Introduction to Programming with R

Dries Debeer & Benjamin Becker

31. September and 01. October 2021

Zurich R Courses

Table of Content

Introduction

Who are we?

Dries Debeer

Senior Researcher at itec (imec research group at KU Leuven)
scDIFtest, permimp, eatATA, mstDIF

dries.debeer@kuleuven.be

Benjamin Becker

Researcher at IQB (Statistics Department)

eatGADS, eatDB, eatATA, pisaRT

b.becker@iqb.hu-berlin.de

Introduction

Who are you?

1. Occupation, employer?
2. Previous knowledge and experience
   - with R?
   - with other statistical software?
   - with other programming languages?
3. Specific interest/motivation for this workshop?

1. Increase efficiency!
   - Save time and nerves
   - Avoid errors and bugs
   - High transfer effect to all projects (with data analyses)
2. Successful collaborations (including with your future self!)
3. Code as deliverable (i.e., part of research paper)

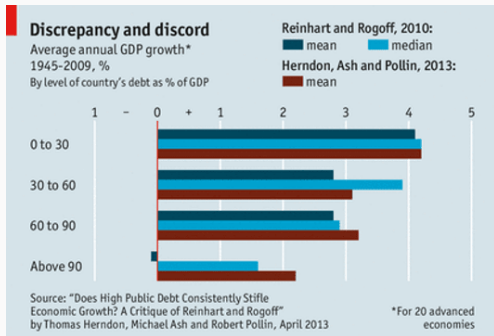Two of your worst collaborators:

1. Past Self
   - the biggest mess in existence
   - Did not document anything
   - Uses a completely different style of writing code
   - does not reply to e-mails
2. Future Self
   - Has the memory of a goldfish
   - Will have zero understanding for your current brilliance

Motivation

Coronavirus

Modelling behind lockdown was an unreliable buggy mess, claim experts

Data that predicted 500,000 could die in UK unless extreme measures were taken are impossible to replicate, say scientific teams

Science

By Hannah Boland and Ellie Zolfagharifard

'In our commercial reality, we would fire anyone for developing code like this'

'Any business that relied on it to produce software for sale would likely go bust'

'It looks more like a bowl of angel hair pasta than a finely tuned piece of programming'

'The early 2000s were yet another confirmation that their modelling approach was flawed to the core'

Motivation

Concept of Technical Debt

- We write (messy) code for data cleaning/analyses
- We decide on data sets/models/graphs/tables/...
- We try to publish it, get a major revision
- We need to rerun some analyses
- Modifying/extending our code is more difficult than it should be

Trade-off

- Being fast vs. writing (or refactoring) perfect code

But also

- Write better R code

An introduction to R as a Programming language

- Better practical R skills
- Better understanding of R (and programming)
- Different framing: R as a programming language

Agenda

Day 1

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

Day 2

- Functions (part II)
- Debugging
- Functions (part III)
- Good programming practices

RStudio setup

RStudio setup

1. Copy the course content from the usb-stick to a directory on your machine
2. Open RStudio
3. Choose `File < New Project ...`
4. Choose `Existing Directory`
5. Browse to the directory on your machine where you copied the course content and select the "Intro-R-programming" folder as the `Project working directory`
6. Click `Open in new session`
7. Click `Create Project`

RStudio setup - optional

1. Choose `Tools < Global options`
2. Under `General`
   - DON'T `Restore .RData into workspace at startup`
   - NEVER `Save workspace to .Rdata on exit:`
3. Further personalize RStudio

Basic elements & data types

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

— John Chambers

- What are objects?
- Atomic vectors
- Vector structures
- Subsetting
- Replacement

What are objects?

- Data-structures that can be used in computations
- Collections of data of all kinds that are dynamically created and manipulated
- Can be very small, or very big. → *Everything in R is an object*
- Elementary data structures can be combined in more complex data structures
- Creating new types of *complex* objects is part of programming in R (S3, S4)

Basic object types	
logical	TRUE, FALSE, NA
integer	1L, 142, -5, …, NA
double	1.0, 1.25784, pi, …, NA
	NaN, -Inf, Inf
character	"1", "Some other string", …, NA

mulitple values in one object $\rightarrow$ `length()` starting from 0

Elements of the same type can be combined into an atomic vector using c.

```
c(3.3, 2.44, 9, 634)

> [1]   3.30   2.44   9.00 634.00
```

All elements are of the same type!

