Introduction to R

Krzysztof Gajowniczek, PhD
Intelligent Systems Group
Department of Artificial Intelligence
Institute of Information Technology

Academic year: 2020/2021

Contents

1 Course outline

The students' theoretical knowledge of data analysis, machine learning, and other computational methods often does not go hand-in-hand with their abilities to implement such algorithms on their own.

The main aim of this very course is to fill this gap, so that you will have necessary skills to develop high quality software for your own scientific or any other purposes, but also to share it within the user community.

Most "statistics/data analysis/machine learning in R" courses won't teach you how to program in R. They may instruct you how to use R to apply some built-in methods. You might have written some R functions before. You might be able to cleanse/transform/import/export a data set. But most probably your R knowledge is a mess. Thus, I do not assume that you know R at all.

You will be struck by how much this language is easy to learn. You can become an intermediate-level R programmer very fast, even if you did not program computers at all before. We will of course go much deeper than that. And if you know C/C#/Java, you'll be astonished with the beauty of R – getting rid of your C-like habits will be challenging.

By completing the course, you should be able to:

- understand some general, advanced programming concepts,
- analyze a problem and determine how to represent it with R language elements, which algorithms and data structure to use in order to obtain the most effective solution,
- automatize and optimize data processing tasks,
- write complex R applications, especially by composing them from simpler parts (if they are available),

- properly and efficiently implement data analysis, machine learning, or any other in-memory computational methods,
- debug, test, benchmark, and profile your code.

In other words, you will become an advanced R user (You will get a complete understanding of how R works) and an advanced R developer/programmer (You will know how to write high quality, reliable, maintainable, and fast data analysis software that can be shared within the open source community).

2 Introduction to R

2.1 What is R?

According to {R-project.org} R is...

 a language and environment for statistical computing and graphics.

R was initially written by **R**oss Ihaka and **R**obert Gentleman. and now is maintained and developed by the R Core Team. Development of R was inspired by the commercial S language (John Chambers et al.), which *has forever altered the way people analyze, visualize, and manipulate data.* R is de-facto standard language for data analysis, data/text mining, machine learning, etc. It includes virtually any data manipulation procedure, statistical model, and chart that a modern data scientist could ever need. It allows you to create reproducible data analyses faster than with the use of other statistical software. New techniques can be saved, reused, and shared with others. Among notable users of R we find Microsoft, Facebook, Google, Mozilla, New York Times, Twitter, see {revolutionanalytics.com/companies-using-r} for some case studies.

Because of that, I propose to redefine the above definition slightly. R is:

 [One of the most popular and powerful] [programming] language and environment for statistical computing, [data analysis, data mining], and graphics.

We call R an **environment**, because it consists of the following "building blocks".

- **A programming language**:
  - general-purpose,
  - high-level, but C-like syntax,
  - functional (Scheme-inspired),
  - having some OOP facilities,
  - interpreted (not compiled),
- **Run-time environment**:
  - graphics,
  - debugger,
  - garbage collector,
  - access to system functions.
- **Contributed extension (add-on) packages**:
  - CRAN (The Comprehensive R Archive Network),
  - BioConductor,
  - R-Forge,
  - GitHub.
- **Documentation.**

R is an open source GNU project which is licensed under the GNU GPL v2 (it is free of charge). However, it can be (and is) used for commercial purposes. (See: R FAQ, Sec. 2.11). Notably, R's "core" is written

in the C programming language and most of user-visible functions are written in R itself. However, there is an interface to procedures in C/C++/Fortran libraries (we'll play with C++ soon) and we are able to communicate with Java (so we may use R in e.g. connection with the Hadoop platform), Python, Julia, SAS.

We should also know about some of R's limitation. Out-of-the-box R performs single-threaded computations (STC) on in-memory data (IND). Luckily, there exist solutions to overcome these restrictions. For example, there is a commercial version of R called Revolution R Enterprise – a complete system for big data analysis. Moreover, a number of contributed R extension packages grouped within the CRAN Task View: {High-Performance and Parallel Computing with R} addresses more demanding computational problems like:

- **Parallel computing**:
  - implicit/explicit parallelism,
  - grid computing,
  - Hadoop,
  - GPUs.
- **Large memory and out-of-memory data (OMD)**:
  - OMD algorithms,
  - OMD data structures,
  - Database access: MySQL/MariaDB, Oracle, PostgreSQL, SQLite, etc.

Unfortunately, because of time and space limits, the above won't be covered during the course of this course. On the other hand, we will be comprehensive in STC-IMD, which makes a best start for your own further studies.

2.3 Installing R

Please install the most recent version of R (> 3.1) on your machine. It can be downloaded from {R-project.org}. Notably, R runs on: Windows, OS X, and UNIX/Linux.

We are interested in developing advanced data analysis software, so a couple of additional tools will be required. First of all, make sure you have {GCC} or {clang} C/C++ compiler suite installed (other ones like {Oracle Solaris Studio} or {ICC} may be problematic). Note that a quite recent g++ is required for C++11, i.e. v. > 4.7. Windows users should download the {Rtools} package and OS X fans should get {Xcode}.

Moreover, we will need git for source code management (git for Windows and OS X may be downloaded from {git-scm.com}). Also, a *LATEX* distribution is recommended for building R packages.

2.4 Bibliography

- *The R manuals* by R Core Team, {cran.r-project.org/manuals.html}.
- Chambers J.M., *Programming with data*, Springer, 1998.
- Chambers J.M., *Software for data analysis.* Programming with R, Springer, 2008.
- Venables W.N., Ripley B.D., *S programming*, Springer, 2000.
- Eddelbuettel D., *Seamless R and C++ integration with Rcpp.* Springer, 2013.
- Wickham H., *Advanced R*, Chapman and Hall, 2014.
- Matloff N., *The art of R programming*, No Starch Press, 2011.

3 Working in RStudio

A good programmer is a productive programmer. He/she does not think much how to write a program. He/she knows just writes a program. To make our work flow efficient, we will need some development environment. Among them we may find e.g. RStudio, an Eclipse plugin named {StatET} and an Emacs

plugin {ESS}. Personally, I am not fully satisfied with any of these. I find most convenient to work with RStudio, but it is still quite distant from my ideal. Anyway, at least it is great for learning purposes.

3.1 RStudio basics

RStudio (see {www.rstudio.com/products/rstudio/}) is an Integrated Development Environment (IDE) for `R`. I suggest using the latest preview release or this program, which is available at {www.rstudio.com/products/-rstudio/download/preview/}. Please update this IDE as often as possible, as new features and fixes to old bugs appear quite often. Of course, `R` must be installed first.

Here is a list of the most important RStudio features. This IDE provides:

1. A built-in `R` console,
2. `R`, LaTeX, HTML, C++ syntax highlighting,`R` code completion,
3. Easy management of multiple projects,
4. Integrated `R` documentation,
5. Interactive debugger,
6. Package development tools.

Its typical uses include:

1. Playing with `R` (writing `R` code, performing data analyses, etc.),
2. Creating reproducible reports and slides.
3. Writing `R` packages (also in C/C++).

Before using RStudio, spend some time setting up some options (Tools → Global Options). In particular, the Code Editing options that I use are depicted in Fig. 1.

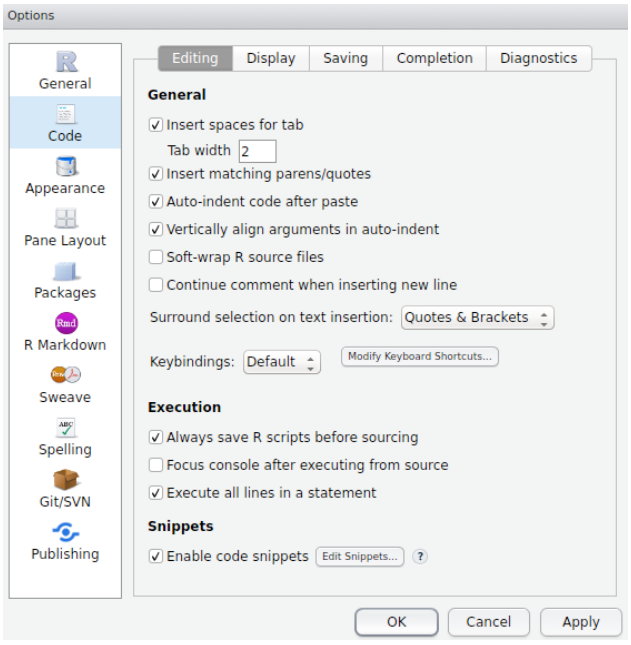


Figure 1: Recommended RStudio code editing options.

3.2 `R` in the interactive mode

Type in the `R` console (see Fig. 2):

```
> 2 + 2 # ENTER
> R.version.string # ENTER
```

In the **Interactive mode** results come immediately. Here, R works in the so-called read-eval-print loop.

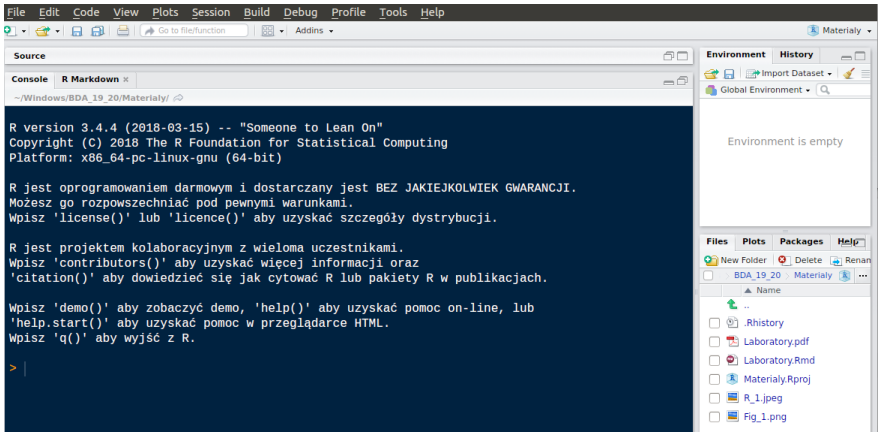


Figure 2: RStudio is ready for use.

You should have obtained the following results:

```
2 + 2 # ENTER
```

```
## [1] 4
```

```
R.version.string # ENTER
```

```
## [1] "R version 3.6.3 (2020-02-29)"
```

You may use <↑>, <↓> to navigate through the history of recently used commands. Try it.

