Introduction to Programming with R

Dries Debeer & Benjamin Becker

31. September and 01. October 2021

Zurich R Courses

Table of Content

Agenda

Functions II

- Why write functions?
- Single return object
- Side effects
- Error, warning, & message
- Default Arguments
- Lazy Evaluation

Functions III

- Scoping rules
- Functional Programming
- dot dot dot
- on.exit()
- Functions are objects
- Work flow
- Good functions?

Debugging

- browser
- traceback
- recover
- warnings

Good programming practices

- Code Style
- Go easy on your eyes
- Go easy on your mind
- R Peculiarities
- Working with RStudio

Wrap Up

Agenda

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?

- They make code ...
  - shorter (less repetition)
  - easier to read and understand
- They help avoid copy-paste errors
- They make it easier to change your code
- They increase transferability to ...
  - other use cases
  - other projects
  - other persons
- They keep your work space clean

Readability

Writing a function:

```
RMSE <- get_RMSE(predictions, observations)
```

Not writing a function:

```
diff <- observations - predictions
sq_diff <- diff^2
m_sq_diff <- mean(dif)
RMSE <- sqrt(m_sq_diff)
```

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
```

Single return object

Pure functions return a single object.

- (Standard) The last evaluated object
- Object defined by return()

Figure 1: A pure function.

Single return object

`return()` stops the computation, and returns the object.

```
return_early <- function(x = 1) {
  x2 <- x*2
  return(x2)
  out <- x + x2     # not executed
  out
}
return_early(2)

> [1] 4
```

Single return object

Multiple return objects can be combined in a list!

```
                                  # Name
do_this <- function(vector, other_vector) { # Arguments
  # many computations                       # Body
  return(list(output1 = this,
              output2 = that))              # Output
}
```

Single Return Object

The return object is a list with multiple objects.

```
get_info <- function(x){
  mean_x <- mean(x)
  median_x <- median(x)
  n_obs_x <- length(x)
  range_x <- range(x)
  return(list(mean = mean_x, median = median_x,
              n_obs = n_obs_x, range = range_x))
}
str(get_info(airquality$Wind))
```

```
> List of 4
>  $ mean  : num 9.96
>  $ median: num 9.7
>  $ n_obs : int 153
>  $ range : num [1:2] 1.7 20.7
```

Side Effects

Functions can have **“side effects”**:

- console output
- plots
- write/save on drive
- ...

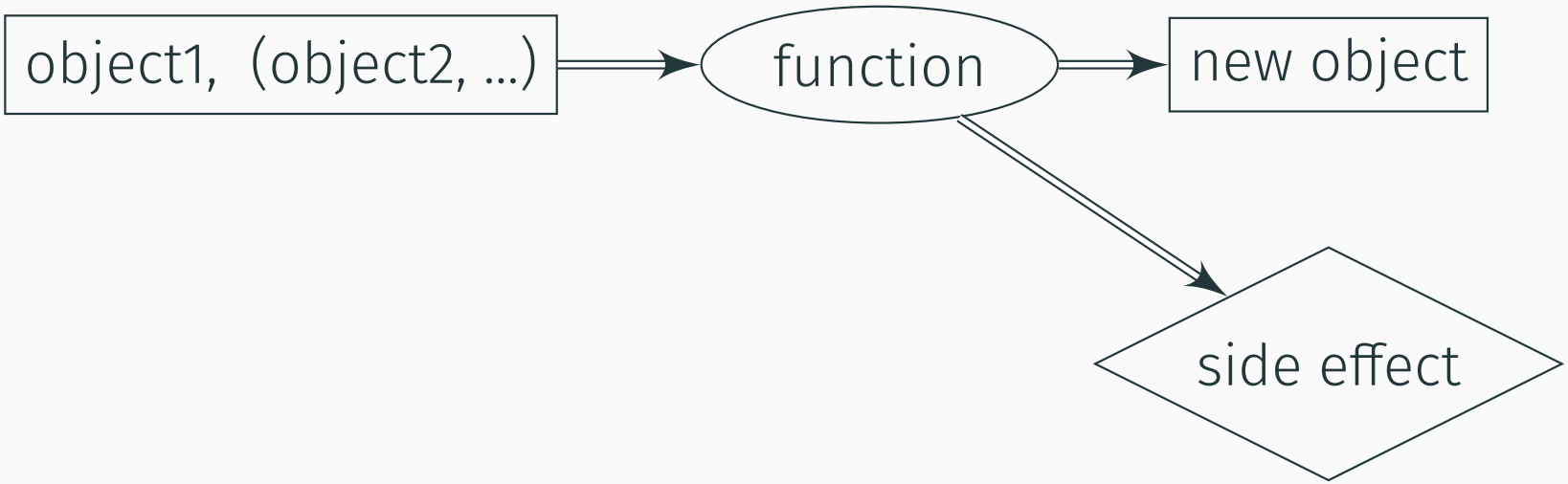


Figure 2: A function with side effect.

Side Effects

Console output: `?cat` and `?print`

```
print_info <- function(x){
  info <- get_info(x)
  cat("There are ", info$n_obs,
      " observed values. \nThe mean is ",
      round(info$mean, 2), ". \nThe median is ",
      round(info$median, 2), ". \n", sep = "")
}
print_info(airquality$Wind)
```

