Introduction to Programming with R

Dries Debeer & Benjamin Becker

31. September and 01. October 2021

Zurich University

Agenda

Day 1

- RStudio setup
- Basic elements & data types of the R language
- Flow & conditional programming
- Loops & iteration
- Writing & using functions (part I)

Day 2

- Writing & using functions (part II)
- Debugging
- Good programming practices

Open questions from day 1?

Functions II

Reasons

Why write functions?

- Readability
  - Shortens the code
  - Removes distractions, like references in a paper
  - Avoids repetition (DRY)
  - Easier understanding
- Transferability
  - Other use cases
  - Other projects
  - Other persons

Readability

Writing a function:

```
mean(mtcars$mpg)
```

```
[1] 20.09062
```

Not writing a function:

```
sum(mtcars$mpg)/dim(mtcars)[1]
```

```
[1] 20.09062
```

Readability

Writing a function:

```
summary(mtcars$mpg)


#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#   10.40   15.43   19.20   20.09   22.80   33.90
```

Readability

Not writing a function:

```
round(c("Min." = min(mtcars$mpg),
  "1st Qu." = as.numeric(quantile(mtcars$mpg)[2]),
  "Median" = median(mtcars$mpg),
  "Mean" = mean(mtcars$mpg),
  "3rd Qu." = as.numeric(quantile(mtcars$mpg)[4]),
  "Max." = max(mtcars$mpg)), 2)

#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#   10.40   15.43   19.20   20.09   22.80   33.90
```

Default arguments

What happens if the user omits an argument?

```
add_things_def <- function(x) {
  x + 10
}
add_things_def()

# Error in add_things_def():  argument "x" is missing,
with no default
```

Default arguments

Default arguments are made for such instances!

```
add_things_def <- function(x = 1) {
  x + 10
}
add_things_def()
```

```
[1] 11
```

Lazy Evaluation

Sometimes missing arguments are irrelevant!

```
add_things3 <- function(x, y) {
  x + 10
}
add_things3(2)
```

```
[1] 12
```

Output

Functions usually return a single object.

- (Standard) The last evaluated object
- Object defined by return()
- An error via stop()
- Additional: Warnings + Messages
- Additional: Console output
- Additional: Output to graphic device

Output

return()

```
add_things_return <- function(x = 1) {
  x2 <- x*2
  return(x)
  out <- x + x2
  out
}
add_things_return(2)
```

```
[1] 2
```

Output

Error: stop()

```
add_things_stop <- function(x = 1) {
  x2 <- x*2
  stop("My own error message")
  out <- x + x2
  out
}
add_things_stop(2)

# Error in add_things_stop(2):  My own error message
```

Output

Abbreviation for if(!test) and stop():

```
mean2 <- function(x, na.rm = FALSE) {
  stopifnot(is.numeric(x))
  sum(x)/length(x)
}
mean2("a")

# Error in mean2("a"):  is.numeric(x) is not TRUE
```

Output

Warnings: warning()

```
add_things_warning <- function(x = 1) {
  x2 <- x*2
  warning("My own warning message")
  out <- x + x2
  out
}
add_things_warning(2)

# Warning in add_things_warning(2):  My own warning
message
```

```
[1] 6
```

Output

Messages: message()

```
add_things_message <- function(x = 1) {
  x2 <- x*2
  message("My own message")
  out <- x + x2
  out
}
add_things_message(2)

# My own message
```

```
[1] 6
```

Output

Console output: cat() and print()

```
add_things_console <- function(x = 1) {
  x2 <- x*2
  print("My own console output")
  cat("The output is not", x2)
  out <- x + x2
  out
}
add_things_console(2)
```

