Module 2 (R basics part II)

Contents

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

Project co-financed by the European Union under the European Social Fund

1 Lists

Lists are ordered tuples that may "wrap" any R objects. Recall that atomic vectors' elements must be of the same type. The elements of a list do not have to be of the same type. That is why lists are also called **generic vectors**. Lists may include sublists: That is why, on the other hand, they also belong to the class of recursive types.

Why to use lists? Many compound R objects are represented as lists, e.g. data frames, statistical tests' outcomes, etc. Additionally, an R function may return only one object: lists overcome this limitation in some sense. Do note that an R list is not based upon a *linked list* data structure. Element access is $O(1)$, insert and delete is $O(n)$.

1.1 Creating lists

List constructor: the `list()` function.

```
(L <- list(1:2, 11:13, 21:24))
## [[1]]
## [1] 1 2
##
## [[2]]
## [1] 11 12 13
##
## [[3]]
## [1] 21 22 23 24
```

A more concise view:

```
str(L)
## List of 3
##  $ : int [1:2] 1 2
##  $ : int [1:3] 11 12 13
##  $ : int [1:4] 21 22 23 24
```

Type information:

```
typeof(L)
## [1] "list"
is.list(L)
## [1] TRUE
is.vector(L)
## [1] TRUE
length(L)  # it's a vector
## [1] 3
is.atomic(L)  # but not an atomic one
## [1] FALSE
is.recursive(L)  # recursive type
## [1] TRUE
```

List elements are of any type:

```
c(TRUE, 1, "one")  # coercion
## [1] "TRUE" "1"    "one"
list(TRUE, 1, "one")  # 3 atomic objects
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] "one"
str(list(TRUE, 1, "one"))
## List of 3
##  $ : logi TRUE
##  $ : num 1
##  $ : chr "one"
```

Also, the `c()` function may sometimes output a list:

```
c(TRUE, 1, "one", sum)  # sum is an object of type function
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] "one"
##
## [[4]]
## function (..., na.rm = FALSE)  .Primitive("sum")
```

Note the coercion order (w.r.t. return type) for `c()`: `NULL < logical < integer < double < complex < character < list`.

Exercise: *Let* `x` *be a numeric vector. Create a list consisting of 3 numeric vectors. First one should store all negative elements in* `x`*, second – all equal to 0, third – all that are positive.*

An exemplary solution

```
x <- c(-1, 5, -3, 9, 7)
result
## [[1]]
## [1] -1 -3
##
## [[2]]
## numeric(0)
##
## [[3]]
## [1] 5 9 7
```

Lists may contain sublists:

```
list(1, list(2, 3))
## [[1]]
## [1] 1
##
## [[2]]
## [[2]][[1]]
## [1] 2
##
## [[2]][[2]]
## [1] 3
str(list(1, list(2, 3)))
## List of 2
##  $ : num 1
##  $ :List of 2
##   ..$ : num 2
##   ..$ : num 3
```

To create an “empty” list of desired length, we call:

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

```
vector("list", 3)
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
```

We can also convert an atomic vector to a list:

```
as.list(1:2)
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
```

1.2 Selecting and modifying subsets of lists

1.2.1 Selecting subsets of lists

The indexing operator “[” returns a subvector of the same type as the original one.

```
c(1, 2, 3)[2]  # numeric vector of length 1
## [1] 2
list(1, 2, 3)[2]  # list of length 1
## [[1]]
## [1] 2
list(1, 2, 3)[-c(1, 3)]
## [[1]]
## [1] 2
list(1, 2, 3)[c(TRUE, FALSE, FALSE)]
## [[1]]
## [1] 1
```

1.2.2 Extracting individual elements

Vectors’ elements may be **extracted** with the “[[” operator.

```
list(1, 2, 3)[[2]]  # numeric vector
## [1] 2
c(1, 2, 3)[[2]]  # also works on atomic vectors
## [1] 2
list(1, 2, 3)[[-1]]  # only positive integers
## Error: attempt to select more than one element
list(1, 2, 3)[[c(TRUE, FALSE, FALSE)]]  # only positive integers
## Error: recursive indexing failed at level 2
```

Only one element may be extracted at a time (otherwise, we are selecting subsets).

If an index vector is of length > 1, recursive indexing is used.

```
L <- list(1, list(2, 3))
L[[2]]
## [[1]]
## [1] 2
##
## [[2]]
## [1] 3
L[[c(2, 1)]]  # the same as L[[2]][[1]]
## [1] 2
L[[c(2, 1, 2)]]  # don't go too far
## Error:  subscript out of bounds
```

1.2.3 Modifying lists

There are also "replacement" versions of "`[`" and "`[[`".

```
L <- list(1, 2, 3)
L[[1]] <- 1:5
str(L)
## List of 3
##  $ : int [1:5] 1 2 3 4 5
##  $ : num 2
##  $ : num 3
```

```
L[2:3] <- list(c(TRUE, FALSE), mean)
str(L)
## List of 3
##  $ : int [1:5] 1 2 3 4 5
##  $ : logi [1:2] TRUE FALSE
##  $ :function (x, ...)
```

Note:

```
L <- list(1, 2, 3)
L[2:3] <- c(TRUE, FALSE)  # L[[2]] <- TRUE; L[[3]] <- FALSE
str(L)
## List of 3
##  $ : num 1
##  $ : logi TRUE
##  $ : logi FALSE
```

```
L[2:3] <- 10:15
## Warning:  number of items to replace is not a multiple of replacement length
str(L)
## List of 3
##  $ : num 1
##  $ : int 10
##  $ : int 11
```

Keep in mind how does inserting `NULL`s work:

```
L <- list(1, 2, 3)
L[[2]] <- NULL  # different meaning
str(L)
## List of 2
##  $ : num 1
##  $ : num 3
```

The 2nd element has been removed.