There are two types of prompt characters it the R console. Normally, ">" means that R is ready for new input. On the other hand, "+" appears when R expects us to continue input (expression entered so far is not yet ready for eval). For example, let's type 1+2*3 in the following way:

```
> 1+
+ 2*
+ 3
```

Also note that "+" often appears when there is no matching, ", ', }, or). For example:

```
> f <- function(x) {
+   if (all(test %in% x))
+     x <- paste(x)
+   x
+ }
+
```

When you are sure that you have made a syntax error, press ESC to suppress the command being entered.

3.3 R in the "mixed" mode

The interactive mode is most often unproductive: most often, we will write R scripts and send their fragments to the R console. To create a new R script, click File → New file → R script (or press <CTRL+SHIFT+n>), see also Fig. 3.

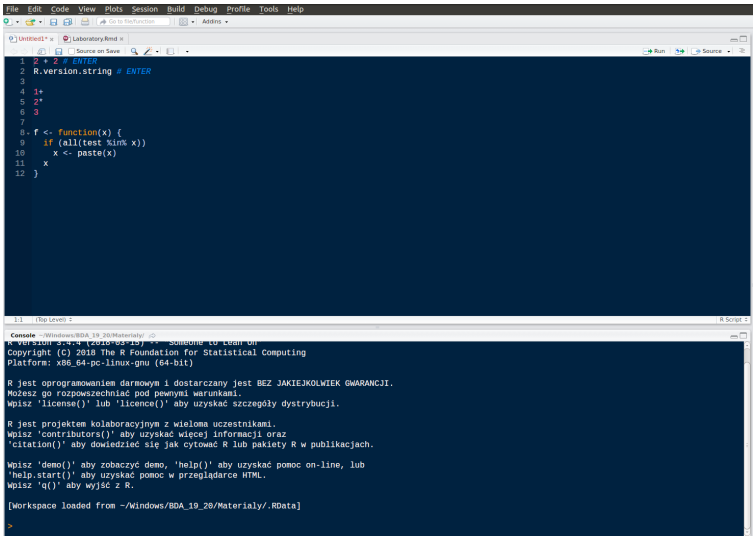


Figure 3: Editing an R source script in RStudio.

For maximum portability, when saving your script, always use the UTF-8 encoding (File → Save with encoding, see also Tools → Global options → General → Default text encoding; we will discuss Unicode-related issues later on).

It is advisable to document your code: "#" starts a comment – all characters until the end of line will be ignored by the interpreter.

```
# my code:
2 + 2 # two plus two
```

```
## [1] 4
```

```
"test" # another comment
```

```
## [1] "test"
```

Comment/uncomment keyboard shortcut: <CTRL+SHIFT+c>.

Typically, we will use one of the following possible work-flows in RStudio:

- Write functions in an R script, test/use in the console.
- Write functions in an R script, test/use in the same script (then some code may be commented out).
- Write functions in an R script, test/use in another one.

Useful keyboard shortcuts: <CTRL+ENTER> – run current line/text selection in the console, <CTRL+SHIFT+s> – run (source) whole script.

By the way, RStudio is quite "safe". If you forget save your script, it will still be available when you restart the application. R session may be stored for later use. (save workspace image → .RData file). Moreover, the history of recently used commands may be saved. (→ .Rhistory file).

Additionally, you might wish to consider organizing your workflow into **projects**, as you will surely work on many tasks at the same time. In order to create a new project, click File → New project. Interestingly, we will discuss revision control systems (git) later on – it is a very good way to manage changes made in the project files and to work on some code with others.

3.4 R in the batch mode

UNIX-like systems users (this includes Linux and OS X) might be also interested in the fact that R can also be run in the **batch mode**. For example, let us create a file /test.R with the following contents:

```
#!/usr/bin/Rscript --vanilla
cat("2 + 2 = ", 2+2, "\n")
```

If Rscript is not in /usr/bin, run whereis -b Rscript in the terminal.

If you equip this file with execute permissions (chmod u+x /test.R), you will be able to run it directly in the terminal:

```
[krzysiek@krzysiek ]$ ./test.R
2 + 2 = 4
```

Also, it is also possible to run this script via R CMD BATCH:

```
[krzysiek@krzysiek ]$ R CMD BATCH --vanilla test.R
2 + 2 = 4
```

3.5 Documentation

R comes with an extensive documentation system. To get help on a specific topic (e.g. a function name), type one of the following:

```
?rep
?"rep" # for special characters, e.g. ?"=="
help("rep")
```

See Table 1 for a brief overview of a man page structure.

To perform a fuzzy search (within locally installed packages) for a specific topic, type:

```
??replicate
help.search("replicate")
```

You may also search within all the CRAN packages by calling:

```
install.packages("sos") # install a package (once)
library("sos")
findFn("replicate")
```

If you need more help on an R feature, there are also some search engines on the web, for example {rdocumentation.org} and {rseek.org}.

Table 1: Excerpts from the ?rep man page.

R Documentation
rep {base} (*The rep function from package base.*)
Description (*What does this function do?*)
Usage (*How is it defined?*)
Arguments (*Meaning of each function's argument.*)
Details (*Technical details.*)
Value (*Description of return value.*)
References (*Bibliography.*)
See Also (*Links to similar functions.*)
Examples (*Exemplary R code & exercises.*)

3.6 Keyboard shortcuts

Many tasks can be done much more effectively without the mouse – You should use your keyboard as of ten as possible. An extensive list of RStudio Keyboard Shortcuts is available at {support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts}. Below we list the most interesting and useful ones.

- <↑>, <↓>,<←>, <→> - moving the caret
- <HOME>, <END> – beginning/end of line
- <CTRL+←>, <CTRL+→> – word boundaries
- <PAGE UP>, <PAGE DOWN> – one "screen" up/down

You surely use the clipboard regularly, so let us just recall that: <CTRL+C> – copy, <CTRL+V> – paste, <CTRL+X> – cut. But how to select a piece of text without the mouse?

- <SHIFT+↑>, <SHIFT+↓>, <SHIFT+←>, <SHIFT+→>
- <SHIFT+CTRL+←>, <SHIFT+CTRL+→> – whole words
- <CTRL+a> – whole source
- <SHIFT+END>, <SHIFT+HOME>
- <SHIFT+CTRL+END>, <SHIFT+CTRL+HOME>, etc.

Name completion in the console/source editor: <TAB> or <CTRL+Space>, see Fig. 4.

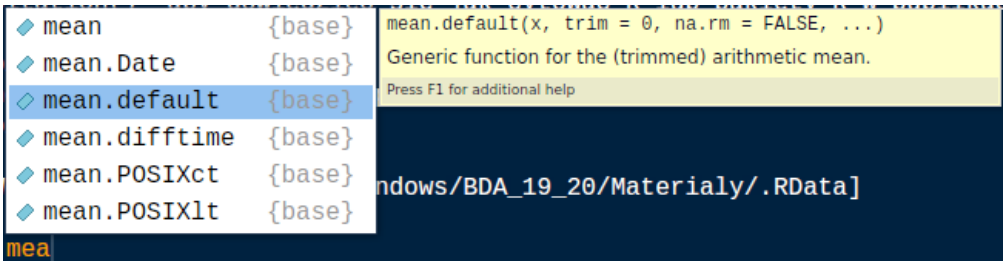


Figure 4: Name completion in RStudio.

To find (and then possibly replace) a piece of text, press <CTRL+f>. Then <CTRL+g>, <CTRL+SHIFT+g> may be used to proceed to the next or previous match. By the way, we will discuss regular expressions later during this course – these are a extremely powerful tool for text searching.

RStudio's main window is split into 4 subwindows. <CTRL+1> moves focus to the source editor. Then <CTRL+PgUp>, <CTRL+PgDown> may be used to go to the next/previous tab.

<CTRL+2> moves focus to the console and <CTRL+3> moves focus to the help pane. In the help pane <↑>, <↓>, etc. can be used to navigate through the man page. Press <CTRL+f> to find a piece of text in current topic, hTABi to go to the next link, and <ENTER> to move to a linked man page.

We already mentioned <CTRL+SHIFT+c>, which comments/uncomments a chunk of code. What is more, in the source editor, <TAB> may be used indent line, <SHIFT+TAB> to unindent it, and <CTRL+i> to reindent lines automatically. You will use them often when writing e.g. `R` functions.

3.7 Summary

Golden rule: You should know the tools you use. They are made to serve you and make your work more productive. So do not waste your own time. Use the keyboard. Spend some time playing with RStudio until you feel convenient with it.

4 Atomic types in R

4.1 R data types

Fig. 5 presents one of the possible ways to classify R data types.

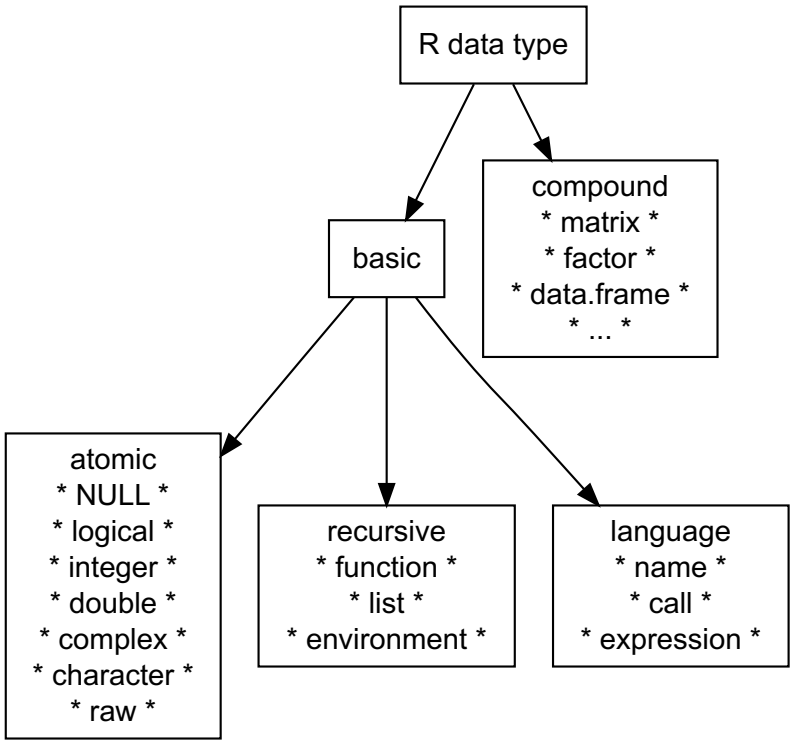


Figure 5: A classification of R data types.