[1] "My own console output" The output is not 4[1] 6

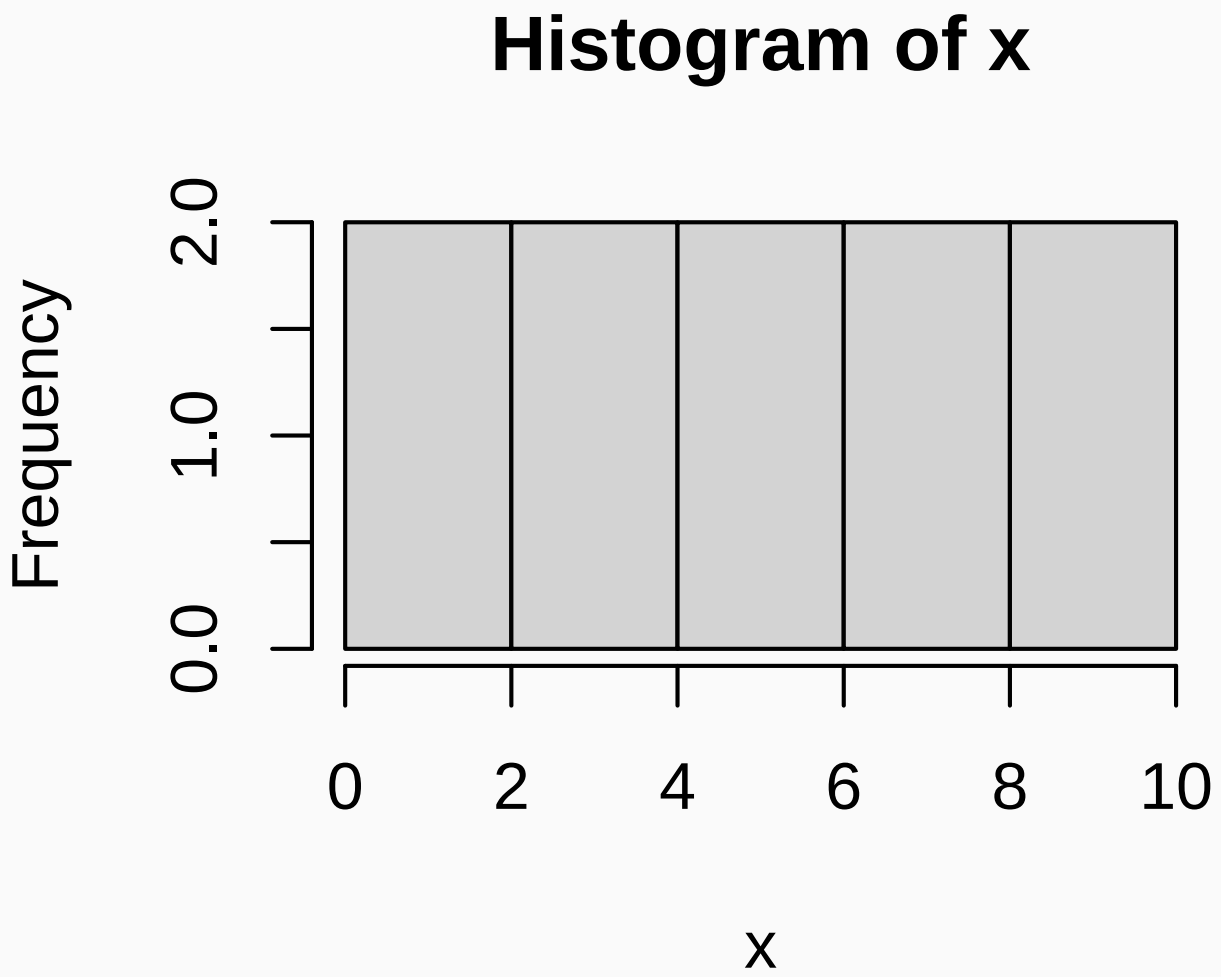
Output

Graphics output: Standard plots, ggplot2

```
createHist <- function(x) {
  hist(x)
  NULL
}
createHist(1:10)
```

Output

Graphics output: Standard plots, ggplot2



Output

Programming advice

- Avoid output to the console
- Choose carefully if something warrants a message, warning or error
- Plotting functions should return nothing or the plot as an object

Exercises

Functions III

Writing Functions

Before creating the function

- What should my function do?
- Input (Arguments)
- Output

After creating the function

- Test it
- Add input validation
- Document it

What makes a good function?

Pure functions!

- no side effects
- the only output is returned
- no dependency on global environment
- only input via arguments

Results in easier understanding and higher portability.