```
> There are 153 observed values.
> The mean is 9.96.
> The median is 9.7.
```

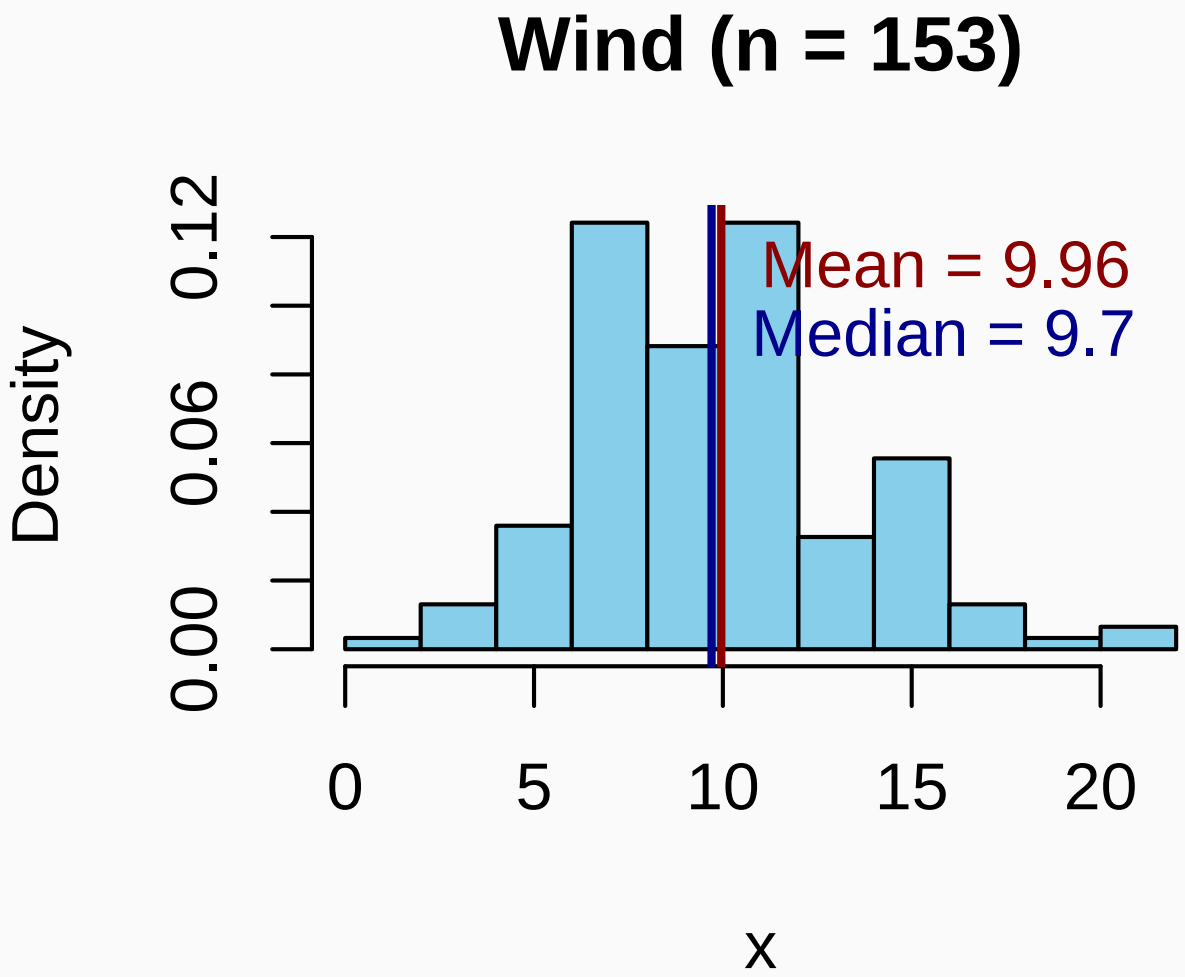
Side effects

Graphics output: Standard plot, ggplot2, lattice

```
hist2 <- function(x, title){
  info <- get_info(x)
  mean_median <- as.numeric(info[c("mean", "median")])
  hist(x, col = "skyblue", freq = FALSE,
       main = paste0(title, " (n = ", info$n_obs, ")"))
  abline(v = mean_median, lwd = 2,
         col = c("darkred", "darkblue"))
  text(mean_median, y = c(.11, .09),
       labels = paste(c("Mean", "Median"),
                      round(mean_median, 2),
                      sep = " = "),
       col = c("darkred", "darkblue"), pos = 4)
}
hist2(airquality$Wind, "Wind")
```

Side effects

Graphics output



Output

Programming advice

- Write pure functions (no-side effects)
- Write separate functions for side effects
- Plotting functions should return `NULL` or the plot as an object

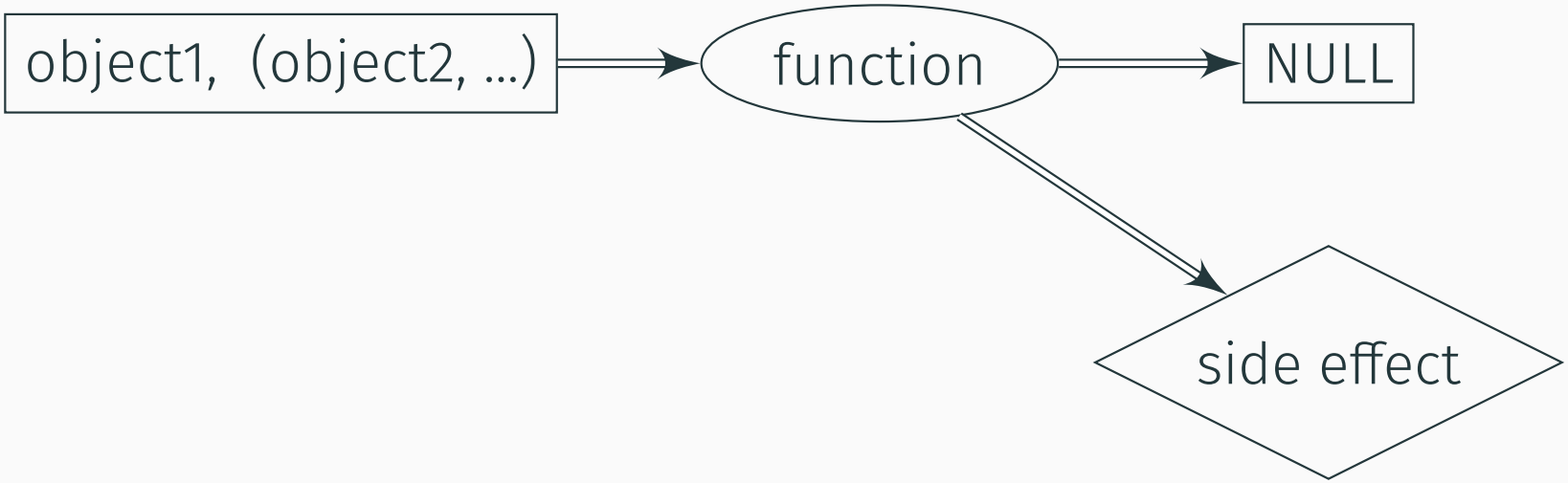


Figure 3: A side effect function.

Error, warning, & message

Error: computation is interrupted without return object!

`?stop`

```
get_log_xtox <- function(x) {
  if(!is.numeric(x)) stop("This does not work!")
  x_x <- x^x
  return(log(x_x))
}
get_log_xtox("a")