A proper way to insert a `NULL` object is thus:

```
L <- list(1, 2, 3)
L[2] <- list(NULL)  # OK
str(L)
## List of 3
##  $ : num 1
##  $ : NULL
##  $ : num 3
```

1.3 Basic list operations

Lists are most often used only as data containers. Thus, there are not too many operations dedicated to lists. We will discuss the following ones:

- list merging,
- list unwinding,
- list replicating,
- applying operations on consecutive elements.

1.3.1 List merging

Here is a way to merge 2 list into one:

```
L1 <- list("a", 1)
L2 <- list(TRUE)
str(list(L1, L2))  # 2 sublists
## List of 2
##  $ :List of 2
##   ..$ : chr "a"
##   ..$ : num 1
##  $ :List of 1
##   ..$ : logi TRUE
```

```
str(c(L1, L2))  # merge
## List of 3
##  $ : chr "a"
##  $ : num 1
##  $ : logi TRUE
```

```
str(c(L1, L2, recursive = TRUE))  # merge + coerce
##  chr [1:3] "a" "1" "TRUE"
```

1.3.2 List unwinding and replicating

The `unlist()` function unwinds a list and coerces it to an atomic vector:

```
unlist(list(1, list("two", list(FALSE))))
## [1] "1"     "two"   "FALSE"
```

To replicate elements in a list, call:

```
rep(list(1:10, TRUE), 2)
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
## [1] TRUE
##
## [[3]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[4]]
## [1] TRUE
```

HUMAN CAPITAL NATIONAL COHESION STRATEGY EUROPEAN UNION EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

1.3.3 Applying functions on consecutive elements

`lapply(X, FUN)` applies the `FUN` function on each element of `X`.

```
lapply(list(c(1, 3, 2), c(3, 6, 2)), max)
## [[1]]
## [1] 3
##
## [[2]]
## [1] 6
```

The resulting list of the same size as `X`. We have:

```
result[[1]] <- FUN(X[[1]])
        ⋮
result[[n]] <- FUN(X[[n]])
```

```
lapply(list(1:5, 3), "-")  # unary operator `-`
## [[1]]
## [1] -1 -2 -3 -4 -5
##
## [[2]]
## [1] -3
```

Consult `?lapply`: There's a "`...`" parameter. The man page states that "`...`" is used to pass optional arguments to `FUN`.

```
lapply(list(3.1415, c(1.23, 9.99)), round, digits = 1)
## [[1]]
## [1] 3.1
##
## [[2]]
## [1]  1.2 10.0
```

Thus, we have: `result[[1]] <- round(3.1415, digits=1)` and `result[[2]] <- round(c(1.23, 9.99), digits=1)`.

Another example (see `?c`):

```
lapply(1:3, c, 10, 11, 12)
## [[1]]
## [1]  1 10 11 12
##
## [[2]]
## [1]  2 10 11 12
##
## [[3]]
## [1]  3 10 11 12
```

Make sure you understand why we get the above result.

Exercise: *Given a list consisting of character vectors, join them all to a single string.*

Exercise: *Given a list of numeric vectors, find the index of a vector with the largest sum of elements.*

Exercise: *Given a list of nonempty numeric vectors, order that list w.r.t. first vector's elements.*

Note the `mapply()` function that vectorizes a given function over all of given vectors' elements. A call to `mapply(FUN, X1, ..., Xk, SIMPLIFY=FALSE)` gives:

```
result[[1]] <- FUN(X1[[1]], ..., Xk[[1]])
        ⋮
result[[n]] <- FUN(X1[[n]], ..., Xk[[n]])
```

A very basic example:

HUMAN CAPITAL NATIONAL COHESION STRATEGY
EUROPEAN UNION EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

```
mapply("+", 1:3, 11:13, SIMPLIFY = FALSE)
## [[1]]
## [1] 12
##
## [[2]]
## [1] 14
##
## [[3]]
## [1] 16
```

Another example:

```
mapply("*", list(1:3, 4:6), list(10, -1), SIMPLIFY = FALSE)
## [[1]]
## [1] 10 20 30
##
## [[2]]
## [1] -4 -5 -6
```

And another one:

```
mapply(c, 1:3, 11:13, 21:23, 31:33, SIMPLIFY = FALSE)
## [[1]]
## [1]  1 11 21 31
##
## [[2]]
## [1]  2 12 22 32
##
## [[3]]
## [1]  3 13 23 33
```

1.4 Summary

Lists are vectors that store elements of any type. Notable functions: `lapply()`, `mapply()`, `unlist()`.