An important object type with special behavior is NULL.
It is an empty object that can be interpreted as *nothing.* It's
length is 0.

```
length(NULL)

> [1] 0
```

NULL is mostly used as a default argument in functions, in
order to create some default behavior.

?seq Creates a vector with a sequence of numerical values.

```
seq(0, 10, by = 2)

> [1]  0  2  4  6  8 10

seq(0, 1, length.out = 11)

>  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

seq_along and seq_len are shortcuts.

```
seq_along(c("a", "b", "c", "d"))

> [1] 1 2 3 4

seq_len(10)

>  [1]  1  2  3  4  5  6  7  8  9 10
```

Avoid : when programming!

?rep Creates a new vector by repeating the elements of a vector.

```
rep(1:3, each = 2)

> [1] 1 1 2 2 3 3

rep(1:3, times = 2)

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

?rep Creates a new vector by repeating the elements of a
vector.

```
rep(c("a", "b", "c"), times = 2)

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

rep(c("this", "may", "be", "useful", "!"), 1:5)

> [1] "this"   "may"    "may"    "be"     "be"     "be"     "
> [9] "useful" "useful" "!"      "!"      "!"      "!"      "
```

Useful Functions

?paste Creates a character vector by pasting multiple vectors together.

```
paste("one", "big", "string", sep = " ")

> [1] "one big string"

paste0("word_", seq(1, 4))

> [1] "word_1" "word_2" "word_3" "word_4"

paste(c("ONE", "TWO"), seq(1, 3),
      sep = " || ", collapse = "_-_")

> [1] "ONE || 1_-_TWO || 2_-_ONE || 3"
```

?unique Creates a vector with the unique values of a vector.

```
unique(c("b", "a", "a", "b"))

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

Useful Functions

?`sort` Creates a sorted version a Vector.

```
sort(c("b", "a", NA, "a", "b"))

> [1] "a" "a" "b" "b"

sort(c("b", "a", NA, "a", "b"), na.last = TRUE)

> [1] "a" "a" "b" "b" NA

sort(c(4, 2, 6, 1, 3, 5), decreasing = TRUE)

> [1] 6 5 4 3 2 1
```

Exercises

Coercion/Conversion

Automatic conversion:
NULL → logical → integer → double → character

```
1 + TRUE
```

```
> [1] 2
```

Explicit conversion:
as."type"() as.vector(, mode = "type")

```
as.logical(0:5)
```

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

Check type using: `is."type"()`

```
is.null(NULL)

> [1] TRUE
```

Check type using: `typeof()`

```
typeof(TRUE + FALSE)

> [1] "integer"
```

Assignment

In order to compute with objects efficiently, names can be assigned to the objects using the assignment operator `<-` (or `=`)

```
my_object <- TRUE
my_object

> [1] TRUE
```

- The objects (with references) that are available to a user can be seen in the global environment using `ls()`.
- R overrides previous assignments without a message. Removed objects (`rm(objectName)`) cannot be restored.

→ *May the source code be with you!*

Attributes

Attributes can be attached to objects. An attribute:

- has a name
- is itself also an object
- attributes are easily lost in computations. (One of the reasons to use OOP with classes and methods.)

```
my_object <- structure(5,
                       my_attribute = "string",
                       other_attribute = FALSE)
attributes(my_object)

> $my_attribute
> [1] "string"
>
> $other_attribute
> [1] FALSE
```

Attributes

There are several attributes with a specific use: `"names"`, `"dim"`, `"class"`, `"levels"`

- `"names"` is a character vector that contains the names of elements of the vector/object. Names can be printed and set using `names(object) <-` .
- `"dim"` is an integer vector that specifies how we should interpret the vector (i.e., as a matrix, as an array). The dimensions of a vector can be printed and set using `dim(object) <-` .
  $\rightarrow$ a `matrix` or `array` is a vector with a `"dim"` attribute.

Attributes

- `"class"` is a character vector that contains class names.
  Classes can be printed and set using `class(object) <-`
  .
  See Object Oriented Programming (S3).

- `"levels"` is a character vector that contains the names
  levels of a factor. Levels can be printed and set using
  `levels(factor) <-` .

A factor in R is actually an integer vector with

- a `"class"` attribute set to `"factor"`
- a `"levels"` attribute set to the level-labels that correspond to the integer values from 1 to the highest integer value in the integer vector.

More basic object types	
complex	`1 + 2.31i`, … `NA`
raw	`as.raw(2)`, `charToRaw("a")`
expression	`expression(1+1, sum(a, b))`
language	a function call, `quote(1 + y)`
closure	`function(x) x - 1`, `mean`
builtin	`sum`, `c`
special	`for`, `return`
environment	an environment
symbol	`quote(x)`
…	…

More basic object types	
list	`list()`, `as.list()`, …
matrix	a **vector** with `"dim"` argument: two dimensions
	`matrix()`, `as.matrix()`
	matrix algebra
array	a **vector** with with `"dim"` argument
data.frame	a **list** with vectors of equal length
	`data.frame()`, `as.data.frame()`

List

A list is a "vector" that can contain any type of elements

- the types of elements can differ ↔ atomic vectors
- possible elements including lists → recursive
- can have attributes