> Error in get_log_xtox("a"): This does not work!
```

Error, warning, & message

Error: computation is interrupted without return object!

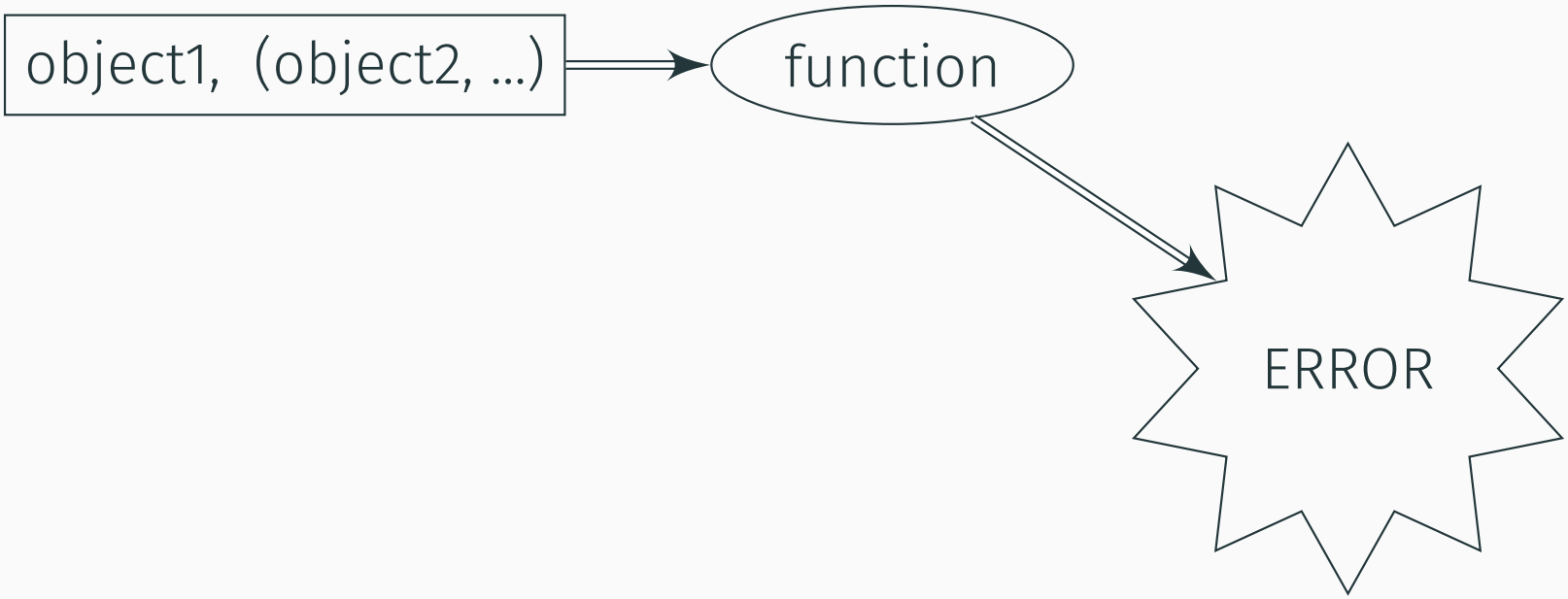


Figure 4: Computation with Error.

Error, warning, & message

`?stopifnot` is an abbreviation for `if(!test) stop()`:

```
get_log_xtox <- function(x) {
  stopifnot(is.numeric(x))
  x_x <- x^x
  return(log(x_x))
}
get_log_xtox("a")

> Error in get_log_xtox("a"): is.numeric(x) is not TRUE
```

Error, warning, & message

Message: To inform the user about something.

```
?message
```

```
get_log_xtox <- function(x) {
  x_x <- x^x
  message("Thank you for using this function!")
  return(log(x_x))
}
get_log_xtox(2)
```

```
> Thank you for using this function!

> [1] 1.386294
```

Error, warning, & message

Warning: Warn the user that something may be fishy.

```
?warning
```

```
get_log_xtox <- function(x) {
  if(x < 0 && (x %% 2 == 0))
    warning("Not sure you can trust the result.",
            call. = FALSE)
  x_x <- x^x
  return(log(x_x))
}
get_log_xtox(-2)
```

```
> Warning: Not sure you can trust the result.

> [1] -1.386294
```

Error, warning, & message

Message & warning: computation is NOT interrupted!

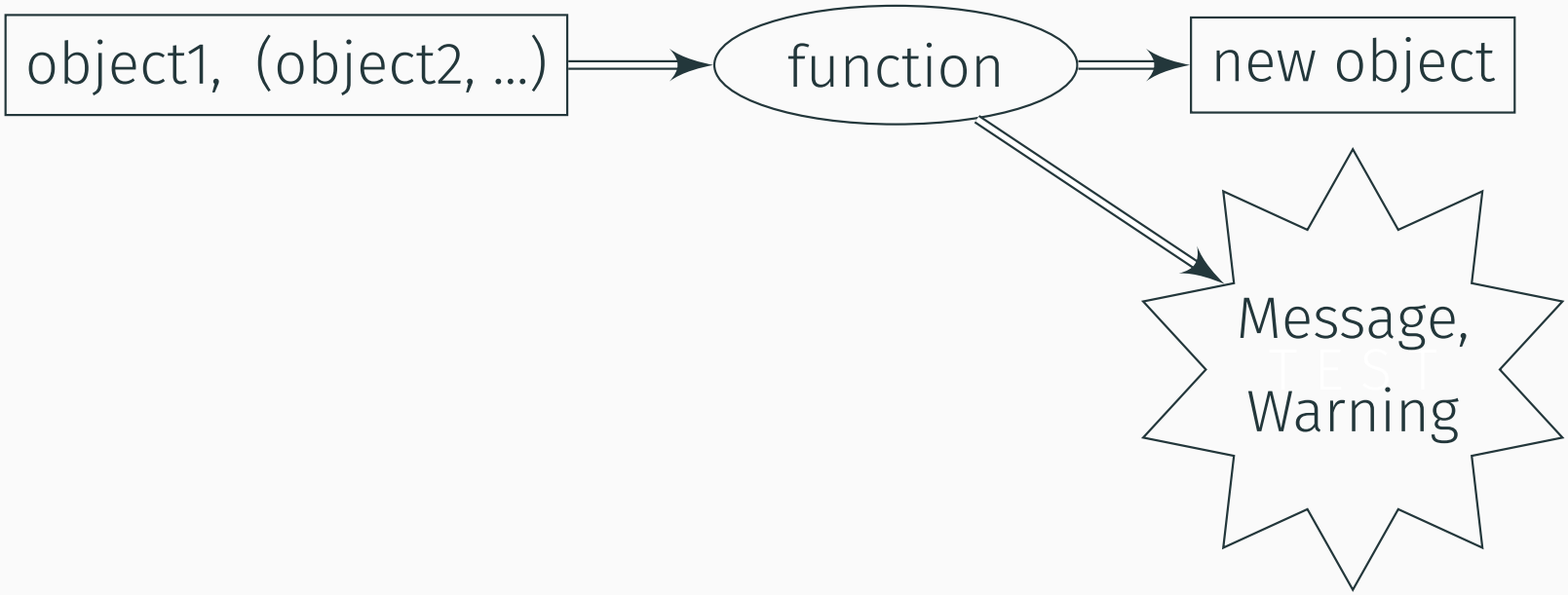


Figure 5: A message or warning.

Output

Programming advice

- Choose carefully when something warrants a message, warning or error
- Write clear and helpful warnings, errors, messages

Default arguments

What happens if the user omits an argument?