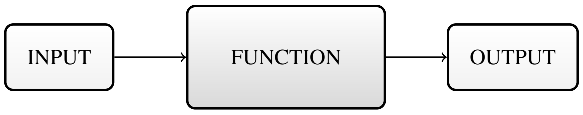
1.5 Bibliography

- R Core Team, *An introduction to R*, 2014, Sec. 6.1, 6.2
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 4
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 4

2 Functions

2.1 Introduction to functions

The functional programming paradigm emphasizes the role of functions in coding tasks (for theoretical foundations, see Church's λ-calculus developed in 1930s). Any computation is treated as the evaluation of mathematical functions which produce results that depend only on input data and not the program's state[1].



[1]Some state dependence is unavoidable; e.g. random number generators output results that rely on current *seed*; graphical functions use the drawing device's context; the `print()` functions changes the console's state, etc.

Exercise: *Approximate the value of π by adding $n+1$ initial terms of the Leibniz series, $\pi = 4\sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$.*

Input data: $n \in \mathbb{N}$. Output data: $x \in \mathbb{R}$ – approximation of π. Thus, $x = f(n)$ for some $f : \mathbb{N} \to \mathbb{R}$ that we would like to define now. Here is a solution:

```
n <- 10000   # input value
x <- suppressWarnings(4 * sum(c(1, -1)/(2 * (0:n) + 1)))
print(x)   # print output
## [1] 3.141693
```

The problem with the above code is that it **cannot be reused easily**, e.g. to approximate π for two different ns. Functions may help us with creating such abstractions.

```
f <- function(n)
   suppressWarnings(4*sum(c(1, -1)/(2*(0:n)+1)))
```

Some results:

```
f(10000)
## [1] 3.141693
f(1000000)
## [1] 3.141594
```

What is important, now f may be used in other computations, e.g.:

```
sin(f(1000000)) # OK, almost zero
## [1] -9.99999e-07
```

Note that f is made upon other predefined "building blocks". Each of them is a function: `*`, `sum()`, `c()`, `/`, `+`. Each of them also has a well defined input data specification and produces some specific kind of output. For example:

- $+, *, / : \mathbb{R}^i \times \mathbb{R}^j \to \mathbb{R}^{\max\{i,j\}}$
- $\mathtt{sum} : \mathbb{R}^i \to \mathbb{R}$
- $\mathtt{c} : \mathbb{R}^{i_1} \times \cdots \times \mathbb{R}^{i_k} \to \mathbb{R}^{i_1+\cdots+i_k}$

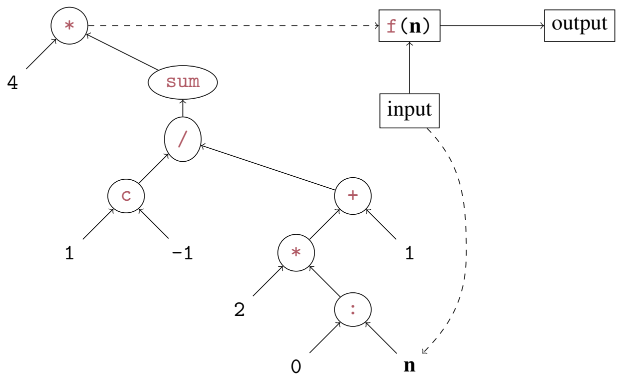
Note that

```
f <- function(n) 4*sum(c(1, -1)/(2*(0:n)+1))
```

may be rewritten as:

```
f <- function(n)
   `*`(4, sum(`/`(c(1, -1), `+`(`*`(2, `:`(0, n)), 1))))
```

Here is the syntax tree:



2.2 Defining R functions

Let us approach the topic in a more formal manner. The following expression creates a function object.

```
function(parameter_list) body
```

where:

- body is an R expression to be evaluated *on* given arguments. The resulting object is the function's output value.
- parameter_list is a comma-separated sequence of items of the form:
  - "parameter_name",
  - "parameter_name=default_value", or
  - "...".

Note that functions do not have to be bound to any name:

```
(function(x) x^2)(1:5)  # square
## [1]  1  4  9 16 25
```

Here, we have applied an *anonymous function* on 1:5. Anonymous functions are useful e.g. in connection with *apply.

```
lapply(list(1:3, 4:6), function(x) x^2)
## [[1]]
## [1] 1 4 9
##
## [[2]]
## [1] 16 25 36
```

However, most often we will use *named* functions.

```
square <- function(x) x^2
square(1:5)
## [1]  1  4  9 16 25
is.function(square)
## [1] TRUE
is.atomic(square)  # not an atomic type
## [1] FALSE
is.vector(square)
## [1] FALSE
is.recursive(square)  # a recursive type
## [1] TRUE
typeof(square)  # ``closure'' -> more on that later
## [1] "closure"
mode(square)
## [1] "function"
```

Note how a function is printed:

```
square
## function(x) x^2
```

Try to do the same with some built-in functions, like: shapiro.test, kmeans, sum, plot, etc.