Basic types are "fundamental" building blocks for our programming-with-data tasks. On the other hand, **compound types** base on basic types (w.r.t. in-memory representation) but may exhibit a much different behavior (when compared to their "basic" counterparts). For example, a *data frame* is a specific kind of a list, a *matrix* is a special atomic vector, and a *factor* is represented by an integer vector.

Let `x` be an object. There are many methods to determine `x`'s type.

```
typeof(x) # (R internal) type of x
mode(x) # mode
class(x) # class (may freely be changed by the user)
```

The `typeof()` function determines how `x` is represented in computer's memory. It returns the *basic* type also for *compound* objects. `mode()` is mostly used for compatibility with the S language. Its return value may be different from the one provided by `typeof()`. On the other hand, `class()` gives the most abstract type information. We will discuss it later on.

4.2 Atomic vectors

Atomic vectors can be thought of as contiguous cells containing data of the same kind. They're much like finite sequences/vectors in mathematics, e.g. $\mathbf{x} = (x_1, ..., x_n) \in \mathbb{R}^n$. Table 2[1] gives the types of atomic vectors available in R.

[1]Note that special values like `NA`, `NaN`, `Inf` will be covered later on.

Table 2: Atomic vectors in R.

Basic type	Elements
logical	$(\mathbb{B}) = \{$TRUE, FALSE$\}$
integer	$(\mathbb{Z}) = \{-2147483647, -2147483646, ..., 2147483647\}$
double	$(\mathbb{R}) \subseteq [-1.8 \times 10^{308}, 1.8 \times 10^{308}]$
complex	$(\mathbb{C}) \subseteq [-1.8 \times 10^{308}, 1.8 \times 10^{308}]^2$
character	character strings, e.g. "string1", 'string2'
raw	bytes, $\{00, \ldots, ff\}$

4.2.1 Logical vectors

There are two logical constants: TRUE and FALSE. Both of them are R's reserved keywords.

```
TRUE
```

```
## [1] TRUE
```

```
typeof(TRUE)
```

```
## [1] "logical"
```

Note that R is a case-sensitive language. "false" is not the same as "FALSE":

```
false # wrong
```

```
## Error in eval(expr, envir, enclos): nie znaleziono obiektu 'false'
```

```
TypeOf # wrong
```

```
## Error in eval(expr, envir, enclos): nie znaleziono obiektu 'TypeOf'
```

```
FALSE # OK
```

```
## [1] FALSE
```

R has no scalar types. "TRUE" denotes in fact a logical vector of length 1

```
length(TRUE) # vector's length
```

```
## [1] 1
```

This will highly affect our the programming style.

Let us proceed with some functions to create a logical vector. Firsty, c() combines a given sequence of values and returns a vector:

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

```
## [1]  TRUE FALSE  TRUE
```

```
c(FALSE) # the same as: FALSE
```

```
## [1] FALSE
```

```
length(c(TRUE, FALSE, TRUE))
```

```
## [1] 3
```

The rep() function may be used to replicate a given vector a number of times.

```
rep(FALSE, 4)
```

```
## [1] FALSE FALSE FALSE FALSE
```

```
rep(c(TRUE, FALSE), 2) # ``recycling''-like
```

```
## [1]  TRUE FALSE  TRUE FALSE
```

In the `rep()` function man page we find that its first two arguments are called `"x"` and `"times"`. Interestingly, the following calls are equivalent:

```
rep(c(TRUE, FALSE), 4)
rep(x = c(TRUE, FALSE), times = 4)
rep(times = 4, x = c(TRUE, FALSE))
rep(c(TRUE, FALSE), times = 4)
rep(times = 4, c(TRUE, FALSE))
```

In the first case we have a simple positional matching of formal arguments to the supplied ones. The 2 following cases we have an exact matching on tags. In the last 2 calls `R` does an exact matching on tags first (`"times"`), and then uses a positional matching to set up `"x"`.

What is more, `R` may also perform *partial matching* of argument names:

```
rep(c(TRUE, FALSE), t = 4)
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

This is convenient for `R` users, but we discourage relying on this feature when writing `R` software.

Apart from the `"times"` argument, `"length.out"` or `"each"` may also be provided:

```
rep(c(TRUE, FALSE), length.out = 5)
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE
```

```
rep(c(TRUE, FALSE), each = 3)
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

```
rep(c(TRUE, FALSE), times = 3) # compare results
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

Note the difference in results. This is a little bit tricky: `"each"` is not mutually exclusive with `"times"` and `"length.out"` (otherwise the `R` authors would just provide us with 3 separate functions):

```
rep(c(TRUE, FALSE), times = 3, each = 2)
```

```
##  [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
```

```
rep(c(TRUE, FALSE), length.out = 8, each = 2)
```

```
## [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
```

4.2.2 Numeric vectors

Let us move on to numeric vectors. They are represented by the `integer` and `double`[2] basic types. Even though a typical `R` user does not distinguish between these two types, we sometimes will.

Again, recall that there are no scalars in `R`. Inputting a numeric constant in fact creates a vector of length 1:

[2] Cf. int and double types in C/C++

```
length(1)
```

```
## [1] 1
```

All the "standard" numeric constants are of `double` type.

```
typeof(1)
```

```
## [1] "double"
```

```
typeof(-3.14)
```

```
## [1] "double"
```

```
typeof(1.2e-16) # 1.2*10^(-16) == almost zero
```

```
## [1] "double"
```

This is because `R` is mostly used for statistical and scientific computing.

To input an integer constant, we use the `"L"` suffix.

```
typeof(1L)
```

```
## [1] "integer"
```

Interestingly, the `modes` (user-friendly versions of `typeof()`) of integer and double vectors are identical:

```
mode(1)
```

```
## [1] "numeric"
```

```
mode(1L)
```

```
## [1] "numeric"
```

This makes sense: both vectors store some numbers.

A bunch of technical details: `integers` are processed by the processor's arithmetic-logic unit (ALU).

```
.Machine$integer.max # smallest integer
```

```
## [1] 2147483647
```

```
.Machine$integer.max # largest integer
```

```
## [1] 2147483647
```

On the other hand, doubles are processed by CPU's floating-point unit (FPU).

```
.Machine$double.xmin # smallest positive double
```

```
## [1] 2.225074e-308
```

```
.Machine$double.xmax # largest finite double
```

```
## [1] 1.797693e+308
```

All the `integers` may be represented in the `double` type. Theoretically, ALU-based ops are faster. This is why later on when writing C++ and `R` code (for increased performance) we will pay special attention to use the `int` type for performing calculations on `integers` (for example when performing `array` indexing). In the `R` layer, however this may not be the case, e.g. `R` does integer overflow checking, which slows down computations, but of course makes them more reliable when it comes to scientific computing. Note the following:

```
.Machine$integer.max + 1L
```

```
## Warning in .Machine$integer.max + 1L: wyprodukowano wartości NA na skutek
## przepełnienia zakresu liczb całkowitych
```

```
## [1] NA
```

On the other hand, keep in mind general floating-point arithmetic's limitations (common to all computer programming languages):

```
1e+34 + 1e-34 - 1e+34 - 1e-34 == 0 # big+small-big-small
```

```
## [1] FALSE
```

```
0.1 + 0.1 + 0.1 == 0.3
```

```
## [1] FALSE
```

```
print(0.1 + 0.1 + 0.1, digits = 22)
```

```
## [1] 0.3000000000000000444089
```

```
print(0.3, digits = 22)
```

```
## [1] 0.2999999999999999888978
```

Moreover, we most often[3] do not test for $f(x) = c$ but for $|f(x) - c| \leq \varepsilon$, e.g. with $\varepsilon = 1e - 8$.

```
sin(pi) == 0
```

```
## [1] FALSE
```

```
abs(sin(pi)) <= 1e-08
```

```
## [1] TRUE
```

```
identical(all.equal(sin(pi), 0), TRUE) # more or less the same
```

```
## [1] TRUE
```

Such topics are covered in courses on basic numerical analysis, and I assume that you are perfectly aware of these. If not, see e.g. What Every Computer Scientist Should Know About Floating-Point Arithmetic? by D. Goldberg, 1991 for a good introduction to this key issue. Those of you who need to perform more accurate computations (at the cost of greater CPU usage), will certainly be interested in `R` packages like `gmp` (multiple precision integers/rationals) or `Rmpfr` (multiple precision floating-point numbers).

Let us get back to the discussion on how to deal with numeric vectors in `R`. The `c()` and `rep()` functions work as expected:

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

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

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

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

To generate regular sequences we may use the ":" operator:

```
1:5 # step = 1
```

```
## [1] 1 2 3 4 5
```

[3] Except when we only use `integer` arithmetic, which is always exact.

```
-5:-10 # step = -1; not the same as -(5:-10)
```

```
## [1]  -5  -6  -7  -8  -9 -10
```

```
1:2.5 # # step = 1 (till 2 <= 2.5)
```

```
## [1] 1 2
```

Interestingly:

```
typeof(1:2.5)
```

```
## [1] "integer"
```

```
typeof(1.1:2.5)
```

```
## [1] "double"
```

More generally, the `seq()` function can be used for generation of **seq**uences with arbitrary step.

```
seq(1,10, 2) # step=2, cf. ?seq
```

```
## [1] 1 3 5 7 9
```

```
seq(0,1 , length.out = 6)
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

4.2.3 Complex vectors

`Complex` vectors are not of our interest during this course. Just note how to input a complex number and the fact that each `complex` number is internally represented by two doubles.

```
1+2i # ``i'' suffix -> imaginary part
```

```
## [1] 1+2i
```

```
typeof(1+2i)
```

```
## [1] "complex"
```

4.2.4 Character vectors

There are two ways to input a single character string `"using double quotes"` and `'using apostrophes'`. Although these two forms are almost equivalent, the first notation will probably be preferred by the C/C++ programmers.

Once again, a "string" is in fact stored in a vector of length 1.

```
"test"
```

```
## [1] "test"
```

```
length("test")
```

```
## [1] 1
```

```
typeof("test")
```

```
## [1] "character"
```

In R there is no built-in type representing a "single code point" (character, byte). So a character vector is in fact a vector of sequences of code points. Thus, this is quite a complex basic type.