```
my_list <- list("this",
                a = list(a = c(1:2)))
my_list

> [[1]]
> [1] "this"
>
> $a
> $a$a
> [1] 1 2
```

Matrix & Array

A matrix or an array is a vector with a `"dim"`-attribute

- mostly useful for numeric vectors (integer and double)
- matrix algebra! `t(matrix)`, `%*%`, `aperm(array)`, ...
- matrix has two dimensions, array has *n* dimensions You can create an matrix array using:

- `cbind(vector1, vector2)`
- `rbind(vector1, vector2)`
- `matrix(vector, ncol = 4, nrow = 2)`
- `array(vector, dim = c())`

Data.frame

A data.frame is a list of (named) vectors of equal length.

- has dimensions (but not a `"dim"`-attribute)
- the columns are the vectors
- the vectors can be lists (using `I( )`).
- a data.frame has row names (but ignore these)

A subset of elements from a vector can be accessed using
`object[selection]`, where `selection` is:

1. a **logical** vector with the same length of the original vector
   (`TRUE`: select; `FALSE`: don't select)
2. an **integer** vector indicating the indexes of the elements to
   select (or exclude)
3. a **character** vector with the names of the elements to
   select

Using a **logical** vector:

- the logical vector should have the same length as the object. If shorter, the logical is repeated; if longer, NAs are added if TRUE. → always use the same length!
- handy when you want to select based on a condition related to the object values

Subsetting - Atomic vectors

Using a **logical** vector:

```
my_object <- c(a = 1, b = 5, c = 3, d = 8)
my_object[my_object > 4]

> b d
> 5 8
```

Subsetting - Atomic vectors

Using an **integer** vector:

- the integer vector can have any length (repeated indices are repeatedly selected)
- positive values mean *select*, negative values mean *drop*
- positive and negative values cannot be combined
- for integers higher than the number of elements in the vector, NAs are added
- using `which()` a logical vector is transformed in an integer vector with the indices of the elements that were TRUE
- double elements are truncated towards zero (using `as.integer()`)

Using an **integer** vector:

```
my_object <- c(a = 1, b = 5, c = 3, d = 8)
my_object[c(1, 2, 1, 2, 1, 2, 1 , 2, 1, 2, 1, 2)]

> a b a b a b a b a b a b
> 1 5 1 5 1 5 1 5 1 5 1 5
```

Using a **character** vector:

- the strings that match with the names of the elements in the vector are returned
- the character vector can have any length (repeated names are repeatedly selected)
- only selection is possible (dropping is not)
- strings that are not matched with names return NA

Using a **character** vector:

```
my_object <- c(a = 1, b = 5, c = 3, d = 8)
my_object[c("a", "c")]

> a c
> 1 3
```

A **sinlge** element from a vector can be accessed using `object[[selection]]`, where `selection` is:

- an **integer** value indicating the index of the element to select
- a **character** vector with the name of the element to select

```
my_object <- c(a = 1, b = 5, c = 3, c2 = 8)
my_object[[2]]

> [1] 5
```

Because arrays and matrices are atomic vectors (with a `"dim"` argument), the rules for atomic vectors apply.

Subsetting - Matrix & Arrays

In addition, selection is possible per dimension:

- separated by a comma [,]
- selection via character (match row or column names), integer (row and column number) or logical vectors
- the first vector selects the rows, the second the columns (and so on)
- **dimensions are dropped**, unless `drop = FALSE`

```
my_matrix <- matrix(c(11, 21, 12, 22), ncol = 2,
                    dimnames = list(paste0("row", 1:2),
                                    paste0("col", 1:2)))
my_matrix[,2]

> row1 row2
>   12   22
```

Finally, the selection element can also be a matrix (with one column per dimension). Each row in the matrix selects one value.