A function's body is a single R expression. Many expressions may be "grouped" by using curly braces. Interestingly, "{" is another R function. Its return value is the result of evaluation of the last expression.

```
"{"(1, 2, 3) # just for the curious
## [1] 3
```

A user-friendly form:

```
{
   1
   2
   3
}
## [1] 3
```

HUMAN CAPITAL NATIONAL COHESION STRATEGY EUROPEAN UNION EUROPEAN SOCIAL FUND

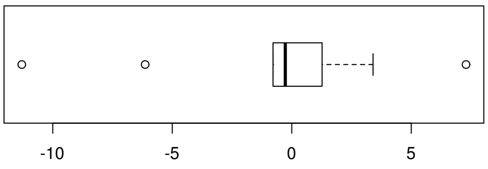
Project co-financed by the European Union under the European Social Fund

Exercise: *Write a function to remove outliers from a numeric vector.*

Let $\mathbf{x} = (x_1, \dots, x_n)$. x_i is an outlier if $x_i < Q1 - 1.5IQR$ or $x_i > Q3 + 1.5IQR$, where $Q1, Q3$ denotes the first and the third sample quartile, respectively, and $IQR = Q3 - Q1$. In other words, `Q1 <- quantile(x, 0.25)`, and `Q3 <- quantile(x, 0.75)`.

Here are some input data (for testing purposes):

```
set.seed(123)  # assure reproducibility
v <- rcauchy(10)  # some random data
boxplot(v, horizontal = TRUE)  # there are 3 outliers
```



Solution #1:

```
remove_outliers1 <- function(x)
  x[
    x >= quantile(x,0.25)-1.5*(quantile(x,0.75)-quantile(x,0.25))
    &
    x <= quantile(x,0.75)+1.5*(quantile(x,0.75)-quantile(x,0.25))
  ]
```

Check:

```
remove_outliers1(v)  # 7 observations - OK
## [1]  1.2691296 -0.7842432  3.4011811 -0.3850032 -0.1892392  0.1441052
## [7] -0.3514607
```

The problem with the above solution is that `remove_outliers1()` calls the (computationally) costly `quantile()` function far too many times.

Solution #2:

```
remove_outliers2 <- function(x) {
   Q13 <- quantile(x, c(0.25, 0.75))
   IQR <- diff(Q13)
   x[x >= Q13[1]-1.5*IQR & x <= Q13[2]+1.5*IQR]
}
```

Check:

```
remove_outliers2(v)  # 7 observations - OK
## [1]  1.2691296 -0.7842432  3.4011811 -0.3850032 -0.1892392  0.1441052
## [7] -0.3514607
```

Exercise: *Hirsch's h-index of a nonnegative numeric vector* `x` *of size n is defined as* $\mathsf{H}(\mathtt{x}) = \max\{i = 0, 1, \dots, n : x_{(n-i+1)} \ge i\}$, *where $x_{(i)}$ is the ith smallest value in* `x`, *with convention $x_{(n+1)} = x_{(n)}$.*

(An author has h-index H, if he/she has H publications with at least H citations and $n - H$ papers cited no more than H times)

It may be shown that

$$\begin{aligned}\mathsf{H}(\mathbf{x}) &= \left\lfloor \max\left\{ \min\{x_{(n)}, 1\}, \min\{x_{(n-1)}, 2\}, \dots, \min\{x_{(1)}, n\} \right\} \right\rfloor \\ &= \left\lfloor \bigvee_{i=1}^{n} x_{(n-i+1)} \wedge i \right\rfloor.\end{aligned}$$

Write a function `index.h()` *to calculate the* h*-index of a given numeric vector.*

Exercise: *The* `rle()` *function returns a named list (we'll cover named vectors later on), see* `?rle`. *Each of its components may be accessed by calling:*

```
rle(x)$lengths
rle(x)$values
```

Write a function `mode()` *to calculate the mode of a given integer vector, i.e. the most often occurring value.*

Exercise: *Let* `x`, `y` *be numeric vectors. Write a function to calculate the values of the empirical distribution function of* `x` *at each point in* `y`, *i.e.*

$$\widehat{F}_{\mathtt{x}}(y_j) = \frac{|\{x_i : x_i \le y_j\}|}{|\mathtt{x}|},$$

where $|\mathtt{x}|$ *denotes the number of observations in* `x`.

Hint: you may call `findInterval()` *or* `rle()` *(but not necessarily). Use neither* `splinefun()` *nor* `approxfun()`. *Note that elements in* `x` *may be non-unique.*

2.2.1 Invisible results

Each function must return some object. But sometimes we may wish to write functions that don't return anything interesting (cf. e.g. `plot()`). In such a case, the result may be "wrapped" with a call to the `invisible()` function.

```
printvec <- function(x) {
    cat(x, sep = ", ")
    cat("\n")
    invisible(NULL)  # return 'nothing', invisibly
}
```

```
printvec(1:4)  # retval suppressed from printing
## 1, 2, 3, 4
val <- printvec(1:4)
## 1, 2, 3, 4
print(val)
## NULL
```

2.2.2 Argument checking

We may pass *any* R object to an R function. This does not mean that the function is meant to accept it.

```
remove_outliers2(remove_outliers2)
## Error:  anyNA() applied to non-(list or vector) of type 'closure'
remove_outliers2(c("a", "b"))
## Error:  non-numeric argument to binary operator
remove_outliers2(list(1, 2, 3))
## Error:  'x' must be atomic
```