Some operations:

```
rep(c("a", "b"), 3) # rep(), c() work as usual
```

```
## [1] "a" "b" "a" "b" "a" "b"
```

```
length(c("R", "programming")) # two strings
```

```
## [1] 2
```

```
nchar(c("R", "programming")) # number of code points
```

```
## [1]  1 11
```

Note that some special characters may be included in a string. They are given by "control sequences" – escape characters. For example, `"\n"` denotes a newline character. When printed with `print()`, they are "escaped": their special meaning is neglected:

```
"test\nbest" # in fact: print('test\nbest')
```

```
## [1] "test\nbest"
```

To reveal their meaning, use `cat()`:

```
cat("test\nbest")
```

```
## test
## best
```

Table 3 lists the most interesting control sequences, see also `?Quotes`.

Table 3: Escape characters in R.

Escape sequence	Meaning
\n	newline
\r	carriage return
\t	tab
\b	backspace
\\	backslash
\'	ASCII apostrophe
\"	ASCII quotation mark

4.2.5 Raw vectors

The vectors of type `raw` consists of single bytes, i.e. `integer` $\in \{0, 1, ..., 255\}$. Even though such vectors are rarely used in daily R programming, it does not mean that we do not meet them in the near future.

```
as.raw(c(0, 1, 2, 254, 255))
```

```
## [1] 00 01 02 fe ff
```

```
typeof(as.raw(c(0, 1, 2, 254, 255)))
```

```
## [1] "raw"
```

Note that each byte is printed in {hexadecimal notation}.

4.2.6 Type hierarchy and coercion

In each of the atomic vectors, all elements are of the same type. An attempt to create a vector of objects of different types leads to (automatic) **type coercion**:

```
typeof(c(TRUE, 1L, 1, 1 + (0+1i), "one"))
```

```
## [1] "character"
```

```
typeof(c(TRUE, 1L, 1, 1 + (0+1i)))
```

```
## [1] "complex"
```

```
typeof(c(TRUE, 1L, 1))
```

```
## [1] "double"
```

```
typeof(c(TRUE, 1L))
```

```
## [1] "integer"
```

Note how are the types ordered from the least to the most general: `logical` $\prec$ `integer` $\prec$ `double` $\prec$ `complex` $\prec$ `character`.

When coercing from `logical` to `numeric`, `TRUE` is always converted to 1, and `FALSE` to 0. Conversely, coercion from `numeric` to `logical` for values $6 = 0$ always gives `TRUE`, and $= 0$ result in `FALSE`. To perform an explicit coercion, we call `as.type()` or `as.mode()` functions

```
as.numeric(c(TRUE, FALSE))
```

```
## [1] 1 0
```

```
as.logical(c(-2, -1, 0, 1, 2))
```

```
## [1]  TRUE  TRUE FALSE  TRUE  TRUE
```

In general, coercion from `numeric` to `character` and back again is not exactly reversible, because of round-off errors in the character representation:

```
as.double(as.character(0.1 + 0.1 + 0.1)) == 0.1 + 0.1 + 0.1
```

```
## [1] FALSE
```

Other rules/examples:

```
as.integer(c(1.5, -1.5)) # truncates fractional part
```

```
## [1]  1 -1
```

```
as.double("3.1415") # parse string
```

```
## [1] 3.1415
```

```
as.character(c(TRUE, FALSE))
```

```
## [1] "TRUE"  "FALSE"
```

```
as.logical(c("true", "false", "TRUE", "FALSE", "T", "F"))
```

```
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

The `is.type()` or `is.mode()` functions have been defined to check if a given object is of desired type:

```
is.logical(TRUE)
```

```
## [1] TRUE
```

```
is.numeric(1L)
```

```
## [1] TRUE
```

We also have some more abstract tests:

```
is.atomic(1:5) # atomic type
```

```
## [1] TRUE
```

```
is.vector(TRUE) # some vector (there are also ``generic'' ones)
```

```
## [1] TRUE
```

```
is.function(c(1, 5)) # definitely not a function
```

```
## [1] FALSE
```

We may also be interested in fast operations to create (pre-allocate) vectors of given type and length: `logical()`, `integer()`, `double()` (= `numeric()`), `complex()`, and `character()`.

```
logical(3) # also: vector('logical', 3)
```

```
## [1] FALSE FALSE FALSE
```

```
integer(3) # also: vector('integer', 3) and so on.
```

```
## [1] 0 0 0
```

```
double(3)
```

```
## [1] 0 0 0
```

```
character(3)
```

```
## [1] "" "" ""
```

Note the default values: `FALSE`, 0, *empty string*. We will investigate performance pitfalls of *not pre-allocating* vectors in some situations later on.

4.3 Missing values and other special values

Recall that `R` is also a language for data analysts. Sometimes it is useful to indicate that some values/observations 4 are missing or `N`*ot* `A`*vailable*.

```
c(TRUE, NA, FALSE)
```

```
## [1]  TRUE    NA FALSE
```

```
typeof(NA)
```

```
## [1] "logical"
```

Note that there are also `NA` constants for other types:

```
typeof(NA_integer_)
```

```
## [1] "integer"
```

```
typeof(NA_real_)
```

```
## [1] "double"
```

```
typeof(NA_complex_)
```

```
## [1] "complex"
```

```
typeof(NA_character_)
```

```
## [1] "character"
```

When printed, each missing value indicates itself as `NA`.

```
NA_character_
```

```
## [1] NA
```

Most `R` users of course use only the `NA` constant (and rely on coercion):

```
c("test", NA) # here, NA_character_ will be used
```

```
## [1] "test" NA
```

Missing values may appear e.g. when a result is unavailable:

```
as.numeric("thirty three") # R has no rule for that
```

```
## Warning: pojawiły się wartości NA na skutek przekształcenia
```

```
## [1] NA
```

A general rule is: operations applied on `NA`s will also result in `NA` (in the `Rcpp` part we will see that `NA` handling needs special care from the programmer).

Vectors of type double may also consist of `NaN`s (not a number).

```
c(sqrt(-1), 0^0)
```

```
## Warning in sqrt(-1): wyprodukowano wartości NaN
```

```
## [1] NaN   1
```

... or include positive or negative infinities (`Inf`):

```
c(1/0, Inf - Inf, log(0))
```

```
## [1]  Inf  NaN -Inf
```

There are various functions for testing occurrences of special values, e.g. `is.nan()`, `is.finite()`, `is.na()`. Also, `is.finite()` tests if a value $\notin$ {`NA`, `NaN`, `Inf`, `-Inf`}.

4.4 NULL type

Apart from atomic vectors, there is another atomic type called `NULL`. The `NULL` object (the only instance of the `NULL` type in computer's memory) is used whenever there is a need to indicate or specify that an object is absent.

```
typeof(NULL)
```

```
## [1] "NULL"
```

```
is.null(NULL)
```

```
## [1] TRUE
```

```
is.vector(NULL) # not a vector
```

```
## [1] FALSE
```

```
is.atomic(NULL) # atomic type
```

```
## [1] TRUE
```

Note that `NULL` should not be confused with a vector of zero length. It is because an atomic vector, even of zero length, includes a type information.

```
length(NULL) # coerced to an empty vector
```

```
## [1] 0
```

```
identical(NULL, c()) # no type information here
```

```
## [1] TRUE
```

```
identical(NULL, logical())
```

```
## [1] FALSE
```

Some functions use `NULL` to indicate that "no value" is returned.

```
cat("test\n") # returns NULL, invisibly
```

```
## test
```

```
is.null(cat("test\n"))
```

```
## test
## [1] TRUE
```

Also, `NULL`s are used e.g. as *empty* lists' elements, etc.

4.5 Assignment

A programming language would be useless if there was be no possibility to store objects in "computer's memory" for later (re)use. To assign (bind) a object (value) to a name (symbol) (i.e. *to create a named variable*), we use e.g. the *assignment operator*, `<-`.

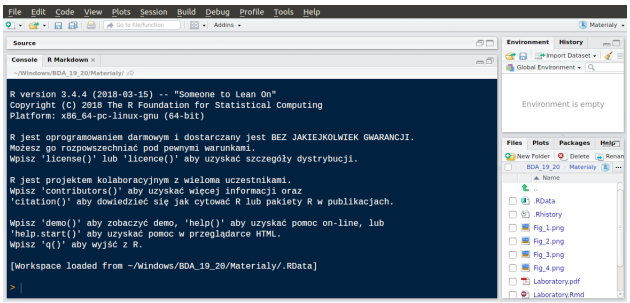
```
x <- 1:3
-x # use
```

```
## [1] -1 -2 -3
```

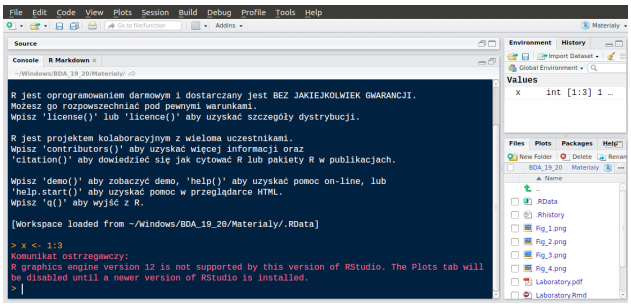
```
rep(x, 2) # use
```

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

Note the *Environment* tab in RStudio. Before assignment:



and after:



We see that, as opposed e.g. to C++, we do not declare names before their first use. Moreover, names may be changed anytime to point at an object of different type.

```
x <- TRUE # logical now
x <- 7 # and now numeric
```

There are other ways to assign:

```
name <- value
value -> name
name = value # may be ambiguous in some contexts
assign("name", value)
```

But only the first two are recommended.

By the way, `R` is a functional language: `"x <- 1:5"` is just a syntactic sugar denoting a call to:

```
"<-"(x, 1:5)
```

There are some restrictions on which names we may use. Syntactically valid names (cf. `?make.names`) consist of letters, numbers and the dot or underline characters and start with a letter or the dot not followed by a number. Names starting with the dot are hidden, e.g. they don't appear in RStudio's Environment pane. Also words reserved (cf. `?Reserved`) by `R`'s parser are forbidden: `if, else, repeat, while, function, for, in, next, break, TRUE, FALSE, NULL, Inf, NaN, NA, NA_integer_, NA_real_, NA_complex_, NA_character_`, ..., ..1, etc. Recall that `R` is a case-sensitive language.

Interestingly, `"name <- value"` returns the value invisibly. To force printing the value, use:

```
x <- 1:5 # suppress printing
(x <- 1:5) # force printing
```

```
## [1] 1 2 3 4 5
```

Thanks for that, cascading calls to `"<-"` are possible:

```
x <- y <- 1:5 # the same as x <- (y <- 1:5)
```

Note that `R` is shipped with some predefined names, e.g. `T`, `F`, `pi`, `letters` or `LETTERS`.