...

How can functions receive flexible numbers of inputs?

Examples:

- sum()
- save()
- ...

via dot dot dot (...)

```
add_all_things2 <- function(...) {
  l <- list(...)
  do.call(sum, l)
}
add_all_things2(2, 3, 5, 10)
```

```
[1] 20
```

on.exit()

Performing an action when the function terminates

```
add_things <- function(x, y) {
  on.exit(cat("Sum of", x, "and", y))
  x <- x + 20
  x + y
}
out <- add_things(1, 2)
```

Sum of 21 and 2

```
out
```

[1] 23

Accessing the function call

Accessing the function call

```
showArgs <- function(x, y) {
  match.call()
}
showArgs(1, 2)
```

showArgs(x = 1, y = 2)

Exercises

Debugging

Debugging

- browser()
- traceback()
- options(error = recover)
- options(warn = 2)

Also:

- trace() & untrace()
- debug() & undebug(), debugonce()

browser()

Inspecting a function interactively

```
some_function <- function(x, y) {
  z <- x + y
  browser()
  z
}
some_function(x = 1, y = 5)
```

browser()

```
> some_function <- function(x, y) {
+   z <- x + y
+   browser()
+   z
+ }
> some_function(x = 1, y = 5)
Called from: some_function(x = 1, y = 5)
Browse[1]>
```

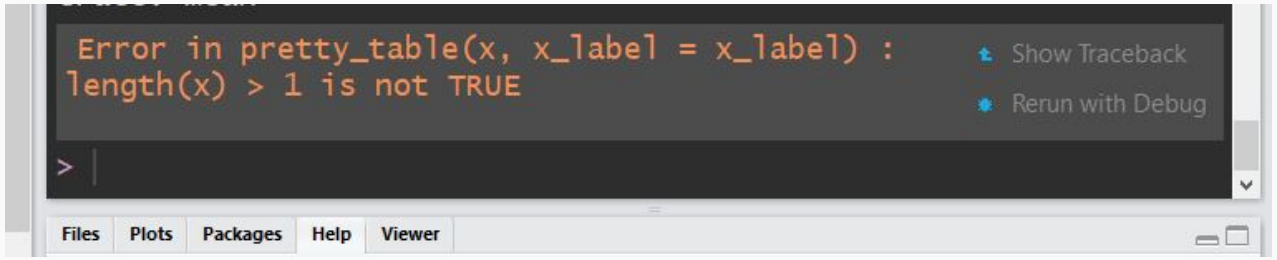
browser()

Navigating within a browser:

ls()	Show existing objects in the current environment
c	Exit the browser and continue execution
Q	Exit the browser, return to top level
where	Show call stack

traceback()

Understanding the call stack:



traceback()

Understanding the call stack:

```
Error in pretty_table(x, x_label = x_label) :
length(x) > 1 is not TRUE
                                        Hide Traceback
                                        Rerun with Debug
13. stop(simpleError(msg, call = if (p <- sys.parent(1L)) sys.c
    all(p)))
12. stopifnot(length(x) > 1)
11. pretty_table(x, x_label = x_label)
10. pretty_statistics(sub_dat$cyl, x_label = "Cyl")
 9. FUN(data[x, , drop = FALSE], ...)
 8. FUN(X[[i]], ...)
 7. lapply(X = ans[index], FUN = FUN, ...)
 6. tapply(seq_len(32L), list(`mtcars$carb` = c(4, 4, 1, 1, 2,
    1,
    4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2, 2, 4, 2, 1,
     2,
    2, 4, 6, 8, 2)), function (x)
    FUN(data[x, , drop = FALSE], ...), simplify = TRUE)
 5. eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = s
    implify)),
        data)
 4. eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = s
    implify)),
        data)
 3. structure(eval(substitute(tapply(seq_len(nd), IND, FUNx, si
    mplify = simplify)),
        data), call = match.call(), class = "by")
 2. by.data.frame(mtcars, mtcars$carb, function(sub_dat) {
        pretty_statistics(sub_dat$cyl, x_label = "Cyl")
    })
 1. by(mtcars, mtcars$carb, function(sub_dat) {
        pretty_statistics(sub_dat$cyl, x_label = "Cyl")
    })
>
```