It is our responsibility to restrict the function's domain appropriately. For example, `remove_outliers2()` is designed to be applied on non-empty numeric vectors consisting of finite values only. In such a case, stopifnot() may be used. stopifnot(cond_1, ..., cond_n) throws an error if some cond_i does not meet all(!is.na(cond_i)) && all(cond_i).

```
remove_outliers3 <- function(x) {
   stopifnot(is.numeric(x))
   stopifnot(length(x) > 0, is.finite(x))
   Q13 <- quantile(x, c(0.25, 0.75))
   IQR <- diff(Q13)
   x[x >= Q13[1]-1.5*IQR & x <= Q13[2]+1.5*IQR]
}

remove_outliers3(v)
## [1]  1.2691296 -0.7842432  3.4011811 -0.3850032 -0.1892392  0.1441052
## [7] -0.3514607
remove_outliers3(list(1, 2, 3))
## Error:  is.numeric(x) is not TRUE
remove_outliers3(c(1, NA, Inf, NaN))
## Error:  is.finite(x) are not all TRUE
```

Exercise: *Add necessary* stopifnot() *calls to the functions you wrote.*

2.2.3 Side effects

Generally, our functions should have no side effects (of course except cases when we write a function that *aims* to print something on the console or draw some object in a figure). For instance, it is up to the function's user if he/she wants to print out the result on the console.

Exercise: *Write a function to calculate* $\sum_i x_i^p$ *for given* $\mathbf{x}$ *and* p*.*

This is **not** the right solution:

```
sumsq <- function(x, p) {
    stopifnot(is.numeric(x), is.numeric(p), length(p) == 1)
    cat("result =", sum(x^p), "\n")  # wrong, wrong, wrong
}
```

```
sumsq(1:5, 2)
## result = 55
```

This is not funny: many beginners write functions like that... The result cannot be used in further computations:

```
sumsq(1:5, 2)/5
## result = 55
## numeric(0)
```

So, in our case, a simple "sum(x^p)" at the end would be everything we need.

2.3 Storing functions for later use

2.3.1 Function libraries in R scripts

If your project consists of a couple of helper functions, think of putting them in one .R script. For instance, assume we have a file my_functions.R in the current working directory (see getwd()). The "main" script can input the functions' definitions by calling:

```
source("my_functions.R")  # see also CTRL+SHIFT+S in RStudio
```

Exercise: *Put all the functions you developed so far in a source script.*

Larger R projects (at least a few functions, some R source files, accompanying data, etc.) are usually developed as *R packages*. We will get to that in the near future. Until then we will make use of R packages made by other users.

2.3.2 R packages

CRAN (The Comprehensive R Archive Network) is by far the largest source of contributed open source packages. To download and install a package from CRAN, we call:

```
install.packages("packagename")
```

Once installed, we can *attach* the package's namespace to the current workspace:

```
library("packagename")
```

and then we are free to use the package's facilities.

Among other sources of R packages we find the Bioconductor archive (although it is not of everybody's interest).

Getting help on packages' facilities:

```
help(package = "pkgname")
example("function", package = "pkgname")  # e.g. pie, graphics
demo(package = "pkgname")  # e.g. graphics
vignette(package = "pkgname")  # e.g. Rcpp
```

For example, let us do some benchmarking. Call `install.packages("microbenchmark")` first.

```
library("microbenchmark") # attach the package
v <- rcauchy(10) # some random data
microbenchmark(sol1=remove_outliers1(v),
               sol2=remove_outliers2(v),
               sol3=remove_outliers3(v))
## Unit: microseconds
##   expr      min       lq    median        uq      max neval
##   sol1 1155.419 1183.512 1220.6430 1352.6820 2781.188   100
##   sol2  255.257  260.949  271.2765  291.0515  618.736   100
##   sol3  279.318  290.284  303.4480  343.7995  563.463   100
```

Note that we may call `microbenchmark()` without loading the microbenchmark package by writing: `microbenchmark::microbenchmark()`.

Another benchmark:

```
w <- rcauchy(10000) # a much longer vector
microbenchmark(sol1=remove_outliers1(w),
               sol2=remove_outliers2(w),
               sol3=remove_outliers3(w))
## Unit: milliseconds
##   expr      min       lq   median       uq      max neval
##   sol1 5.608257 5.649665 5.789402 6.643906 8.633521   100
##   sol2 2.108907 2.131800 2.157259 2.247371 3.904539   100
##   sol3 2.213817 2.237507 2.267257 2.344083 4.451001   100
```

Here is a function which *needs* the `stringi` package to operate.

```
random_string <- function() {
   library("stringi") # not installed -> error
   stri_rand_strings(1, 8, "[A-Za-z0-9#_!?$@%]")
}

random_string()
## [1] "DlMrv%Us"
```

2.4 Variable scope

Each assignment within a function's call makes a local binding.

```
f <- function(x) {
    fy <- x + 1
    fy
}

f(3)
## [1] 4
fy
## Error:  object 'fy' not found
```

We see that fy exists only when f() is being evaluated.

By the way, it is *possible* (but strongly discouraged, especially if you do not know what is lexical scope.) to refer to an “external” name from a function.

```
f <- function(x) {
    x + y
}

y <- 1:5
f(11:15)
## [1] 12 14 16 18 20
```

We will also see that it’s also *possible* to change the “external” variable. However, such a code is very difficult to maintain. Avoid such constructs at all means. Keep to the golden rule of functional programming: if you need an object, add an argument for it.

2.5 Functions’ arguments

2.5.1 Pass-by-value

Virtually all[2] R objects are passed by value. In other words, an argument behaves as a local variable.

```
g <- function(x) {
    x[1] <- x[1] + 1
    x
}

x <- 1
g(x)
## [1] 2
x
## [1] 1
```

Here x in g() and x in the user’s “workspace” have nothing in common. Changing the value of an argument within a function does not affect the value of the variable in the calling environment.