```
c(T, F)
```

```
## [1]  TRUE FALSE
```

Never rely on `T` and `F` in your programs: these are not logical constants.

```
T <- FALSE; F <- TRUE; c(T, F)
```

```
## [1] FALSE  TRUE
```

4.6 Summary

Things to remember:

- `R` is a case sensitive language.
- There are no scalar values (The power of `R` will be revealed when we'll discuss *vectorized* ops).
- Most common types of atomic vectors: `logical, numeric, character`.
- Missing values (`NA`s) may appear in vectors.

4.7 Bibliography

- `R` Core Team, *{An introduction to `R`}*, 2014, Sec. 2
- Goldberg D., *{What every computer scientist should know about floating-point arithmetic}*, ACM Computing Surveys 21(1), 1991, pp. 5-48.

5 Basic operations on atomic vector

C/C++ (among others) programmers are used to implement vector operations using loop-like statements. However, we will see that in `R`, for performance reasons, loops should generally be avoided. That is why we will introduce `R` loops much later on. In `R` we often rather try to solely rely on built-in vector operations – the operations presented in this section are very important. What is more, built-in operations speed up code writing – using "prefabricated" elements is easier than starting from scratch.

5.1 Operators

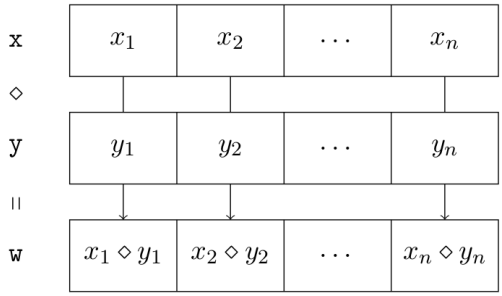
5.1.1 Arithmetic operators

Table 4 lists all the arithmetic operators available in `R`. As a general rule, by applying an arithmetic operation on two numeric vectors, in result we will get numeric vector.

Table 4: Arithmetic operators in `R`.

Operation	**Meaning**
`x + y`	addition
`x - y`	subtraction
`x * y`	multiplication
`x \ y`	division
`x ^ y`	exponentiation (also: x ** y)
`x %% y`	division remainder (modulo)
`x %/% y`	integer division

All these operations are vectorized. First of all, they are performed in an element-wise manner.



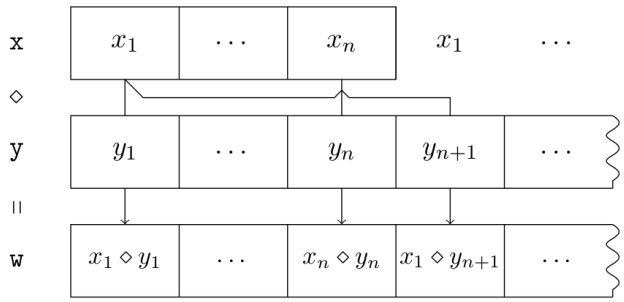

```
2 + 2
```

```
## [1] 4
```

```
c(1, 2, 3) * c(1, 10, 100)
```

```
## [1]   1  20 300
```

If the operands are of different lengths, the **recycling rule** is applied.




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

```
## [1] 2 3 4
```

```
c(1, -1) * 1:6
```

```
## [1]  1 -2  3 -4  5 -6
```

```
c(1, -1) * 1:7 # result OK, but a warning given
```

```
## Warning in c(1, -1) * 1:7: długość dłuszego obiektu nie jest wielokrotnością
## długości krótszego obiektu
```

```
## [1]  1 -2  3 -4  5 -6  7
```

Among similar operations we find the parallel minimum and maximum (often denoted in mathematics as $\wedge$ and $\vee$).

```
pmin(c(1, 2, 3, 4), c(4, 1, 2, 3))
```

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

```
pmax(c(1, 2, 3, 4), c(4, 1, 2, 3))
```

```
## [1] 4 2 3 4
```

Operations on `NA`s result in `NA`.

```
c(NA, 2, 3) + c(1, 2, NA)
```

```
## [1] NA  4 NA
```

This is sensible: "I don't know how much plus 1 equals I don't know".

Recall that special values may also come out:

```
c(1, Inf)/c(0, Inf)
```

```
## [1] Inf NaN
```

5.2.1 Logical operators

Table 5 list the logical operators. Apart from them, there is also the `xor(x, y)` function, which performs an "exclusive or" operation. All of the operators take logical vectors as arguments and result in a logical vector. C++/Java programmers should be careful: `&&` and `||` operators are also available in `R`. Later on we will see that they act on first elements of given vectors.

Table 5: Logical operators in `R`.

Operation	Meaning
`!x`	negation (*not*, unary)
`x | y`	alternative (*or*)
`x & y`	conjunction (*and*)

"Usual" rules apply:

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

```
## [1]  TRUE FALSE FALSE FALSE
```

```
c(TRUE, FALSE) | NA # Lukasiewicz's logic
```

```
## [1] TRUE   NA
```

```
!c(0, 1, 2, 0) # coercion to logical
```

```
## [1]  TRUE FALSE FALSE  TRUE
```

Here are the truth tables for `&`, `|`, and `xor()`:

```
v <- c(TRUE, FALSE, NA)
structure(outer(v, v, "&"), dimnames=rep(list(paste(v)), 2))
```

```
##        TRUE FALSE    NA
## TRUE   TRUE FALSE    NA
## FALSE FALSE FALSE FALSE
## NA       NA FALSE    NA
```

```
structure(outer(v, v, "|"), dimnames=rep(list(paste(v)), 2))
```

```
##       TRUE FALSE   NA
## TRUE  TRUE  TRUE TRUE
## FALSE TRUE FALSE   NA
## NA    TRUE    NA   NA
```

```
structure(outer(v, v, "xor"), dimnames=rep(list(paste(v)), 2))
```

```
##         TRUE FALSE NA
## TRUE   FALSE  TRUE NA
## FALSE   TRUE FALSE NA
## NA        NA    NA NA
```

5.1.3 Comparison operators

Table 6 lists the comparison operators. Given two `numeric` vectors of any type, all of them always return a `logical` vector, for example:

```
(1:5) < (5:1)
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE
```

```
rep(c(TRUE, FALSE), 3) == 0 # recycling rule + coercion
```

```
## [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

Table 6: Comparison operators in `R`.

Operation	Meaning
`x < y`	less than
`x > y`	greater than
`x <= y`	less than or equal to
`x >= y`	greater than or equal to
`x == y`	equal
`x != y`	not equal

Note that these are binary operators. Do not write `"a < b < c"` – use `"a < b & b < c"` instead. Interestingly, lexicographic order is used for comparing strings 5[4].

Read more on operators' precedence: `?Syntax`.

5.2 Selecting and modifying subsets of vectors

5.2.1 Selecting subsets of vectors

Vectors (unlike sets) are *ordered* tuples. In other words, each vector's element has its own position. For example, assume that `x` has been defined in the following way:

```
x <- c("a", "b", "c", "d", "e")
```

The index of the first element is 1.

Index	1	2	3	4	5
Value	`"a"`	`"b"`	`"c"`	`"d"`	`"e"`

To generate a subvector from a given vector, the index operator, `"["`, may be used. Its syntax is: `subsetted_vector[index_vector]`, where `index_vector` is one of:

[4]This is locale dependent, more on that later on.

- positive integer quantities,
- negative integer quantities,
- logical values,
- character strings.

If an index vector consists of positive integers, the corresponding elements are selected and concatenated, in that order, in the result.

```
(x <- 10:20) # exemplary vector
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

```
x[1] # first
```

```
## [1] 10
```

```
x[length(x)] # last
```

```
## [1] 20
```

```
x[c(1, length(x), 1)] # first, last, and first again
```

```
## [1] 10 20 10
```

```
x[1000] # no such element
```

```
## [1] NA
```

```
x[c(1.9, 2.1, 2.7)] # fractional part truncated
```

```
## [1] 10 11 11
```

An index vector consisting of negative `integers` specifies the elements to be excluded from the resulting vector.

```
(x <- 10:20) # exemplary vector
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

```
x[-1] # all but the first
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
x[c(-(1:5), -1, -5)]
```

```
## [1] 15 16 17 18 19 20
```

```
x[c(1, -1)] # don't mix positive and negative indices
```

```
## Error in x[c(1, -1)]: tylko zera mogą być mieszane z ujemnymi indeksami
```

```
x[0] # empty vector
```

```
## integer(0)
```

```
x[c(-1, 0)] # 0 ignored
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

For index vectors of `logical` type, values corresponding to `TRUE` are selected and those corresponding to `FALSE` are omitted.

```
(x <- 10:20) # exemplary vector
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

```
x[c(TRUE, rep(FALSE, 9))]
```

```
## [1] 10 20
```

```
x[c(TRUE, FALSE)] # recycling rule
```

```
## [1] 10 12 14 16 18 20
```

`Logical` index vectors implement the concept of *data filtering*.

```
(x <- sample(100:999, 10)) # random sample
```

```
##  [1] 439 183 314 238 296 431 443 947 684 972
```

```
x[x > 500] # select only elements > 500
```

```
## [1] 947 684 972
```

```
x[x >= 250 & x <= 750]
```

```
## [1] 439 314 296 431 443 684
```

For example, let us consider a simple database:

```
name <- c('John', 'Mary', 'Gerard', 'Steven')
height <- c(181,164,192,NA)
```

Here are some "queries". Note that the syntax is very readable.

```
name[height > 180]
```

```
## [1] "John"   "Gerard" NA
```

```
name[height > 180 & !is.na(height)]
```

```
## [1] "John"   "Gerard"
```

5.2.2 Modifying subsets of vectors

Index operator also works in a "replacement" mode.

```
(x <- 1:6)
```

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

```
x[1] <- 10; x
```

```
## [1] 10  2  3  4  5  6
```

```
x[x>3] <- (x[x>3])^2; x
```

```
## [1] 100   2   3  16  25  36
```

```
x[-c(1,length(x))] <- c(1, -1); x
```

```
## [1] 100   1  -1   1  -1  36
```

Its usage may lead to a vector's "size extension":

```
(x <- 1:6)
```

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

```
x[length(x)+1]
```

```
## [1] NA
```

```
x[10] <- 10; x
```

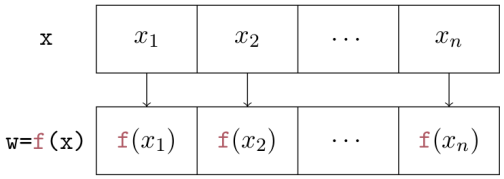
```
##  [1]  1  2  3  4  5  6 NA NA NA 10
```

Note that this is a linear-time (not: constant-time) operation. A new vector is created and all its old contents are copied. Although it is useful for "ordinary" `R` users, do not rely on such features when writing professional software (we will perform some benchmarks later on).