Recover

Being able to chose an environment from the call stack:

```
# on
options(error = recover)

# off
options(error = NULL)
```

Recover

Being able to chosse an enrivonment from a call stack:

```
Error in pretty_table(x, x_label = x_label) : length(x) > 1 is not TRUE

Enter a frame number, or 0 to exit

 1: by(mtcars, mtcars$carb, function(sub_dat) {
    pretty_statistics(sub_dat$cyl, x_label = "Cyl
 2: by.data.frame(mtcars, mtcars$carb, function(sub_dat) {
    pretty_statistics(sub_dat$cyl, x_l
 3: structure(eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = simplify)), data), call =
 4: eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = simplify)), data)
 5: eval(substitute(tapply(seq_len(nd), IND, FUNx, simplify = simplify)), data)
 6: tapply(seq_len(32), list(`mtcars$carb` = c(4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4,
 7: lapply(X = ans[index], FUN = FUN, ...)
 8: FUN(X[[i]], ...)
 9: FUN(data[x, , drop = FALSE], ...)
10: #2: pretty_statistics(sub_dat$cyl, x_label = "Cyl")
11: #3: pretty_table(x, x_label = x_label)
12: #2: stopifnot(length(x) > 1)

Selection:
```

Warnings

Turning warnings into errors

```
# on
options(warn = 2)

# off
options(warn = 1)
```

Exercises

Good programming practices

“Write code for humans, not for machines!”

Code Style

Invest time in writing readable R-code.

- It will make collaborations easier
- It will make debugging easier
- It will help make your analyses reproducible

There is a complete *tidyverse* style-guide `https://style.tidyverse.org/`.

Go easy on your eyes

- with spaces before and after: `- + / * = <- < == >`
- always use <- for assignments
- only use = in function calls
- use indentation (largely automatic in RStudio)
- `CamelCaseNames` vs `snake_case_names`
- be consistent!
- wrap long lines at column 70-80 (Rstudio)

White spaces

```
new_var=(var1*var2/2)-5/(var3+var4)

# versus

new_var <- (var1 * var2 / 2) - 5 / (var3 + var4)
```

Indentation

```
for(name in names){formula=as.formula(paste0("y~.-",name))
fit<-lm(formula,data=my_data)
coefs[["name"]]=coef(fit)
print(name)
print(summary(fit))}

# versus

for(name in names){
  formula <- as.formula(paste0("y~.-", name))
  fit <- lm(formula, data = my_data)
  coefs[["name"]] <- coef(fit)
  print(name)
  print(summary(fit))
}
```

Wrap long lines

```
final_results <- data.frame(first_variable =
sqrt(results$mean_squared_error), second_variable =
paste0(results$condition, results$class, sep = ":"),
third_variable = results$bias)

# versus

final_results <- data.frame(
  first_variable = sqrt(results$mean_squared_error),
  second_variable = paste0(results$condition,
                           results$class, sep = ":"),
  third_variable = results$bias)
```

Go easy on your mind

- use meaningful names: “self-explainable”
- always write the formal arguments in function calls (except the first)
- benefit from autocompletion (<tab>) => embrace longer names
- use `TRUE` and `FALSE` not `T` and `F`
- comment, comment, comment
  - not what (should be clear from the code)
  - but why
  - explain the reasoning, not the code

Use meaningful names

```
V <- myFun(m1_B)

# versus

RMSE_age_gender <- get_RMSE(lm_age_gender)
```

Programming advice

Use verbs for functions and nouns for objects.