2.5.2 Default arguments

Arguments may be given default values. If such arguments are omitted from a call, defaults are used.

```
f <- function(a = 1, b = 2) a + b

f(10, 100)
## [1] 110
f(10)
## [1] 12
f(, 100)
## [1] 101
```

[2]The only exception: environments.

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN
SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

```
f(b = 100)
## [1] 101
f()
## [1] 3
```

Default arguments (if any) should be provided at the end of a function's parameter list.

2.5.3 Lazy evaluation

For efficiency reasons, arguments are not evaluated unless needed.

```
f <- function(x) { cat("before "); cat(x);  cat(" after\n") }
```

```
f(1)
## before 1 after
```

```
f({cat("now "); 1})
## before now 1 after
```

Such a behavior is called *lazy evaluation*. Note that in some cases an argument might never be evaluated. Laziness has some nice side effects: We may check whether an argument was omitted from the call:

```
f <- function(x = 1) {
    cat(missing(x), x)
}

f(1)
## FALSE 1
f()
## TRUE 1
```

Moreover, e.g. the `match.arg()` function may be used to match an argument to a predefined set of strings.

```
f <- function(kind = c("first", "second", "fird")) {
    kind <- match.arg(kind)
    kind
}

f("fird")
## [1] "fird"
f("s")
## [1] "second"
f("fir")
## Error:  'arg' should be one of "first", "second", "fird"
f("tenf")
## Error:  'arg' should be one of "first", "second", "fird"
```

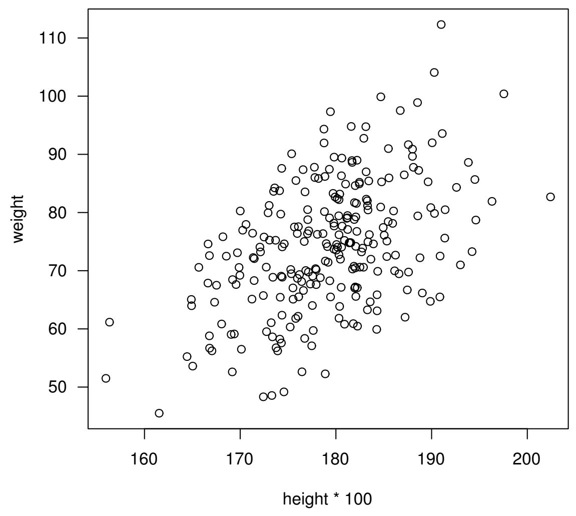
We can also find out what was the expression passed as an argument:

```
f <- function(x) cat(deparse(substitute(x)), "=", x)

f(5)
## 5 = 5
vals <- 1:5
f(vals)
## vals = 1 2 3 4 5
f(round(log(vals^2) + vals, 1))
## round(log(vals^2) + vals, 1) = 1 3.4 5.2 6.8 8.2
```

This is used in some graphical functions, like:

```
height <- rnorm(250, 1.79, 0.07)
weight <- rnorm(250, 23, 3) * height^2
plot(height * 100, weight, las = 1)  # note the axes labels
```



2.5.4 dot-dot-dot

The "..." parameter groups multiple arguments into one. It allows to create a function that takes an arbitrary (not known a priori) number of arguments or to easily pass a set of arguments to another function.

Each of the arguments "wrapped" with "..." may be accessed using "..1, ..2, ..."

```
f <- function(...) {
    print(..1)
    print(..2)
}

f(1, 2, 3)
## [1] 1
## [1] 2
f(1)  # sorry
## [1] 1
## Error:  the ...  list does not contain 2 elements
```

"..." may easily be converted to a list:

```
f <- function(...) {
    list(...)
}

str(f(1, 2, 3))
## List of 3
##  $ : num 1
##  $ : num 2
##  $ : num 3
str(f())
##  list()
```

Some interesting R functions with the dot-dot-dot param: `list()`, `c()`, `sum()`, `cbind()`, `rbind()`. Note that "`list(...)`" also catches argument names:

```
f <- function(x, ..., z = 1) {
    str(x)
    str(list(...))
    str(z)
}
```

```
f(1, 2, y = 3, 4, z = 5)  # we must refer to z explicitly here
##  num 1
## List of 3
##  $  : num 2
##  $ y: num 3
##  $  : num 4
##  num 5
```

dot-dot-dot arguments may be passed to another function:

```
f <- function(x, f, ...) {
    x + f(...)
}

f(1, c, 20, 30, 40)
## [1] 21 31 41
f(1, sum, 100, 200)
## [1] 301
```

See also: `?plot`, `?lapply`, `?optim`, `?uniroot`, etc.

Exercise: *Let U_1, U_2 be two independent random variables following the uniform distribution $U[0,1]$, cf.* `?runif`.
The Box-Muller transform provides a way to generate two independent R.V.s $Z_1, Z_2 \sim N(0,1)$:

$$\begin{aligned} Z_1 &= \sqrt{-2\ln U_1}\cos(2\pi U_2); \\ Z_2 &= \sqrt{-2\ln U_1}\sin(2\pi U_2). \end{aligned}$$

Write a function `rnorm2()`*, to generate* `n` *observations following $N($*`mu`*,*`sigma`*$)$, where* `mu` *$\in \mathbb{R}$ (0 by default) and* `sigma` *> 0 (defaults to 1). Note that* `n` *doesn't have to be even.*

Hint: If $Z \sim N(0,1)$, then $\mu + \sigma Z \sim N(\mu, \sigma)$.