```
my_matrix <- matrix(c(11, 12, 21, 22), ncol = 2,
                    dimnames = list(paste0("row", 1:2),
                                    paste0("col", 1:2)))
selection_matrix <- rbind(c(1, 1), c(1, 2), c(2, 1))
my_matrix[selection_matrix]

> [1] 11 21 12
```

Subsetting - Lists

For lists, the rules are similar as for atomic vectors.

- `list[selection]` gives a list (i.e., a subset of the original list)
- `list[[selection]]` gives the element (which can be a list)
- `list[["element_name"]]` is the same as `list$element_name`

```
my_list<- list(a = 1, b = 5, c = 3, d = 8)
is.list(my_list["a"])

> [1] TRUE

is.list(my_list[["a"]])

> [1] FALSE
```

Because data.frames are lists, the rules for lists apply.

```
my_dat <- data.frame(col1 = c(11, 21),
                     col2 = c(12, 22))
my_dat[1]

>   col1
> 1   11
> 2   21
```

In addition, the selection rules for matrices can be used:

- selection per row and column (note the `drop` argument)
- selection via a matrix with two columns

```
my_dat <- data.frame(col1 = c(11, 21),
                     col2 = c(12, 22))
my_dat[,"col1", drop = FALSE]

>   col1
> 1   11
> 2   21
```

Programming advice

Code defensively: always use `, drop = FALSE`

Element Replacement

A subset of elements from a vector or vector structure can be replaced using `object[selection] <- new_values`:

- the modifications are done in place
- the structure and class of the object stay unchanged
- the length of the new values should correspond with the length of the selection (the number of elements to replace should be a multiple of the number of new values)
- *only for lists*: the replacement can be `NULL` (which removes the element from the list)

```
my_dat <- data.frame(col1 = c(11, 21),
                     col2 = c(12, 22))
my_dat[1, 2] <- 33
my_dat

>   col1 col2
> 1   11   33
> 2   21   22
```

Exercises

Flow & conditional programming

R has specific tools (functions) that help organize the flow of computations.
You can make computations conditional on other objects ("conditional computation")
The most commonly used tools are:

- if (+ else)
- ifelse

if statements have the basic form

```
if(test){
  some_computations
}
```

- test should be either TRUE or FALSE (or code that results in one of both).
- If test == TRUE, than some_computations is executed, if test == FALSE, than not.
- Important: test should have length 1. If not, only the first element is considered.

Conditional Computation - if

else can be added, but it is optional

```
if(test){
  some_computations
} else if (test_2){
  other_computations
} else {
  more_computations
}
```

Vectorized, elementwise	
==	equal to
!=	NOT equal to
>, >	is greater, less than
>=, >=	is greater, less than or equal to
&	AND operator
|	OR operator
xor	exclusive OR

Not Vectorized	
`identical()`	identical to
`any()`	at least one TRUE
`all()`	all TRUE
`&&`	AND operator
`||`	OR operator
`is.character()`, `is.data.frame()`, …	

Compare:

```
c(TRUE, TRUE) & c(FALSE, TRUE)

> [1] FALSE   TRUE

c(TRUE, TRUE) && c(FALSE, FALSE)

> [1] FALSE
```

The *test* should have length 1!

```r
# only the first element is evaluated
age <- c(8, 17, 39, 55)
if (age >= 18) {
  "can vote"
} else {
    "too young"
}

> Warning in if (age >= 18) {: the condition has length
> 1 and only the first element will be used

> [1] "too young"
```

Typical uses

```r
if(any(is.na(x))){
  stop("computation impossible due to NA values")
}

if(!is.integer(vector)){
  warning("'vector' is automatically converted to interger.
          This may affect the results")
  vector <- as.integer(vector)
}

if(is.null(argument)){
  # default computations
} else if (argument == specific_value) {
  # other computations
}
```

Programming advice

- *if* is almost always used inside of functions or loops
- If possible, avoid using *else*
- Use meaningful initialization, early return(), stop(), etc. instead

Solution using *if* and *else*

```
age <- 17
if (age >= 18) {
  vote <- "can vote"
} else {
  vote <- "too young"
}
vote

> [1] "too young"
```

Solution using meaningful initialization

```
age <- 17
vote <- "too young"
if (age >= 18) {
  vote <- "can vote"
}
vote

> [1] "too young"
```

A vectorized version is `ifelse()`.