5.3 Built-in functions

5.3.1 Mathematical functions

All the mathematical functions discussed in this section are vectorized. Given a numeric vector of length n, they output a vector of the same length.



Among such functions we find e.g. `abs()`, `sign()`, `floor()`, `ceiling()`, `round()`, `sqrt()`, `exp()`, `log()`, `sin()`, `cos()`, etc.

Some examples:

```
abs(c(1, -1, 2, -2, 0)) # vectorized
```

```
## [1] 1 1 2 2 0
```

```
round(3.141592, 3)
```

```
## [1] 3.142
```

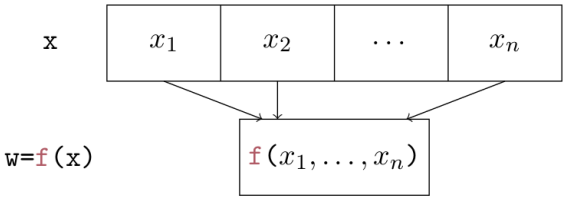
See `?round`: digits argument has default value of 0.

```
round(3.141592) #equiv. round(3.141592, 0) if `digits` omitted
```

```
## [1] 3
```

5.3.2 Aggregation functions

Aggregation functions, on the other hand, take an atomic vector on input and output one value. Among such functions we find e.g. `sum()`, `prod()`, `mean()`, `median()`, `min()`, `max()`, `var()`, `sd()`, `quantile()`, `any()`, `all()`, etc.



```r
x <- c(1, 5, 4, NA, 3)
sum(x)
```

```
## [1] NA
```

```r
sum(x[!is.na(x)]) # cf. sum(x, na.rm=TRUE)
```

```
## [1] 13
```

```r
sum(is.na(x)) # how many missing values?
```

```
## [1] 1
```

```r
any(is.na(x))
```

```
## [1] TRUE
```

```r
all(x == 1:5)
```

```
## [1] FALSE
```

5.3.3 Other functions

Cumulative sum, product, minimum, and maximum:

```r
x <- c(4, 2, 3, 5, 1)
cumsum(x)
```

```
## [1]  4  6  9 14 15
```

```r
cumprod(x)
```

```
## [1]   4   8  24 120 120
```

```r
cummin(x)
```

```
## [1] 4 2 2 2 1
```

```r
cummax(x)
```

```
## [1] 4 4 4 5 5
```

Iterated difference:

```r
diff(x) # lag=1 by default, see ?diff
```

```
## [1] -2  1  2 -4
```

which family:

```r
x <- c(40, 20, 30, 50, 20)
which(x > 3)
```

```
## [1] 1 2 3 4 5
```

```r
which.min(x) # like which(x == min(x))[1], but faster
```

```
## [1] 2
```

```r
which.max(x) # which(x == max(x))[1]
```

```
## [1] 4
```

Permutation-based operations:

```
x <- c(40, 20, 30, 50, 20)
sort(x) # stable sorting algorithm
```

```
## [1] 20 20 30 40 50
```

```
sort(x, decreasing = TRUE) # nonincreasing
```

```
## [1] 50 40 30 20 20
```

```
x[order(x)] # same as sort(x)
```

```
## [1] 20 20 30 40 50
```

```
sample(x) # random permutation
```

```
## [1] 20 30 50 40 20
```

See also: `rank()`, `is.unsorted()`, `rev()`.

Set-theoretic functions:

```
basket1 <- c("apples", "bananas", "apples")
basket2 <- c("bananas", "oranges", "cherries")
union(basket1, basket2)
```

```
## [1] "apples"   "bananas"  "oranges"  "cherries"
```

```
intersect(basket1, basket2)
```

```
## [1] "bananas"
```

```
setdiff(basket1, basket2)
```

```
## [1] "apples"
```

```
is.element("pears", basket1)
```

```
## [1] FALSE
```

```
setequal(basket1, c("bananas", "apples"))
```

```
## [1] TRUE
```

See also: `unique()`, `duplicated()`, `anyDuplicated()`.

String concatenation (join):

```
paste(c("a", "b"), 1:2)
```

```
## [1] "a 1" "b 2"
```

```
paste(c("a", "b"), 1, sep = "") # recycling rule, no separator
```

```
## [1] "a1" "b1"
```

```
paste(c("a", "b"), 1:2, sep = ":", collapse = ", ")
```

```
## [1] "a:1, b:2"
```

```
paste(c("a", "b"), c(1, NA)) # wrong NA handling :(
```

```
## [1] "a 1"  "b NA"
```

`cat()` and `format()`:

```
x <- c(1, 10, 100)
cat(x)
```

```
## 1 10 100
```

```
cat(x, sep = "\n")
```

```
## 1
## 10
## 100
```

```
cat(format(x), sep = "\n")
```

```
##   1
##  10
## 100
```

Note that string processing functions will be covered in very detail later on.

5.4 Summary

This is fascinating: You are now ready to solve a tremendous number of interesting tasks. Just remember to try not to use any `R` loops (if you already know them). In `R`, simplicity is the key to success.

5.5 Bibliography

- `R` Core Team, *{An Introduction to R}*, 2014, Sec. 2
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 2

6 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 $O(n)$.

6.1 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:

```r
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`.

Lists may contain sublists:

```r
list(1, list(2, 3))
```

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

```r
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:

```r
vector("list", 3)
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
```

We can also convert an atomic vector to a list:

```r
as.list(1:2)
```

```
## [[1]]
## [1] 1
##
```

```
## [[2]]
## [1] 2
```

6.2 Selecting and modifying subsets of lists

6.2.1 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
```

6.2.2 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 in list(1, 2, 3)[[-1]]: attempt to select more than one element in get1index <real>
```

```
list(1, 2, 3)[[c(TRUE, FALSE, FALSE)]] # only positive integers
```

```
## Error in list(1, 2, 3)[[c(TRUE, FALSE, FALSE)]]: rekursywne indeksowanie nie powiodło się na poziomi
```

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 in L[[c(2, 1, 2)]]: indeks jest poza granicami
```

6.2.3 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 in L[2:3] <- 10:15: liczba pozycji do zastąpienia nie jest
## wielokrotnością długości zamiany
```

```
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)
str(L)
```

```
## List of 3
##  $ : num 1
##  $ : NULL
##  $ : num 3
```

6.3 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.

6.3.1 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"
```

6.3.2 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
```

6.3.3 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:

```
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.

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:

```
mapply("+", 1:3, 11:13, SIMPLIFY = FALSE)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] 14
##
## [[3]]
## [1] 16
```

```
mapply("*", list(1:3, 4:6), list(10, -1), SIMPLIFY = FALSE)
```

```
## [[1]]
## [1] 10 20 30
##
## [[2]]
## [1] -4 -5 -6
```

```
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
```

6.4 Summary

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

6.5 Bibliography

- R Core Team, *{An introduction to R}*, 2014, Sec. 6.1, 6.2
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 4

7 Functions

7.1 Introduction to functions

The functional programming paradigm emphasizes the role of functions in coding tasks (for theoretical foundations, see Church's $\{\lambda - calculus\}$ 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[5].

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

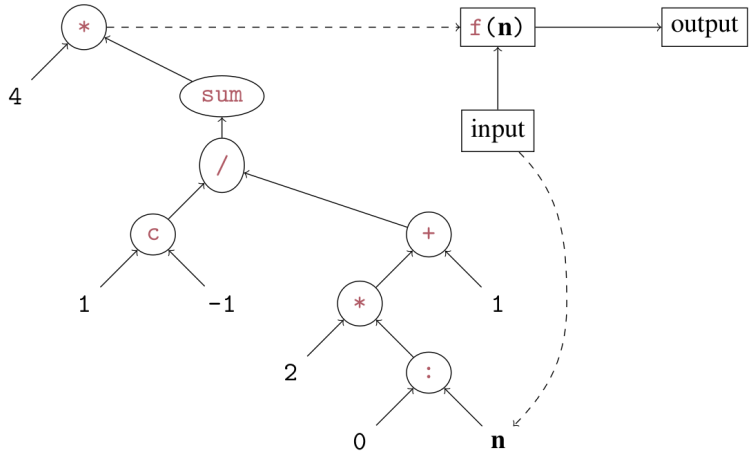
Note that

```
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:



7.2 Defining `R` functions

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

```
function(parameter_list) body
```

```
## function(parameter_list) body
```

[5]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.

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)
```

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
```

7.2.1 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
```

7.2.2 Argument checking

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

```
mean("test")
```

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

```
## [1] NA
```

```
mean(NULL)
```

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

```
## [1] NA
```

```
mean(sum)
```

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

```
## [1] NA
```

It is our responsibility to restrict the function's domain appropriately. For example, `mean()` 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)`.

7.2.3 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.

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.

7.3 Storing functions for later use

7.3.1 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
```

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.

7.3.2 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
```

7.4 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 in eval(expr, envir, enclos): nie znaleziono obiektu 'fy'
```

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.