Exercise: *Write a function* `approxinvert()` *with the following arguments:*

- *a vectorized continuous and strictly monotone function* `f` *: [*`a`*,* `b`*$]^n \to \mathbb{R}^n$;*
- *a numeric vector with elements in [*`f`*(*`a`*),* `f`*(*`b`*)];*
- *a real value* `a`*;*
- *a real value* `b` *>* `a`*;*
- *a positive integer* `k` *> 2 (100 by default).*

The function should calculate the approximation of `f`*$^{-1}$(*`y`*) by using some kind of interpolation (e.g. a linear one) of* `f` *in k equidistributed points in [*`a`*,* `b`*].*

Hint: You can use `approxfun()` *or* `splinefun()`*.*

2.6 Summary

R is a functional language – functions play a key role. Functions are used to wrap a chunk of R code for later, abstract use (on any data).

R packages on CRAN are perfect ways to extend your toolkit.

2.7 Bibliography

- R Core Team, *An introduction to R*, 2014, Sec. 10
- R Core Team, *R language definition*, 2014, Sec. 4
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 5

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

3 Unit testing. Debugging. Exception handling

3.1 Introduction

R is a functional language – functions play a key role. A function should implements one, well-defined task and give correct output for all elements in its domain. A "big task" is implemented by combining smaller code units (i.e. functions), thus we have to make sure that all the "building blocks" do what they are supposed to do to. If it is not the case, we will learn how to locate bugs in a function's code. Moreover, we will discuss how to deal with exceptions raised by other functions.

3.2 Unit tests

Let us imagine that we have a set of functions: `A()` calls `B()` which uses `C()` that calls `D()` in `sapply()`. It turns out that a user gets some unexpected results while calling `A()` with `n<0` argument. Such an error is hard to locate. It is `D()`'s developer that thought: "Nobody will call my function with `n<0`." He made the life of other programmers difficult, because he didn't assure a proper behavior of his/her function in such "boundary" cases. If `D()` was developed with appropriate care, it would echo:

```
## Error:  Error in D: non-negative value of n expected.
```

Unit tests assure us that functions always handle input data correctly. A unit test is a strict, written *contract* that a function must satisfy.

Unit tests may play a role of a function's *behavioral specification*:

- Given (A, B, C) on input, I expect to get a list on output,
- Given (A, B, C) on input, I expect to get (D) on output,
- Given (A, B) on input, I expect to get a warning,
- Given (A, B, D, E) on input, I expect to get an error.

They should be performed **as often as possible**. Thus, they can't be computationally demanding. This enables us to implement the idea of so-called **continuous testing**, which may lead to better code quality.

Unit tests should cover as many test cases as possible. Thanks to them:

- We find problems early.
- We don't recreate bugs that are already fixed.
- We gain increased confidence in our code.

However, software testing is an *art*. Even though there are some rules, we basically use our intuition. Ideally: a programmer should not be testing his/her own functions. Another person should do that. There are at least two approaches. In black box testing, a tester does not at the source code – he/she just state what is the function's *expected* behavior. In white box testing, a tester studies the source code to find as many flaws as possible.

It is always advised to write a bunch of unit tests **before** writing implementing a function. First you specify what a function should do (for many possible input data cases), then you start writing code that tries to fulfill the expectations.

Imagine that we would like to write a function to compute the n-th Fibonacci number.

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2), \quad n \geq 2 \end{aligned}$$

We have: $(1, 1, 2, 3, 5, 8, 13, 21, \dots)$

The key question is: What to test for here?

Exercise: *What do you expect from a function to compute the n-th Fibonacci number?*

In R, a function accepts "anything" on input. It may also output "anything". It is very important to assure that a routine does not give improper results for objects out of the domain of interest.

Moreover, we should assure that a function has R's typical look-and-feel. For instance we should take care of:

- vectorization,
- recycling rule,
- `NA`, `Inf`, `NaN` handling,
- `0`s, negative values, empty vectors,
- FP arithmetic accuracy issues,
- preservation of input object's attributes.

Generally, the `stopifnot()` function is your friend. Use it **within functions** for stating assertions (conditions which should always be true), and verifying if you got expected input data.

```
fib <- function(x) {
    stopifnot(is.numeric(x), is.finite(x))
    # ...
}

fib(mean)  # a proper behavior
## Error:  default method not implemented for type 'closure'
fib("4")  # OK
## Error:  is.numeric(x) is not TRUE
```

However, even though `stopifnot()` may be used to write unit tests, we have better tools for that. Among popular unit testing solutions for R we find the test that package[3].

It is advised to use separate source files for specifying test cases. For example:

```
library(testthat)  # test-A.R
test_that("A", {
    # expectations...
})
```

and

```
library(testthat)  # test-B.R:
test_that("B", {
    # expectations...
})
```

All the tests should be performed as often as possible:

```
test_dir("path")  # see also: autotest()
```

Here are the functions to define the expectations: `expect_true(x)`, `expect_false(x)`, `expect_is(x, class)`, `expect_equal(x, expected)`, `expect_equivalent(x, expected)`, `expect_identical(x, expected)`, `expect_output(x, regexp)`, `expect_message(x)`, `expect_warning(x)`, `expect_error(x)`, etc. For usage and examples, refer to Wickham's paper.