```r
# all elements are evaluated
age <- c(8, 17, 39, 55)
ifelse(age >= 18,
       yes = "can vote",
       no = "too young")

> [1] "too young" "too young" "can vote"  "can vote"
```

Go-to tool for conditional recoding

```
age_estimated <- c(10, 20, 35, 60)
age_self_rep <- c(NA, 17, 39, NA)

# Use available information, prioritize self report
ifelse(!is.na(age_self_rep),
       yes = age_self_rep,
       no = age_estimated)

> [1] 10 17 39 60
```

Exercises

Loops & Iteration

R has specific tools (functions) that help organize the flow of computations.
You can repeat a similar computation multiple times typically with changing options ("iteration"). The most commonly used tools are:

- loops (`repeat`, `while`, `for`)
- functionals (`apply` - family)

for statements have the basic form

```
for (element in vector) {
  computation
}
```

For each element in the vector, the computation is executed. Often, the computation depends on the element in that iteration.

```r
# iterate over a numeric vector
for (index in 1:3){
  cat(" computation -")
}

>  computation - computation - computation -

# iterate over a character vector
for (name in c("Alice", "Bob", "Casey")){
  if(name == "Bob") cat(" This was Bob -")
  else cat(" Not Bob -")
}

>  Not Bob - This was Bob - Not Bob -
```

Nested loops (over the rows and columns of a matrix)

```
matrix <- matrix(NA, nrow = 2, ncol = 3)
for (rowNr in 1:2){
  for (colNr in 1:3){
    matrix[rowNr, colNr] <- rowNr * 10 + colNr
  }
}
matrix

>      [,1] [,2] [,3]
> [1,]   11   12   13
> [2,]   21   22   23
```

while statements have the basic form

```
while (condition){
  computation
}
```

As long as the condition is TRUE, the computation is executed. Often, the computation depends on something that is related to the condition.

Sample five random values from a normal distribution, the distance between the minimum and maximum should be at least 4.

```r
max_dif <- 0
while (max_dif <= 4){
  cat("|")
  values <- rnorm(5)
  max_dif <- max(values) - min(values)
}

> ||||||||||||||||||||||||||||||||||||

max_dif

> [1] 4.830036

round(values, 3)
```

76

repeat statements have the basic form

```
repeat {
  computation
}
```

Without a break the computation is repeated infinite times

- `next` starts next iteration
- `break` ends iteration (of the innermost loop)

```r
index <- 0
repeat {
  index <- index + 1
  if (index %in% c(3, 5)) next
  if (index > 6) break
  print(index)
}

> [1] 1
> [1] 2
> [1] 4
> [1] 6
```

Iteration - Good practice

Programming advice
Use `seq()`, `seq_len()`, or `seq_along()`.

```
x <- numeric()
for (index in 1:length(x)){
  print(index)
}

> [1] 1
> [1] 0

for (index in seq_along(x)){
  print(index)
}
```

Programming advice

Don't grow, replace.

```r
x <- letters
result1 <- numeric()          # grow
result2 <- numeric(length(x)) # replace
for (index in seq_along(x)){
  result1 <- c(result1, paste(index, x[index]))  # grow
  result2[index] <- paste(index, x[index])        # replace
}
```

A functional is a function that takes another function as an argument.

Focus on the `apply`-family. These functions *apply* a function repeatedly.

Can be seen as an abstraction of a for loop, with the following advantages

- requires less code to write
- does not store intermediate results
- no need to replace / grow

Functionals

The most commonly used functionals are:

- `lapply` vector / list $\rightarrow$ list
- `sapply` vector / list $\rightarrow$ vector (matrix)
- `apply` matrix / array / data.frame $\rightarrow$ vector (matrix)
- `tapply`, `by`, `aggregate`
- `mapply`, `Map`
- `rapply`, `eapply`, `vapply`

All of which have an argument that should be a function.

lapply

data.frames are lists with the columns as elements:

```
lapply(iris, FUN = class)

> $Sepal.Length
> [1] "numeric"
>
> $Sepal.Width
> [1] "numeric"
>
> $Petal.Length
> [1] "numeric"
>
> $Petal.Width
> [1] "numeric"
>
> $Species
> [1] "factor"
```

lapply

- any type of element can be used
- other arguments can be passed through

