Practice R

Tutorial 02: Base R

Edgar Treischl

5/7/23

Base R

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

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

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

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

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

I will introduce further `base` R features when we need them and I will not ask you to assign objects, create simple functions, or other probably boring `base` R examples in this tutorial.

Such tasks are important but abstract in the beginning. Instead, we focus on typical errors that may occur while we start to work with R. Why do we not practice the discussed content of `base` R, but concentrate on errors in this tutorial?

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

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

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

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.

Typical error messages

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

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

a + b
```

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

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

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

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

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

This error is hard to spot, but it illustrates that we need to be careful not to introduce mistakes. Moreover, RStudio gives parentheses that belong together the same color which help us to keep overview. Go to the RStudio menu (via the `<Code>` tab) and select *rainbow parentheses* if they are not displayed in color in the Code pane.

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

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

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

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

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

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

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

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

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

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

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"`?
```

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.

Cryptic errors

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

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

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

Unfortunately, some errors and warnings seem more like an enigma than useful feedback. Imagine, R tells you that a *non-numeric argument* has been applied to a *binary operator*. The next console reproduces the error with two example vectors. The last value of the vector `y` is a character (e.g., a missing value indicator: `NA`) and for obvious reasons we cannot multiply `x` with `y` as long as we do clean the latter.

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

result <- x * y
```

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

```
result
```

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

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

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

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

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

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

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

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

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

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

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)
```

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
```

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.

```
# 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:

```
# 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
```

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.

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 2022)
- 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")
```

References

Müller, Kirill, and Hadley Wickham. 2022. *tibble: Simple Data Frames*. https://CRAN.R-project.org/package=tibble.

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

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