```
add_ten <- function(x) {
  return(x + 10)
}
add_ten()

> Error in add_ten(): argument "x" is missing, with no
default
```

Default arguments

Default arguments are made for such instances!

```
add_ten_default <- function(x = 0) {
  return(x + 10)
}
add_ten_default()

> [1] 10
```

Default arguments

Additional arguments give (the user) flexibility. Default arguments keep the function easy to use.

Try `?lm`

Programming advice

- Think which arguments to include, and which should (not) have defaults
- Choose sensible defaults

Lazy Evaluation

R only considers (evaluates) an argument when it is used.

```
add_ten_lazy <- function(x, y) {
  return(x + 10)
}
add_ten_lazy(2, y = stop("This is not evaluated"))

> [1] 12
```

Lazy Evaluation

R only considers (evaluates) an argument when it is used. But, you can `force` the evaluation:

```
add_ten_force <- function(x, y) {
  force(y)
  return(x + 10)
}
add_ten_force(2, y = stop("Evaluation was forced"))

> Error in force(y): Evaluation was forced
```

?force

Exercises

Functions III

Where does a function find objects?

R uses specific rules to find objects, which lead to the following:

```
a <- 55
add_a <- function(x){
  return(x + a)
}
add_a(5)

> [1] 60
```

When a function is called, the computations in the body are run line by line. When R cannot find an object inside the function, it looks outside the function.

Where does a function find objects?

Name masking!

Objects inside the function mask objects outside the function with the same name.

```
a <- 55
add_a <- function(x){
  a <- 5
  return(x + a)
}
add_a(5)

> [1] 10
```

Where does a function find objects?

R uses specific rules to find objects.

```
a <- b <- c <- d <- "fourth"
find_object <- function(a, b = "third", c = "third"){
  a <- "first"
  return(c(a = a, b = b, c = c, d = d))
}
find_object(b = "second")
```

```
>         a         b         c         d
>   "first" "second"  "third" "fourth"
```

Where does a function find objects?

R uses specific rules to find objects.

1. in the function body
2. in the function call
3. in the function definition
4. outside the function

Watch out with number 4! Frequently restart R: `Ctrl + shift + F4`

Functional programming

The return object should only depend on the arguments of the function, ***not*** on the context!

BAD:

```
a <- 55
add_a <- function(x){
  return(x + a)
}
add_a(5)

> [1] 60
```

Functional programming

The return object should only depend on the arguments of the function, ***not*** on the context!

GOOD:

```
add_a <- function(x, a = 55){
  return(x + a)
}
add_a(5)

> [1] 60
```

Functional programming

The function should not change the context.

BAD

```
a <- 55
change_a <- function(new_a){
  a <<- new_a
  return(invisible(NULL))
}
change_a(5)
a

> [1] 5
```

`...` dot-dot-dot

R has a special argument (in the definition of the function):

`...` (dot-dot-dot)

Examples:

- `?sum`
- `?save`
- ...

`...` dot-dot-dot

`...` can take *any* number of additional arguments

Useful for passing arguments to other functions like:

- `?apply`
- `?plot`
- ...

Useful when you don't know how many arguments there will be.

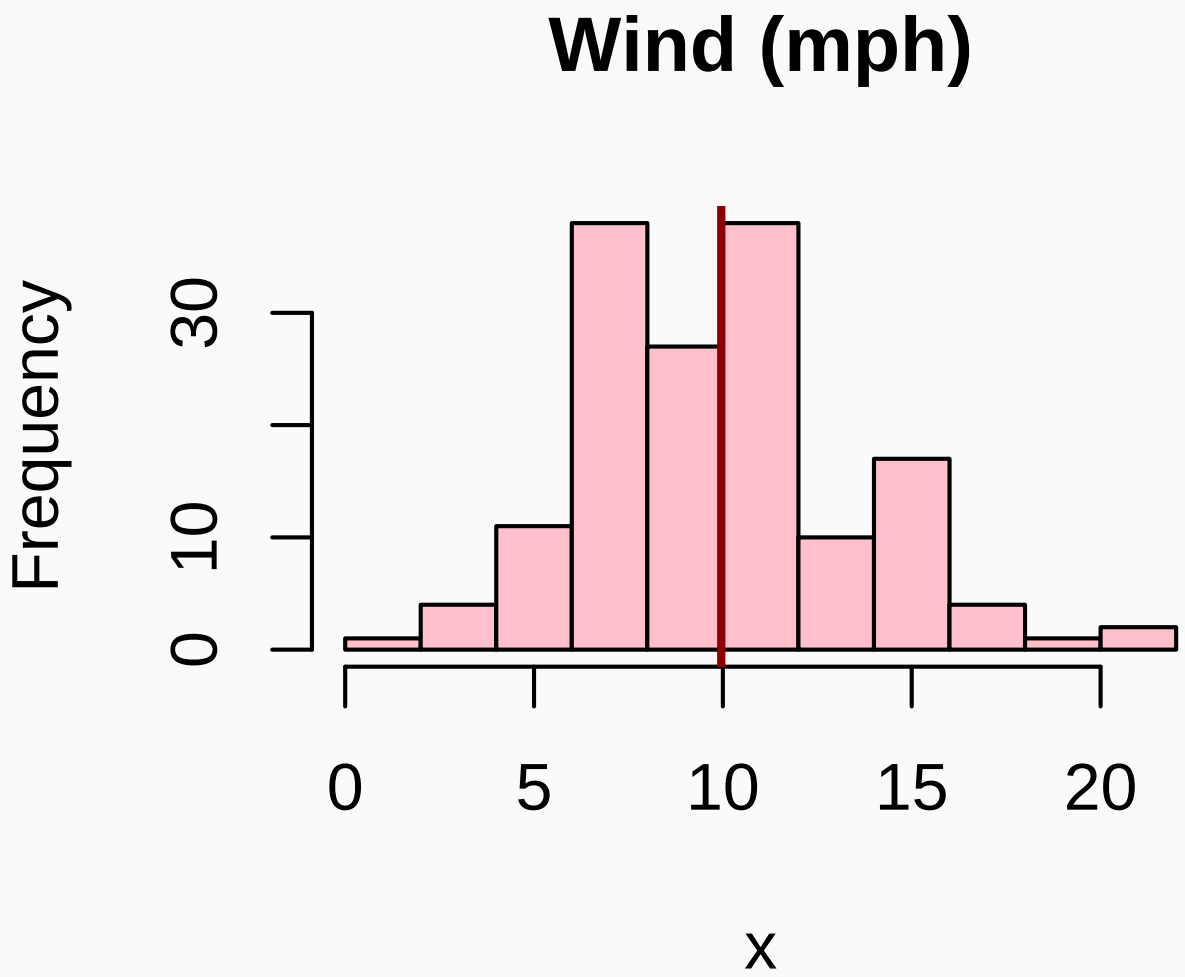
... dot-dot-dot

`plot` example