```
means <- lapply(airquality, FUN = mean, na.rm = TRUE)
str(means)

> List of 6
>  $ Ozone  : num 42.1
>  $ Solar.R: num 186
>  $ Wind   : num 9.96
>  $ Temp   : num 77.9
>  $ Month  : num 6.99
>  $ Day    : num 15.8
```

apply

- for objects with dimension (matrix, array, data.frame)
- apply over (a) chosen dimension(s)

```r
my_matrix <- matrix(1:6, nrow = 2)
apply(my_matrix, 1, max)    # apply per row

> [1] 5 6

apply(my_matrix, 2, max)    # apply per column

> [1] 2 4 6
```

```
my_array <- array(1, dim = c(2, 3, 4))
apply(my_array, c(1, 2), sum)  # per row and column

>      [,1] [,2] [,3]
> [1,]    4    4    4
> [2,]    4    4    4

apply(my_array, 3, sum)        # per "third dimension"

> [1] 6 6 6 6
```

Exercises

Functions I

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

— John Chambers

Function Calls

Computing in R happens through function calls. A function is applied to one or more objects, and returns an object after the computation.

Figure 1: A function call.

The typical use is:
```
function(object1, argument = object2)
```

- Computations that seem not to be done using function calls are actually also function calls. Try `` `<-` ``(a, 5) or `` `>` ``(5, 2)
- most functions that seem not to return an object, return it invisibly. Check (a <- 5).

Building Blocks

Functions are the building blocks of R code. Writing functions allows you to organize and optimize the computations that you want to do.
Functions should:

- have a clear purpose
- be well documented
- be portable

Central stepping stone for R users:
Move from solely using functions written by others to writing your own functions.

- Name
- Arguments/Formals (input)
- Body (what happens inside, R-code with the computations)
- Output

```
countNA <- function(x) {      # Name
                              # Arguments/Formals
  out <- sum(is.na(x))        # Body
  out                         # Output
}
```

Function Names

Every function needs a (meaningful) name!

- Usually **a verb** (what does the function do?)
- Avoid existing names
- Better longer than unclear
- CamelCase vs snake_case

Function Names

Good

- computeAIC()
- removeNAs()
- drop_NA_rows()
- factor_to_dummies()

Bad

- myFun()
- foo()
- statistics()
- data_preparation()

Most functions take one or multiple inputs.
These are usually:

- One or two data arguments
- Additional Options

Arguments

Examples for zero arguments

- getwd()
- Sys.time()

Arguments

Examples for one argument

- dim()
- names()

Examples for multiple arguments

- mean()
- median()
- lm()

Programming advice

Less arguments = better!

Functions usually return a single object, namely the last evaluated object.

```
get_log_xtox <- function(x) {
  x_x <- x^x
  out <- log(x_x)
  out
}
get_log_xtox(2)

> [1] 1.386294
```

Often arguments have to by objects of a specific type.

```r
sum(c("a", "b", "c"))  # gives an error
```

The documentation typically gives (or should give) information about what objects the arguments should be. Check `?sum`

Exercises

Functions II

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

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)
```

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
```

Not writing a function:

```r
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 2: A pure function.

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
```

Multiple return objects can be combined in a list!

```r
                              # 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.

```r
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 3: A function with side effect.
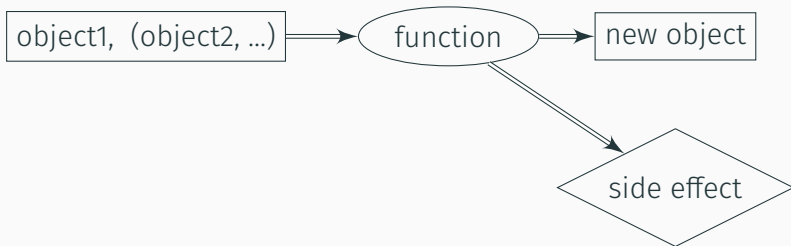
Side Effects

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