Write formal arguments

Benefit from auto completion using tab

```
m1_B <- lm(outcome ~ age*gender,
           exp1, condition_1, freq)

# versus

lm_age_gender <- lm(outcome ~ age*gender,
                    data = exp1,
                    subset = condition_1,
                    weigths = freq)
```

Comment, comment, comment

```
## Start every Rscript with a comment that explains
##  what the code in the script does, why it does
##  this, and to which project it belongs.
##  Your future self will be very thankful!
##
## Mention which packages you are using in this Rscript.

## Use sections to separate chunks ----------------------

## Maybe even subsections ==============================

## Recode variables so that missings are coded as "NA"
dat[dat %in% c(99, 999)] <- NA  # missings coded 99 or 999
```

Keep your code slim

Try to limit your *package-dependencies*.
Only load `library()` the packages that you absolutely need. If you are only using `dplyr`, it does not make sense to load the complete `tidyverse`.
Controversial: when possible, use the `::` operator (and consider not loading the package). `<package>::<function>`

- explicit dependencies
- less name conflicts

Never Attach

Forget about `attach()`!

Don't use it, unless you completely understand what happens (see `?attach`).

Use `with(data.frame, expression)` instead.

```
# using with()
n <- 2e+4
data <- data.frame(x = runif(n),
                   y = runif(n),
                   z = seq_len(n))
result <- with(data, exp(x) / log(z) + 5 * sqrt(y))
```

Testing R code

Writing code is error prone. Incorporate tests and checks in your workflow. For instance, when you do data manipulations like a complex restructuring of the data, or a complex recoding of multiple variable, write some code that allows you the check whether the obtained results are what you want them to be.

- minimal examples
- write tests and checks
- helpful packages: `testthat`, `RUnit`, `testit`, ...

Working with RStudio

"Every project should get an RStudio Project!"

Don't use `setwd("pathtomylocal_folder")`

Issues when:

- folders names are changed
- folders are moved
- a shared drive is used
- you ZIP and send the folder

Working with RStudio

Don't save work space to `.RData`.

- Tools < Global Options < Workspace < Save workspace ....
- Save the code instead!
- `saveRDS()` and `readRDS()` for objects that require long computations

Working with RStudio

Don't use `rm(list = ls())` at the start of an Rscript.

- Start clean, every time.
- Keep it clean. No outside code, no outside computing.
- Regularly completely clean the work space (or restart the session).

```
.rs.restartR()
```

Working with RStudio

Keep it clean

- one folder per project!
- work on different projects in different RStudio instances!
- each with own R console, working directory, ...

Working with RStudio

Organize your project folder

- R-folder with R scripts
- Data-folder with data
- split long scripts in meaningful chunks
- use relative paths (alternative: here-package)

```
# read data
this_data <- read.csv("Data\the-correct-file.csv")

# source Rscript
source("R\0_first-script-to-source.R")
```

Working with RStudio

Use keyboard shortcuts

- Can make working in RStudio more efficient
- Completely tunable: Tools < Modify Keyboard Shortcuts...
- Useful shortcuts (defaults):
  - jump to editor: `ctrl + 1`
  - jump to console: `ctrl + 2`
  - jump to ...: `ctrl + 3-9`
  - jump to next tab: `ctrl + tab`
  - jump to previous tab: `ctrl + shift + tab`

Working with RStudio

More useful shortcuts (defaults):

- run selection/selected line: `ctrl + enter`
- save current file: `ctrl + s`
- close current file: `ctrl + w`
- restart R: `ctrl + shift + F10`
- Show help (for function at cursor) `F1`
- Show source code (for function at cursor) `F2`

More on this HERE.

Exercises

Wrap Up

General Advice

- Investing time in learning R pays off
- It's a steady learning curve
- Learn from masters
- Rewrite important code - the first attempt is usually not the best approach

General R Advice

- Document well
- Use a consistent style
- Write functions
- Split long functions in smaller ones
- Write wrappers
- Use Iteration (don't copy paste)
- Use matrix operations and vectorized functions instead of loops
- Use git

Literature Recommendations

R Resources

- Avanced R Ed. 1 (http://adv-r.had.co.nz/)
- Avanced R Ed. 2 (https://adv-r.hadley.nz/)
- R Inferno (https://www.burns-stat.com/pages/Tutor/R_inferno.pdf)
- R Packages (https://r-pkgs.org/)
- Clean Code (https://enos.itcollege.ee/~jpoial/oop/naited/Clean%20Code.pdf)

Thank you for your attention!

Thank you for your attention!

Questions? Remarks?