```
hist3 <- function(x, ...){
  hist(x, ...)
  abline(v = mean(x, ...),
         col = "darkred",
         lwd = 2)
}
hist3(airquality$Wind, col = "pink",
      main = "Wind (mph)")
```

... dot-dot-dot

`plot` example



... dot-dot-dot

`apply` example.

```
get_quantiles <- function(x, ...){
  out <- lapply(x, quantile, ...)
  return(do.call(rbind, out))
}
get_quantiles(airquality, na.rm = TRUE,
              probs = c(.25, .5, .27))
```

```
>              25%   50%    27%
> Ozone     18.00  31.5  18.05
> Solar.R 115.75 205.0 127.00
> Wind      7.40   9.7   7.40
> Temp     72.00  79.0  73.00
> Month     6.00   7.0   6.00
> Day       8.00  16.0   9.00
```

... dot-dot-dot

WARNING! Watch out with spelling mistakes, arguments can get lost!

```
get_quantiles <- function(x, ...){
  out <- lapply(x, quantile, ...)
  return(do.call(rbind, out))
}
get_quantiles(airquality, na.rm = TRUE,
              prosb = c(.25, .5, .27))
```

```
>            0%    25%   50%    75%  100%
> Ozone     1.0  18.00  31.5  63.25 168.0
> Solar.R   7.0 115.75 205.0 258.75 334.0
> Wind      1.7   7.40   9.7  11.50  20.7
> Temp     56.0  72.00  79.0  85.00  97.0
> Month     5.0   6.00   7.0   8.00   9.0
> Day       1.0   8.00  16.0  23.00  31.0
```

on.exit()

Performing an action when the function terminates.

```
add_ten_on_exit <- function(x) {
  on.exit(cat("Finished 'add_ten_on_exit', with input '",
              x, "'. \n", sep = ""))
  return(x + 10)
}
add_ten_on_exit(1)

> Finished 'add_ten_on_exit', with input '1'.
> [1] 11
```

on.exit()

Performing an action when the function terminates.

```
add_ten_on_exit <- function(x) {
  on.exit(cat("Finished 'add_ten_on_exit', with input '",
              x, "'. \n", sep = ""))
  return(x + 10)
}
add_ten_on_exit("one")