```r
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")
```

Graphics output

Output
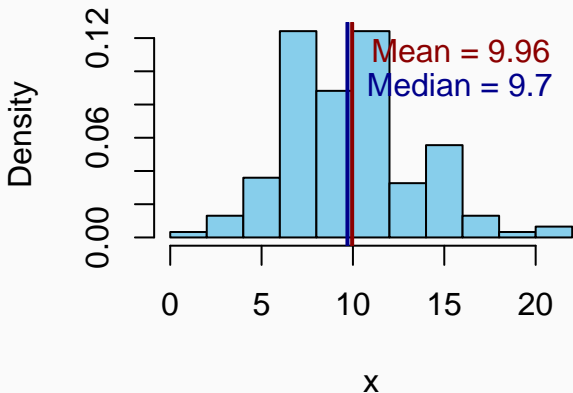
Programming advice

- Write pure functions (no-side effects)
- Write separate functions for side effects
- Plotting functions should return NULL or the plot as an object
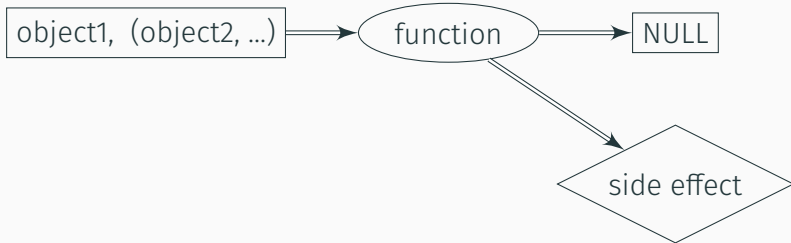
Figure 4: A side effect function.

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: computation is interrupted without return object!
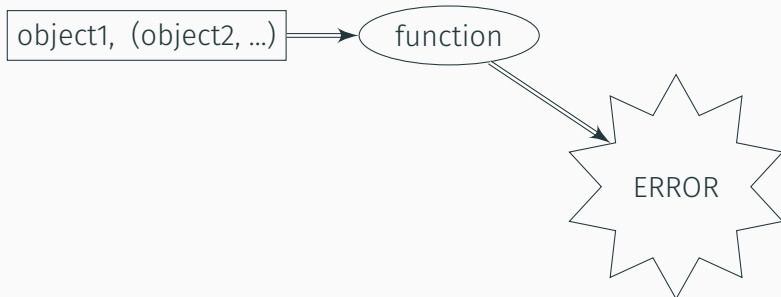
Figure 5: Computation with Error.

?`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`

```r
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
```

Message & warning: computation is NOT interrupted!
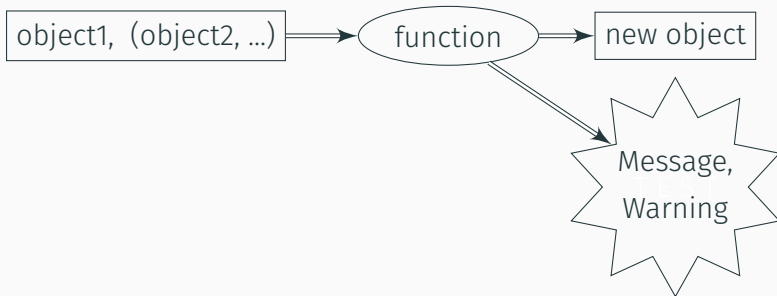
Figure 6: A message or warning.

Programming advice

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

What happens if the user omits an argument?

```r
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
```

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

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
```

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

```r
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

Debugging

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

Inspecting a function interactively

```r
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

Understanding the call stack:

traceback()

Understanding the call stack:

Being able to chose an environment from the call stack:

```
# on
options(error = recover)

# off
options(error = NULL)
```

Being able to chosse an enrivonment from a call stack:

Turning warnings into errors

```
# on
options(warn = 2)

# off
options(warn = 1)
```

Exercises

Functions III

Where does a function find objects?

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

```r
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.

Name masking!

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

```r
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.

```r
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
```

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
```

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

... (dot-dot-dot)

Examples:

- `?sum`
- `?save`
- …

`...` 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.

plot example

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

`plot` example

Wind (mph)
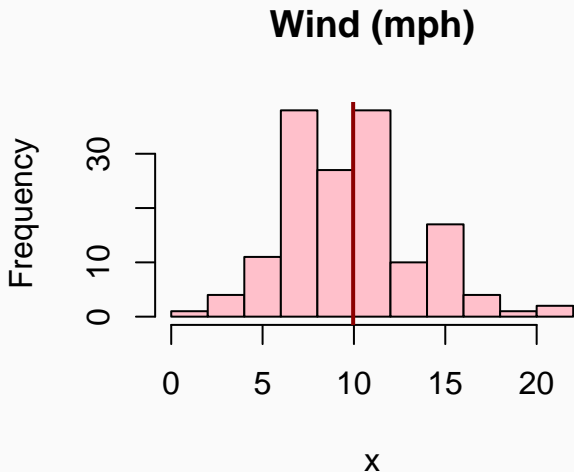
... dot-dot-dot

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
```

Performing an action when the function terminates.

```r
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.