7.5 Functions' arguments

7.5.1 Pass-by-value

Virtually all[6] `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
```

[6]The only exception: environments.

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.

7.5.2 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
```

```
f(b = 100)
```

```
## [1] 101
```

```
f()
```

```
## [1] 3
```

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

7.5.3 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 in match.arg(kind): 'arg' should be one of "first", "second", "fird"
```

```
f("tenf")
```

```
## Error in match.arg(kind): '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)
```

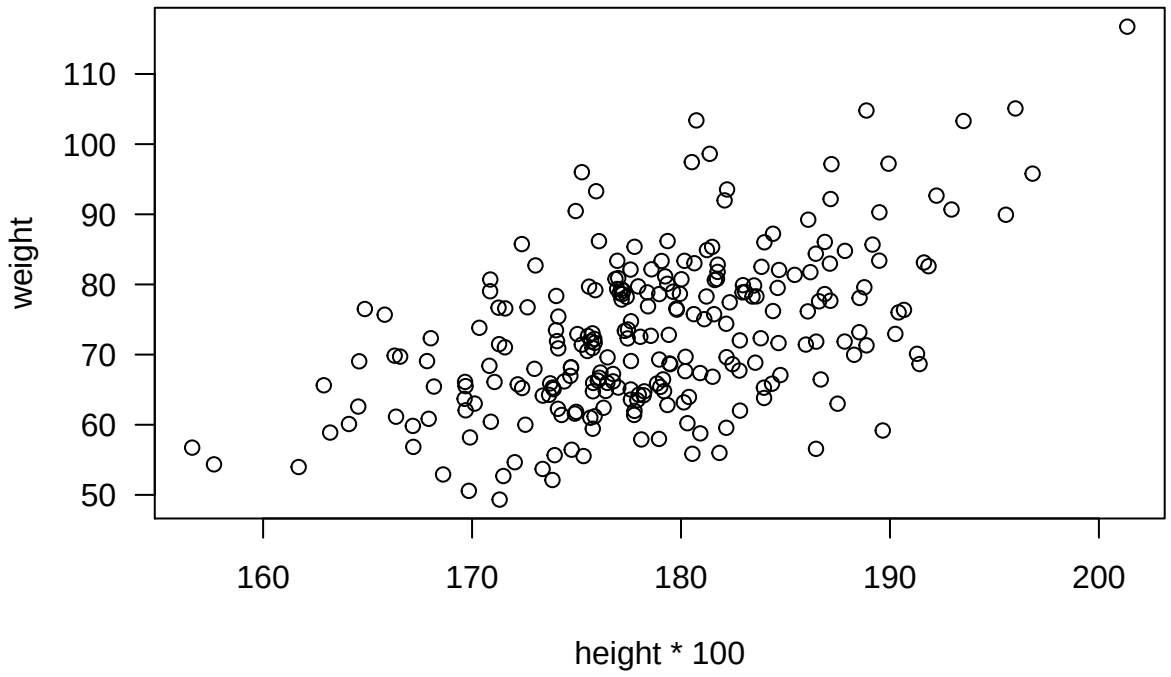
```
## vals = 1 2 3 4 5
```

```
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
```



7.5.2 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 in print(..2): the ... list contains fewer than 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
```

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.

7.6 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.

7.7 Bibliography

- R Core Team, *{An introduction to R}*, 2014, Sec. 10
- R Core Team, *{R language definition}*, 2014, Sec. 4

8 Unit testing. Debugging. Exception handling

8.1 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.

8.2 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:

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. $F(0) = 1; F(1) = 1; F(n) = F(n-1) + F(n-2), n \geq 2$.

We have: $(1, 1, 2, 3, 5, 8, 13, 21, ...)$ The key question is: What to test for here?

In `R`, a function accepts "anything" on input. It may also output "anything". It is very important to assurethat 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,
- 0s, negative values, empty vectors,
- FP arithmetic accuracy issues,
- preservation of input object's attributes.

Generally, the 1stopifnot()1 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 in fib(mean): is.numeric(x) is not TRUE
```

```
fib("4") # OK
```

```
## Error in fib("4"): 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[7].

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.

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

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

8.4 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
```

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

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 in sqrt(-1): wyprodukowano wartości NaN
```

```
## [1] NaN
```

```
1:2 + 1:3
```

```
## Warning in 1:2 + 1:3: długość dłuszego obiektu nie jest wielokrotnością długości
## krótszego obiektu
```

```
## [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): (przetworzone z ostrzeżenia) wyprodukowano wartości NaN
```

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.

```
x <- 1:5
x[10] <- 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 in eval(expr, envir, enclos): This is an error.
```

```
unknown_function(unknown_object)
```

```
## Error in unknown_function(unknown_object): nie udało się znaleźć funkcji 'unknown_function'
```

```
cat(geterrmessage())
```

```
## Error in unknown_function(unknown_object) :
##   nie udało się znaleźć funkcji '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 in test(): 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_e
```

```
## Error in tryCatch(expression_to_try, error = error_handling_function_1arg, : nie znaleziono obiektu
```

```
## Error in tryCatch(expression_to_try, error = error_handling_function_1arg, : nie znaleziono obiektu
```

`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...

```
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
```

8.5 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()`.

8.6 Bibliography

- R Core Team, *{R language definition}*, 2014, Sec. 8, 9
- R Core Team, *{Writing R extensions}*, 2014, Sec. 3
- 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

9 Attributes

9.1 Introduction

Almost all[8] R objects may be equipped with various **attributes**. Attributes are kind of **metadata**, i.e. additional information on objects. Setting some attributes may have a significant impact on the way an R object interacts with R functions.

9.2 Getting and setting attributes

The `attr()` function may be used to set an object's attribute. An attribute is a key-value pair. Except for some special attributes (see below), we are free to associate any metadata with an R object.

```
x <- (-5):5
attr(x, "color") <- "green"
attr(x, "which_positive") <- which(x > 0)
attr(x, "favorite_fun") <- exp
```

The above is equivalent to:

```
x <- structure((-5):5, color="green",
               which_positive=which(x > 0), favorite_fun=exp)
```

The `attr()` function may also be used to get an object's attribute.

[8] All except for `NULL`.

```
attr(x, "which_positive")
```

```
## [1]  7  8  9 10 11
```

```
attr(x, "favorite") # autocompletion
```

```
## function (x)  .Primitive("exp")
```

```
attr(x, "no_such_attribute")
```

```
## NULL
```

Note how an `R` object equipped with such metadata is printed:

```
x
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
## attr(,"color")
## [1] "green"
## attr(,"which_positive")
## [1]  7  8  9 10 11
## attr(,"favorite_fun")
## function (x)  .Primitive("exp")
```

```
str(x)
```

```
##  int [1:11] -5 -4 -3 -2 -1 0 1 2 3 4 ...
##  - attr(*, "color")= chr "green"
##  - attr(*, "which_positive")= int [1:5] 7 8 9 10 11
##  - attr(*, "favorite_fun")=function (x)
```

What is most important, `x` is still an "ordinary" numeric vector.

```
mode(x)
```

```
## [1] "numeric"
```

```
x[1]
```

```
## [1] -5
```

```
mean(x)
```

```
## [1] 0
```

```
x[attr(x, "which_positive")]
```

```
## [1] 1 2 3 4 5
```

In other words, we may do anything with x as with any other numeric vector.

To remove an attribute, "set" its value to `NULL`.

```
attr(x, "favorite_fun") <- NULL
x
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
## attr(,"color")
## [1] "green"
## attr(,"which_positive")
## [1]  7  8  9 10 11
```

On the other hand, to fetch all attributes, use the `attributes()` function.

```
attributes(x) # returns a (named-see below) list
```

```
## $color
## [1] "green"
##
## $which_positive
## [1]  7  8  9 10 11
```

Some R functions set attributes in order to provide their users with additional information on resulting objects:

```
x <- c(1, 2, NA, 4, NA)
na.omit(x)
```

```
## [1] 1 2 4
## attr(,"na.action")
## [1] 3 5
## attr(,"class")
## [1] "omit"
```

Here, `na.action` gives the positions at which `NA`s were found. Such an information is not useful for most users – they are free to ignore it.

9.2.1 Special attributes

There are attributes that have a **special meaning**. Their values must meet some strict constraints, see Tab. 7. What is important, special attributes may be accessed via `comment()`, `class()`[9], etc. functions. These functions may sometimes return a sensible value even if `attr()` has not been called explicitly.

Table 7: Special R attributes.

Attribute	Meaning
`comment`	ignored by `print()`; Value: character vector
`class`	an object's S3 class; Value: character vector
`names`	a vector's elements' names; Value: character vector

9.2.2 The comment attribute

The comment attribute is probably the least interesting one.

```
x <- 1:5
comment(x) <- "What a nice object!"
x # comment printing is suppressed
```

```
## [1] 1 2 3 4 5
```

```
attr(x, "comment")
```

```
## [1] "What a nice object!"
```

```
comment(x)
```

```
## [1] "What a nice object!"
```

[9] Also: `dim`, `dimnames`, `row.names` – see Compound types.

```
comment(x) <- 10
```

```
## Error in `comment<-`(`*tmp*`, value = 10): próba ustawienia niepoprawnego atrybutu 'comment'
```

Note the usage of `comment()` and the fact that only character vectors constitute valid comments.

9.2.3 The names attribute

The names attribute may be fed with a character vector. It is used to label a vector's elements.

```
x <- seq(0, 1, length.out = 5)
names(x) <- c("1st", "2nd", "3rd", "4th", "5th")
x
```

```
##  1st  2nd  3rd  4th  5th
## 0.00 0.25 0.50 0.75 1.00
```

Other example:

```
structure(list(1:10, mean), names = c("vector", "function"))
```

```
## $vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $`function`
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x55a28bae17d8>
## <environment: namespace:base>
```

Note the following:

```
structure(1:4, names = c("a", "b", "c", "d", "e"))
```

```
## Error in attributes(.Data) <- c(attributes(.Data), attrib): atrybut 'names' [5] musi mieć tę samą dłu
```

```
(x <- structure(1:4, names = c("a", "b", "c")))
```

```
##    a    b    c <NA>
##    1    2    3    4
```

```
names(x)
```

```
## [1] "a" "b" "c" NA
```

```
x <- structure(1:4, names = c("a", "b", "c", "d"))
unname(x) # x <- unname(x) is equivalent to attr(x,'names')<-NULL
```

```
## [1] 1 2 3 4
```

```
names(x)[2] <- "zzz"
```

The `c()` and `list()` functions may set the names attribute automatically:

```
list(first=1:10, second=100:110)
```

```
## $first
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $second
##  [1] 100 101 102 103 104 105 106 107 108 109 110
```

```
c("1st"=1, "2nd"=2) # 1st = invalid syntactic name = use quotes
```

```
## 1st 2nd
##   1   2
```

The names attribute affects not only the `print()` function. The `"["` and `"[["` operators also may take it into account.

```
x <- structure(1:4, names = c("a", "b", "c", "d"))
x["a"]
```

```
## a
## 1
```

```
x[c("a", "d", "a")]
```

```
## a d a
## 1 4 1
```

Moreover:

```
y <- list(a = 1, b = 2)
y["a"] # subset
```

```
## $a
## [1] 1
```

```
y[["a"]] # extract
```

```
## [1] 1
```