> Error in x + 10: non-numeric argument to binary
operator

> Finished 'add_ten_on_exit', with input 'one'.
```

Error, warning, & message

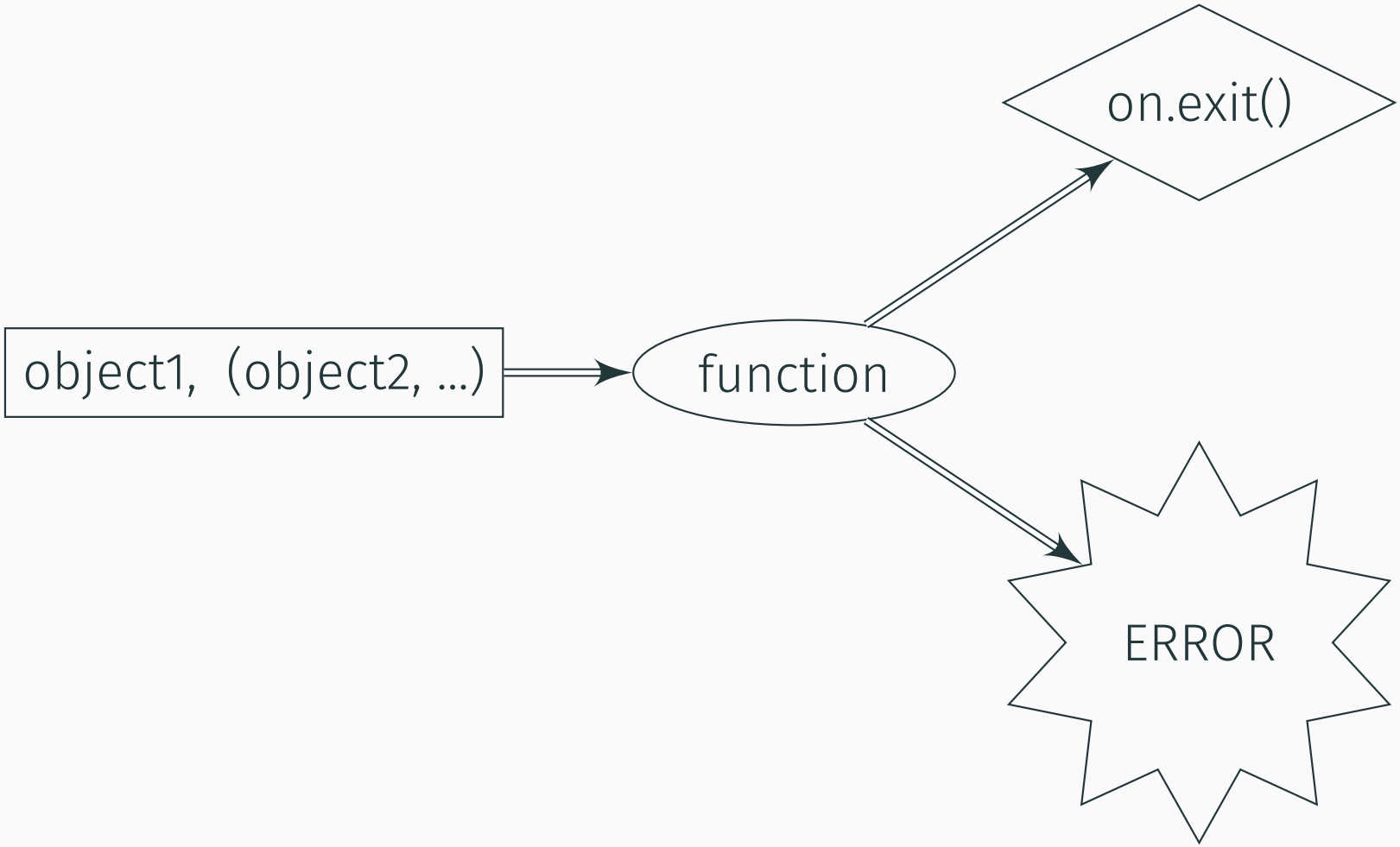


Figure 6: on.exit() with error.

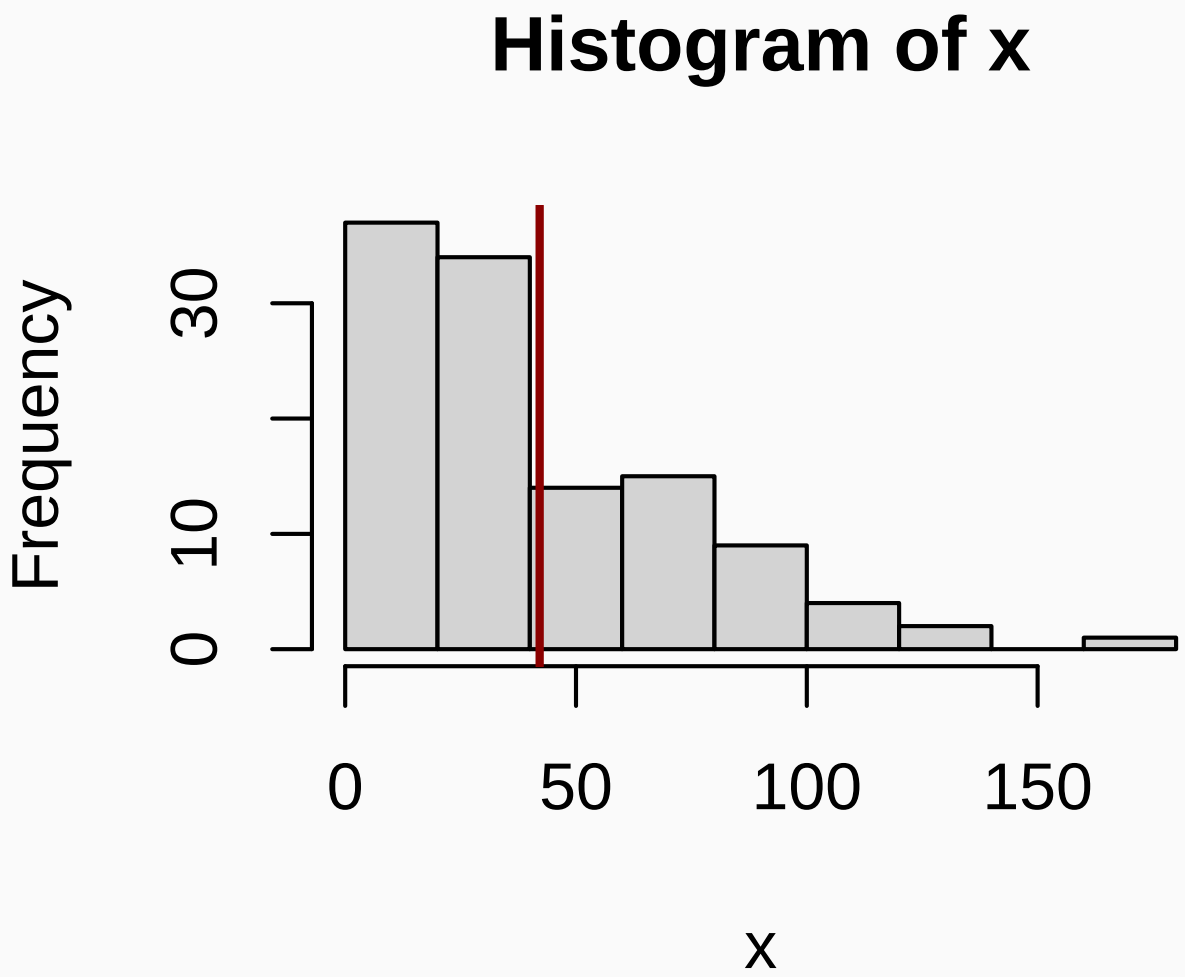
on.exit()

Useful when your function has side effects:

```
hist3 <- function(x, ...){
  old_options <- getOption("warn")
  on.exit(options(warn = old_options))
  options(warn = -1)
  hist(x, ...)
  abline(v = mean(x, ...),
         col = "darkred", lwd = 2)
}
hist3(airquality$Ozon, na.rm = TRUE)
```

on.exit()

Useful when your function has side effects:



"To understand computations in R, two slogans are helpful: Everything that exists is an object. Everything that happens is a function call."

— John Chambers

Functions are objects

Functions are also objects. They can be arguments.

- `apply`-family
- ...

```
do_this_that <- function(function1, function2, x){
  function2(function1(x))
}
do_this_that(sum, log, 0:3)

> [1] 1.791759
```

Functions are objects

Anonymous functions = functions without a name

```
do_this_that(function(x) x^2,
             function2 = function(y) sum(y) / (length(y - 1)),
             -1:5)