Exercise: *Write some Fibonacci testthat tests.*

3.3 Debugging

Unit tests are used to verify if a function does what it's supposed to. But what if a function fails some tests and we cannot find out why? Debugging, just like testing is also an art – there are no solutions that work in every case. Basically it is advised to:

- peer-review your code,

[3] H. Wickham, testthat: Get Started with Testing, *The R Journal* 3(1), 2011, pp. 5-10.

HUMAN CAPITAL NATIONAL COHESION STRATEGY
EUROPEAN UNION EUROPEAN SOCIAL FUND
Project co-financed by the European Union under the European Social Fund

- use paper&pencil,
- formulate assertions (`stopifnot()`),
- use RStudio's visual debugger.

Sorry to say that, but **all bugs are your fault**.

3.4 Exception raising and handling

The R condition system provides a mechanism for signaling and handling unusual conditions. There are thee types of conditions:

- messages
- warnings
- errors

First of all, the `message()` function may be used to print out a *diagnostic message* on `stderr`.

```
message("This is a diagnostic message")
## This is a diagnostic message
```

Messages may be suppressed by calling the `suppressMessages()` function.

Secondly, warnings may be used to attract our attention to a potential issue.

```
warning("This is a warning.")
## Warning:  This is a warning.
sqrt(-1)
## Warning:  NaNs produced
## [1] NaN
1:2 + 1:3
## Warning:  longer object length is not a multiple of shorter object length
## [1] 2 4 4
```

Warnings may be suppressed by calling the `suppressWarnings()` function.

```
suppressWarnings(sqrt(-1))
## [1] NaN
```

Use this function only if you are completely sure what you are doing. Otherwise, it is good if a user is aware that something does not run as smoothly as it is supposed to.

Note that all warnings may be turned into errors. This is a useful behavior while performing software testing:

```
options(warn = 2)
sqrt(-1)
```

```
## Error:  in sqrt(-1) :  (converted from warning) NaNs produced
```

Warnings may also be turned off completely, but doing so is not recommended.

Also note that some situations may issue a warning depending on current R *options*. For example, the `check.bounds` global option (see `?options`) controls if R should warn about a vector's extension.

```
options(check.bounds = TRUE)
x <- 1:5
x[10] <- 10
## Warning:  assignment outside vector/list limits (extending from 5 to 10)
```

Some other global options useful when testing software: `warnPartialMatchArgs`, `warnPartialMatchAttr`, `warnPartialMatchDollar`.

Errors are thrown with the `stop()` function.

```
stop("This is an error.")
## Error:  This is an error.
```

```
unknown_function(unknown_object)
## Error:  could not find function "unknown_function"
cat(geterrmessage())
## Error in eval(expr, envir, enclos) :
##   could not find function "unknown_function"
```

We may enqueue some expressions to be performed at the end of a function's call with the `on.exit()` function. They will be evaluated no matter if an error occurred within a function or not.

```
test <- function() {
    on.exit(print("C"))
    on.exit(print("D"), add = TRUE)
    print("A")
    stop("an error occurred")
    print("B")
}

test()
## [1] "A"
## Error:  an error occurred
## [1] "C"
## [1] "D"
```

The `on.error()` function is useful if we wish to make sure that some resources are restored to their previous state. Typical scenarios include:

- Closing an opened file or database connection automatically
- Restoring graphical settings. (see `print(boxplot.default)`)
- Restoring global options, locales, etc.

The `tryCatch()` function allows to catch any R error and react accordingly. Its syntax is as follows:

```
tryCatch(expression_to_try, error = error_handling_function_1arg, finally = expression_to_eval_at_the_end)
```

`tryCatch()` is useful e.g. in lengthy simulations: What if the 100000th iteration fails? Without exception handling we could loose all the results obtained so far...

An example:

```
test <- function(x) {
    tryCatch({
        sum(as.numeric(x))
    }, error = function(e) {
        NA
    })
}

test(1:5)
## [1] 15
test(test)
## [1] NA
```

A `finally` block is also possible:

```
test <- function(err) {
   tryCatch({
      if (err) stop("error")
      cat("good morning;")
   },
   error=function(e) {
      cat("an error occurred;")
   },
   finally={
      cat("this is the end;")
   })
```

```
    cat("goodbye\n")
}

test(FALSE)
## good morning;this is the end;goodbye
test(TRUE)
## an error occurred;this is the end;goodbye
```

3.5 Summary

Before writing any function, start with a bunch of unit tests.

The RStudio debugger gives a convenient way to debug your code. If you fail to detect a bug, don't hesitate to ask your colleague!

Errors may be caught with `tryCatch()`.

3.6 Bibliography

- R Core Team, *R language definition*, 2014, Sec. 8, 9
- R Core Team, Writing R extensions, 2014, Sec. 3
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 19
- Wickham H., testthat: Get Started with Testing, *The R Journal* 3(1), 2011, pp. 5-10
- *RStudio documentation*: support.rstudio.com/hc/en-us/articles/200713843-Debugging-with-RStudio
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 13