for `mean()`: `trim` and `na.rm`. In our case, the second one is relevant: if `na.rm` is set to `TRUE`, the function removes all `NA` values from the input before computing the mean:

```
mean(test_1, na.rm = TRUE)
```

```
[1] 10.66667
```

3.2 Defining functions

Whenever you perform certain actions several times, it is often a good idea to define a function. This way, you can call the function instead of writing the code for the action every time anew.

Assume we want to define a function that takes as input a vector of numbers and returns the sum of the natural logarithms of these numbers.

We define a new function by using the function `function()`. - We start our definition by associating the new function with a name (here: `log_sum`) so that we can use it later. - The arguments to `function()` are then arguments that our new function should accept. In our case, we only have one argument. - After that comes the *function body*. It contains all the routines that the function should execute when called. The function body is always enclosed by curly brackets. - Finally, we use the `return()` function to specify what the function should return. This is not strictly necessary, but it is a good practice to do so.

```
log_sum <- function(input_vector){
  logs <- log(input_vector)
  sum_logs <- sum(logs)
  return(sum_logs)
}
```

We can then call the function by name:

```
log_sum(c(1, 2, 3, 4, 5))
```

```
[1] 4.787492
```

4 Using factors

> **Question:** If there's time, I would also appreciate a repetition of factors. While I generally understand the concept, the more complex ones are still a bit confusing to me.

On a basic level, you can think of a factor as an atomic vector that can only take a pre-specified number of values, so called *levels* . It is often used to encode ordinal or categorical data:

```
x <- factor(c("Female", "Male", "Female"))
x
```

```
[1] Female Male   Female
Levels: Female Male
```

By default, the levels are the unique elements of the input vector. But you can set them also explicitly, which is useful if some levels are missing in the input vector:

```
x <- factor(c("Female", "Male", "Female"),
            levels=c("Diverse","Female", "Male"))
x
```

```
[1] Female Male   Female
Levels: Diverse Female Male
```

Note that if you set `levels` explicitly, and the input vector contains an element that is not in the levels, it will be set to `NA`:

```
x <- factor(c("Female", "Male", "Female", "Unknown"),
            levels=c("Diverse","Female", "Male"))
x
```

```
[1] Female Male   Female <NA>
Levels: Diverse Female Male
```

You can use `table()` to get the frequencies for each level:

```
table(x)
```

```
x
Diverse  Female    Male
      0       2       1
```

If you want the levels to be ordered, you can use the optional argument `ordered = TRUE`:

```
x <- c("High", "High", "Low", "High")
x <- factor(x,
            levels = c("Low", "Mid", "High"),
            ordered = TRUE)
x
```

```
[1] High High Low  High
Levels: Low < Mid < High
```

Factors are not that easy to work with, especially since they are in fact named `integer`s, a fact that makes their behavior sometimes hard to predict.

```
typeof(x)
```

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

Therefore, I suggest to code data as factors only if this comes with concrete benefits.