> [1] 8
```

Functions are objects

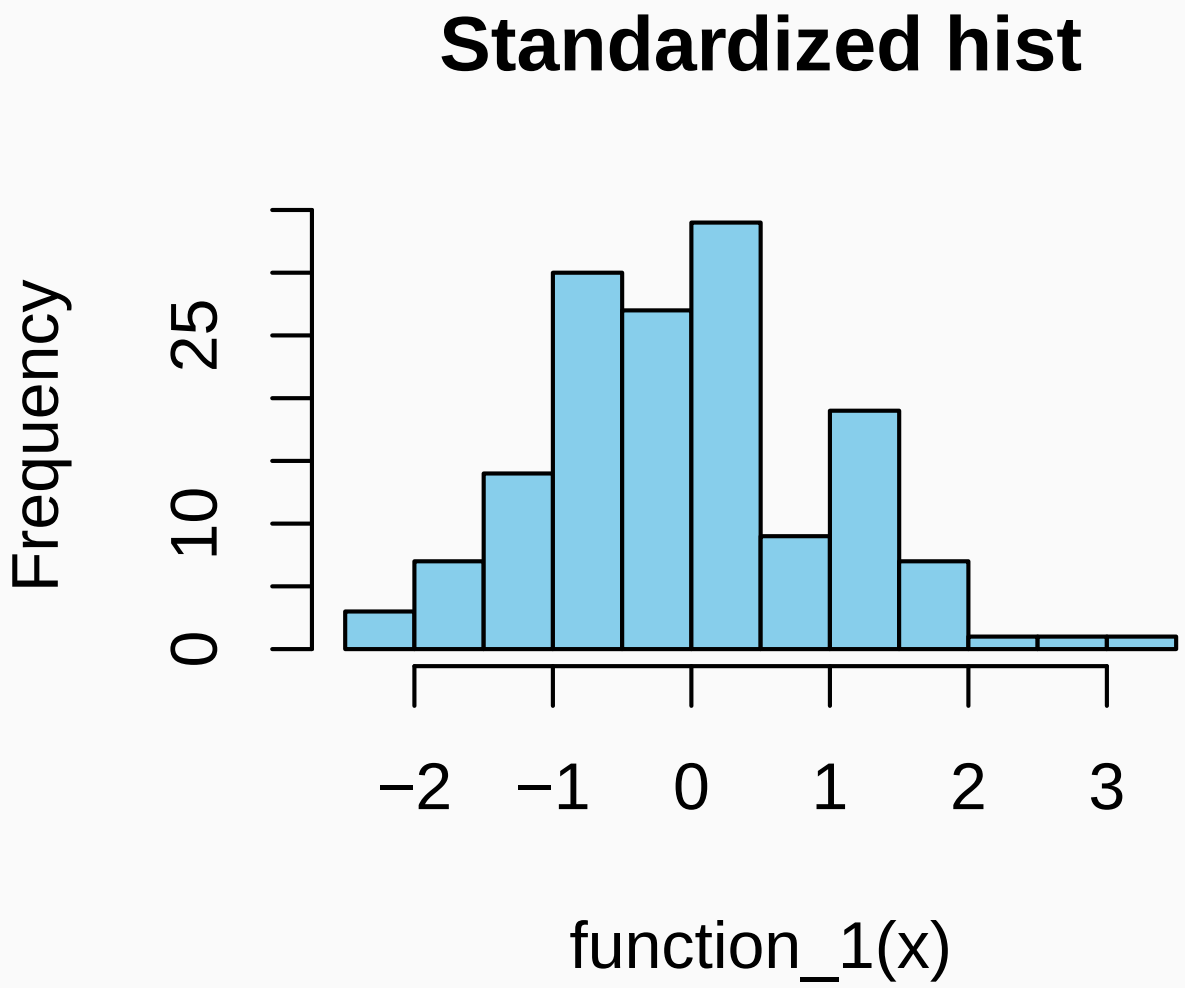
The return objects can also be functions:

```
combine_2fun <- function(function_1, function_2){
  out_function <- function(x, ...) {
    function_2(function_1(x), ...)
  }
  return(out_function)
}

standardized_hist <- combine_2fun(scale, hist)
standardized_hist(airquality$Wind,
                  col = "skyblue",
                  main = "Standardized hist")
```

Functions are objects

The return objects can also be functions:



Functions are objects

The return objects can also be functions:

```
combine_2fun <- function(function_1, function_2){
  out_function <- function(x, ...) {
    function_2(function_1(x), ...)
  }
  return(out_function)
}
mean_abs_deviation <- combine_2fun(abs, mean)
mean_abs_deviation(airquality$Ozone, na.rm = TRUE)

> [1] 42.12931
```

Functions are objects

The return objects can also be functions:

```
normalize <- combine_2fun(
  function(x) {x - min(x, na.rm = TRUE)},
  function(x) {x / max(x, na.rm = TRUE)})
normalize(airquality$Ozone)[1:4]

> [1] 0.23952096 0.20958084 0.06586826 0.10179641
```

Writing Functions

Before creating the function

- What should my function do?
- Which input objects (Arguments)?
- which additional options (Arguments)?
- What should the output object be?

After creating the function

- Test it
- Add input validation
- Document

What makes a good function?

Pure functions!

- no side effects
- no dependency on global environment
- only input via arguments (functional programming)

Results in easier understanding and higher portability.

Exercises

Debugging

Debugging

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

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]>
```

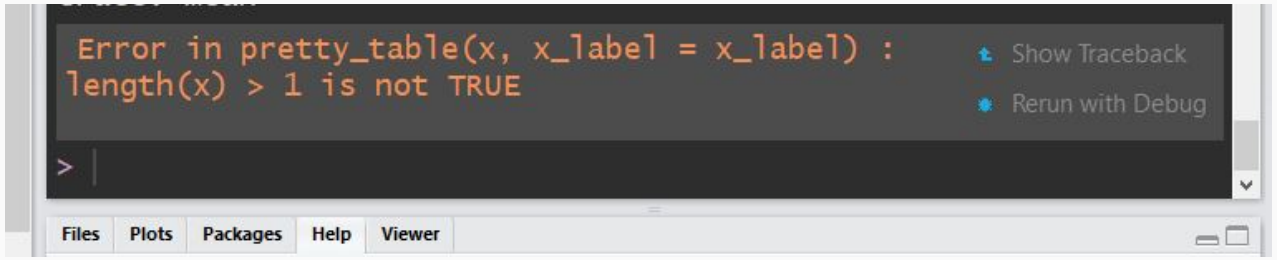
browser()