```r
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'.
```
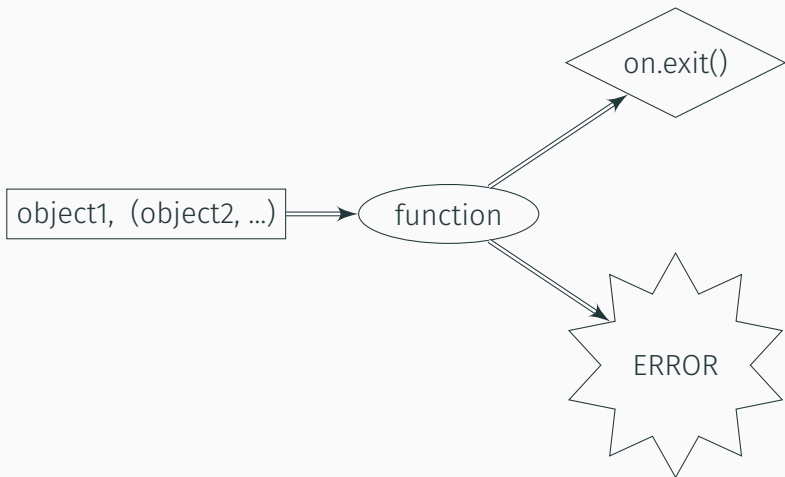
Figure 7: on.exit() with error.

Useful when your function has side effects:

```r
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:

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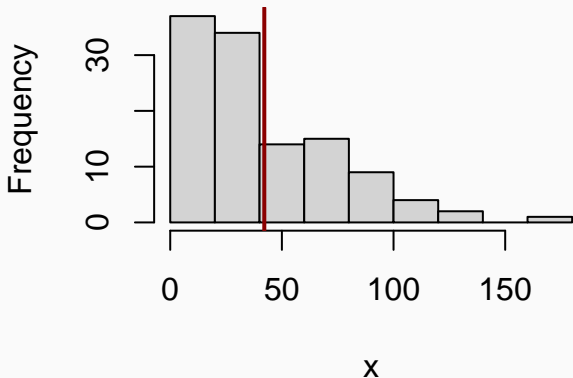
"To understand computations in R, two slogans are helpful: Everything that exists is an object. Everything that happens is a function call."

— John Chambers

Functions are also objects. They can be arguments.

- `apply`-family
- …

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

> [1] 1.791759
```

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
```

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")
```
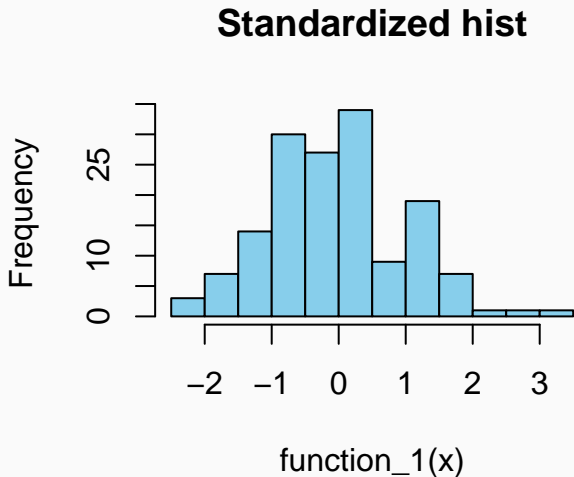
The return objects can also be functions:

Standardized hist

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
```

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

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)

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

# versus

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

Indentation

```r
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

```r
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

```r
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`

```r
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)
```

```r
## 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

Forget about `attach()`!

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

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

```r
# 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))
```

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!

```r
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
>       4       0       4
```

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
>    0.64    0.05    0.68
```

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`

```r
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)
}
```

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)  4.2  4.5   5.212    4.6  4.8   31.0   100   a
>  get_mean1(x) 31.2 31.4  86.838   31.5 31.7 5534.3   100   a
>  get_mean2(x)  1.5  1.6  21.481    1.6  1.8 1968.6   100   a
```

Programming advice

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

"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

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()
```

Keep it clean

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

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)

```r
# 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`

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

- 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?