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

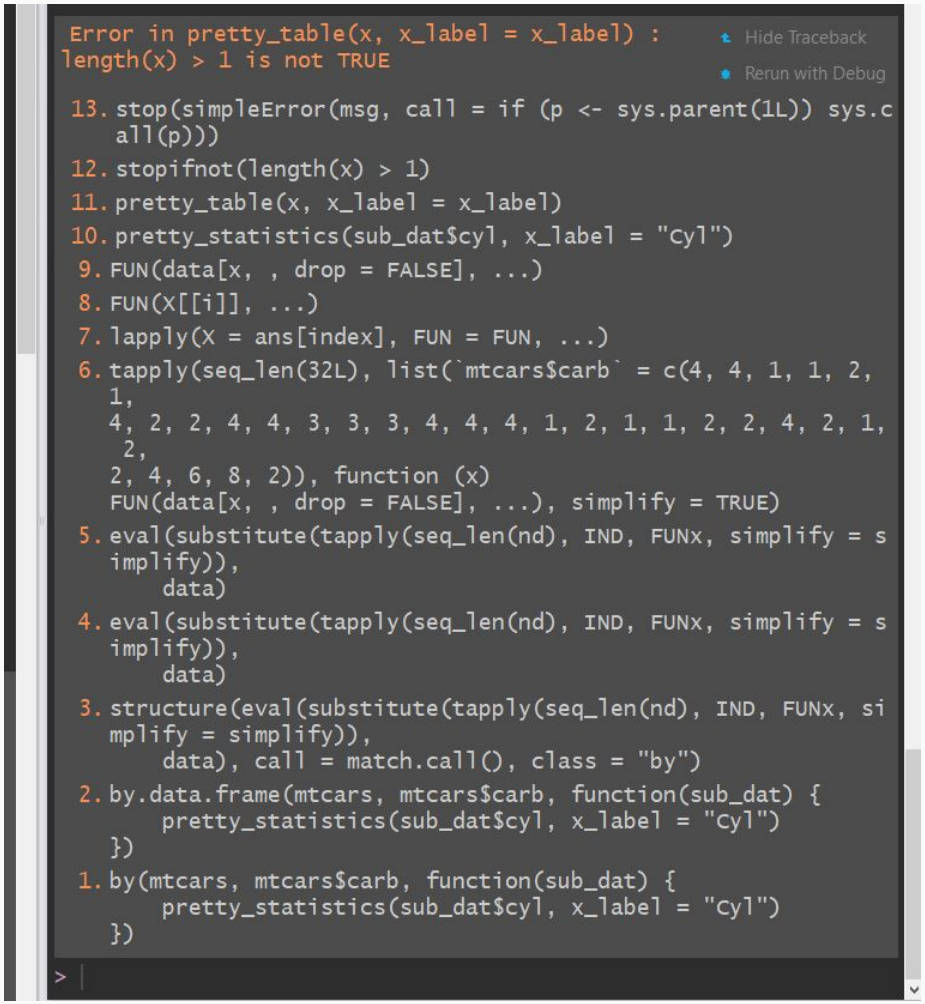
traceback()

Understanding the call stack:



traceback()

Understanding the call stack:



Recover

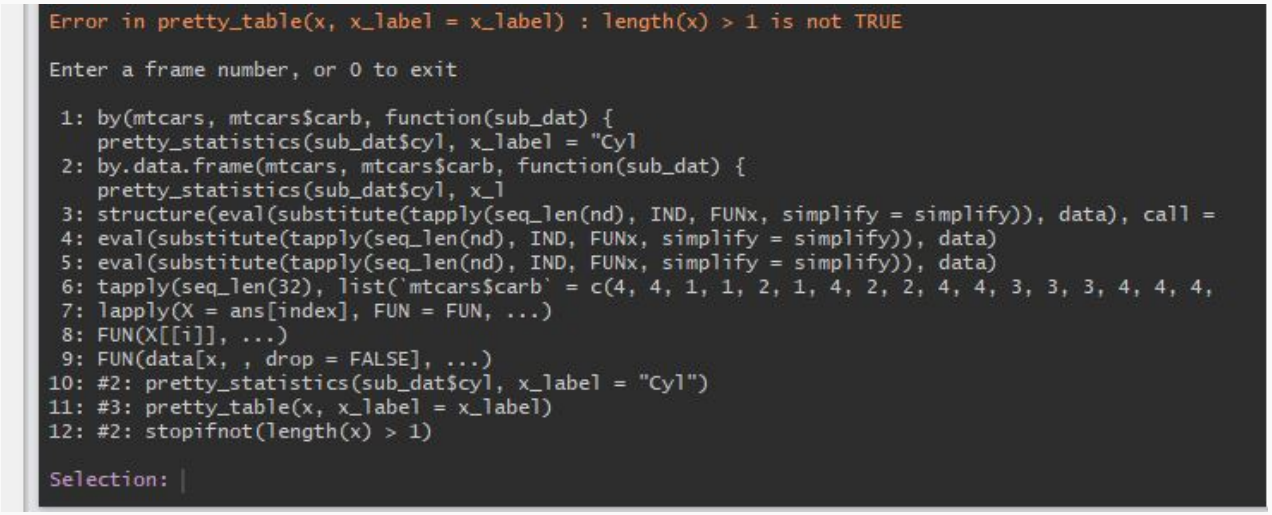
Being able to chose an environment from the call stack:

```
# on
options(error = recover)

# off
options(error = NULL)
```

Recover

Being able to chosse an enrivonment from a call stack:



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 space

```
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 other 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.

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

Speed

Computing speed can become an issue. Avoid common pitfalls:

- don't grow, but replace
- vectorize where possible
- check the computing speed

`?system.time`, microbenchmark or profiling tools

```
n <- 2e+4
data <- data.frame(x = runif(n),
                   y = runif(n),
                   z = seq_len(n))
```

Speed

Don’t grow!

```
system.time({
  new_data <- NULL

  for(row_nr in seq_len(NROW(data))){
    new_data <- cbind(
      data[row_nr,],
      result = exp(data$x[row_nr]) /
        log(data$z[row_nr]) +
        5 * sqrt(data$y[row_nr]))
  }
})

>    user  system elapsed
>    3.77    0.00    3.77
```

Speed

Replace!

```
system.time({
  n_rows <- dim(data)[1]
  data$result <- rep(NA, n_rows)

  for(row_nr in seq_len(n_rows)){
    data$result[row_nr] <- exp(data$x[row_nr]) /
      log(data$z[row_nr]) +
      5 * sqrt(data$y[row_nr])
  }
})

>    user  system elapsed
>    0.51    0.04    0.54
```

Speed

Vectorize!

```
system.time({
  data$result <- exp(data$x) / log(data$z) +
    5 * sqrt(data$y)
})

>    user  system elapsed
>       0       0       0
```

Speed

Compare the speed of different implementations using:

`microbenchmark::microbenchmark`

```
get_mean1 <- function(x){
  weight <- 1/length(x)
  out <- 0
  for(i in seq_along(x)){
    out <- out + x[i] * weight
  }
  return(out)
}

get_mean2 <- function(x){
  sum(x)/length(x)
}
```

Speed

Compare the speed of different implementations using:

`microbenchmark::microbenchmark`

```
x <- rnorm(500)
microbenchmark::microbenchmark(
  mean(x), get_mean1(x), get_mean2(x))

> Unit: microseconds
>          expr  min   lq    mean median    uq     max neval cld
>       mean(x)  4.1  4.2  4.793    4.3  4.50    27.0   100   a
>  get_mean1(x) 29.3 30.3 81.544   30.3 30.45 5128.3   100   a
>  get_mean2(x)  1.5  1.5 20.609    1.6  1.70 1877.8   100   a
```

Speed

Programming advice

Don’t worry about speed before it becomes an issue.

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!
- Use `saveRDS()` and `readRDS()` for objects that require long computations

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")
```

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?