A simple database:

```
(numberOfParticipants <- c(male=20, female=23))
```

```
##   male female
##     20     23
```

```
# a new male participant joined the experiment:
numberOfParticipants["male"] <- numberOfParticipants["male"]+1
numberOfParticipants
```

```
##   male female
##     21     23
```

Note that names are not identifiers: They might be non-unique.

```
x <- c(one = 1, two = 2, one = 3)
x["one"] # first occurrence returned
```

```
## one
##   1
```

```
x[names(x) == "one"]
```

```
## one one
##   1   3
```

Searching for a given label is $O(n)$, pessimistically.

```
library(microbenchmark)
x <- structure(as.list(1:1000000), names=as.character(1:1000000))
microbenchmark(x[[10000]], x[[100000]], x[[1000000]], unit="ms") # index - O(1)
```

```
## Unit: milliseconds
##        expr      min       lq       mean   median       uq      max neval cld
##  x[[10000]] 0.000103 0.000106 0.00011856 0.000107 0.000109 0.000622   100   a
##  x[[1e+05]] 0.000102 0.000107 0.00011457 0.000108 0.000109 0.000425   100   a
##  x[[1e+06]] 0.000103 0.000106 0.00014508 0.000108 0.000109 0.003713   100   a
```

```
microbenchmark(x[["10000"]], x[["100000"]], x[["1000000"]], unit="ms") # name - O(n)
```

```
## Unit: milliseconds
##            expr       min        lq       mean    median         uq        max
##    x[["10000"]]  0.171022  0.193162  0.2058719  0.209069  0.2137055   0.326482
##   x[["100000"]]  1.920029  2.019183  2.0603123  2.045120  2.0908170   2.368511
##  x[["1000000"]] 20.829816 21.442580 24.5386765 21.577449 21.7894400 316.440024
##  neval cld
##    100  a
##    100  a
##    100   b
```

If `x` is a list, then `x$label` is most often equivalent to `x[["label"]]`.

```
x <- list(one = 1, `2nd` = 2)
x$one
```

```
## [1] 1
```

```
x$three <- 3 # adjust length
str(x)
```

```
## List of 3
##  $ one  : num 1
##  $ 2nd  : num 2
##  $ three: num 3
```

```
x$"2nd" # 2nd - not a syntactic name - use quotes
```

```
## [1] 2
```

By the way, `"$"` implements partial matching of labels, but its usage is not recommended.

9.2.4 The class attribute

The class attribute denotes an object's so-called **S3 class**. The S3-style object-oriented programming will be covered in-depth later on. However, the concept is so important that we should at least sketch some basic S3 concepts now.

Firstly, note that `class()` returns a sensible value even though the attribute is not set explicitly.

```
x <- c("a", "b", "c")
class(x)
```

```
## [1] "character"
```

```
attr(x, "class")
```

```
## NULL
```

By default, `class(x) == mode(x)`, except for `typeof(x) == "integer"`.

A function's behavior may change drastically depending on its argument's class attribute.

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x55a28bc58740>
## <environment: namespace:base>
```

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x55a28bae17d8>
## <environment: namespace:base>
```

Each function that makes a call to `UseMethod()` is called a generic function. Each generic function dispatches the control flow to another function, called method.

Let `f()` be a generic function. Assume that we are calling it on an object of class `classname`.

1. If there exists a function named `f.classname()`, this is the routine to be evaluated on a given object.
2. Otherwise, `f.default()` will be called.

Compare, for example, `?plot` with `?plot.default`.

```
print.Pretty <- function(x, ...) {
  cat("PRETTY", paste(x, collapse = ", "), ":-) \n")
}
x <- 1:5
x
```

```
## [1] 1 2 3 4 5
```

```
class(x) <- "Pretty"
x
```

```
## PRETTY 1, 2, 3, 4, 5 :-)
```

```
print.default(x)
```

```
## [1] 1 2 3 4 5
## attr(,"class")
## [1] "Pretty"
```

Example: A hypothesis test.

```
test <- shapiro.test(rnorm(100))
test
```

```
##
##  Shapiro-Wilk normality test
##
## data:  rnorm(100)
## W = 0.99024, p-value = 0.6842
```

What is that?

```
class(test)
```

```
## [1] "htest"
```

```
typeof(test)
```

```
## [1] "list"
```

Thus, an object of class htest is an ordinary R list.

```
str(unclass(test))
```

```
## List of 4
##  $ statistic: Named num 0.99
##   ..- attr(*, "names")= chr "W"
##  $ p.value  : num 0.684
##  $ method   : chr "Shapiro-Wilk normality test"
##  $ data.name: chr "rnorm(100)"
```

```
test$p.value
```

```
## [1] 0.684178
```

```
test[["method"]]
```

```
## [1] "Shapiro-Wilk normality test"
```

The only reason why htest is presented in a non-standard way is due to the overloaded `print()` method.

```
print.htest # inaccessible directly
```

```
## Error in eval(expr, envir, enclos): nie znaleziono obiektu 'print.htest'
```

```
getS3method("print", "htest") # here it is
```

```
## function (x, digits = getOption("digits"), prefix = "\t", ...)
## {
##     cat("\n")
##     cat(strwrap(x$method, prefix = prefix), sep = "\n")
##     cat("\n")
##     cat("data:  ", x$data.name, "\n", sep = "")
##     out <- character()
##     if (!is.null(x$statistic))
##         out <- c(out, paste(names(x$statistic), "=", format(x$statistic,
##             digits = max(1L, digits - 2L))))
##     if (!is.null(x$parameter))
##         out <- c(out, paste(names(x$parameter), "=", format(x$parameter,
##             digits = max(1L, digits - 2L))))
##     if (!is.null(x$p.value)) {
##         fp <- format.pval(x$p.value, digits = max(1L, digits -
##             3L))
##         out <- c(out, paste("p-value", if (substr(fp, 1L, 1L) ==
##             "<") fp else paste("=", fp)))
##     }
##     cat(strwrap(paste(out, collapse = ", ")), sep = "\n")
##     if (!is.null(x$alternative)) {
##         cat("alternative hypothesis: ")
##         if (!is.null(x$null.value)) {
##             if (length(x$null.value) == 1L) {
##                 alt.char <- switch(x$alternative, two.sided = "not equal to",
##                   less = "less than", greater = "greater than")
##                 cat("true ", names(x$null.value), " is ", alt.char,
##                   " ", x$null.value, "\n", sep = "")
##             }
##             else {
##                 cat(x$alternative, "\nnull values:\n", sep = "")
```

```
##                 print(x$null.value, digits = digits, ...)
##             }
##         }
##         else cat(x$alternative, "\n", sep = "")
##     }
##     if (!is.null(x$conf.int)) {
##         cat(format(100 * attr(x$conf.int, "conf.level")), " percent confidence interval:\n",
##             " ", paste(format(x$conf.int[1:2], digits = digits),
##                 collapse = " "), "\n", sep = "")
##     }
##     if (!is.null(x$estimate)) {
##         cat("sample estimates:\n")
##         print(x$estimate, digits = digits, ...)
##     }
##     cat("\n")
##     invisible(x)
## }
## <bytecode: 0x55a295bc7010>
## <environment: namespace:stats>
```

9.3 What attributes are preserved by base R functions?

Indexing operator – `?"["`: *Subsetting (except by an empty index) will drop all attributes except* `names`, `dim` and `dimnames`.

```
structure(1:5, names = letters[1:5], attrib = "val")[2]
```

```
## b
## 2
```

Binary operators – `?"+"`: *Most attributes are taken from the longer argument.* `Names` *will be copied from the first if it is the same length as the answer, otherwise from the second if that is. If the arguments are the same length, attributes will be copied from both, with those of the first argument taking precedence when the same attribute is present in both arguments.*

```
structure(1:5, a1 = "v1") * structure(1, a2 = "v2")
```

```
## [1] 1 2 3 4 5
## attr(,"a1")
## [1] "v1"
```

Vectorized math functions – should generally preserve all the attributes.

```
log(structure(1:10, attrib = "val"))
```

```
##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
## attr(,"attrib")
## [1] "val"
```

Aggregation functions – generally drop all the attributes.

```
mean(1:5, class = "xx", names = letters[1:5], attrib = "val")
```

```
## [1] 3
```

9.4 Summary

Attributes are a way to associate some metadata with `R` objects. Setting some attributes (like `class` or `names`) may have a significant impact on the way an `R` object interacts with `R` functions.

9.5 Bibliography

- R Core Team, *{An introduction to R}*, 2014, Sec. 3
- R Core Team, *{R language definition}*, 2014, Sec. 2.2
- R Core Team, *{Writing R extensions}*, 2014, Sec. 3, 4

10 Compound types

10.1 Introduction

Let us examine the 3 most commonly used `R` compound types:

- `Factors`
- `Matrices` (and their generalization, `Arrays`)
- `Data frames`

Each of the above are in fact "only slightly extended" versions of base types, like atomic vectors and lists.

10.2 Factors

`Factors` are vector-like objects storing qualitative data. The predefined set of categories (*levels*) should be relatively small. We may say that factors are *enumerated types*.

```
factor(c("male", "female", "female", "male", "female"))
```

```
## [1] male   female female male   female
## Levels: female male
```

```
str(factor(c(1, 3, 1, 2, 5, 2)))
```

```
##  Factor w/ 4 levels "1","2","3","5": 1 3 1 2 4 2
```

How are `factors` represented in `R`?

```
f <- factor(c(1, 3, 1, 2, 5, 2))
class(f)
```

```
## [1] "factor"
```

```
typeof(f)
```

```
## [1] "integer"
```

We see that an object of class `factor` is in fact an `integer` vector. Moreover:

```
f <- factor(c(1, 3, 1, 2, 5, 2))
unclass(f)
```

```
## [1] 1 3 1 2 4 2
## attr(,"levels")
## [1] "1" "2" "3" "5"
```

The `levels` attribute is a character vector of all levels' labels. The numeric values (consecutive natural numbers), on the other hand, represent the categories' identifiers.

```
attr(f, "levels")[as.integer(f)]
```

```
## [1] "1" "3" "1" "2" "5" "2"
```

```
as.character(f)
```

```
## [1] "1" "3" "1" "2" "5" "2"
```

```
as.integer(as.character(f)) # not the same as as.integer(f)
```

```
## [1] 1 3 1 2 5 2
```

In fact, `factors` may be created manually:

```
test <- c(1L, 4L, 3L, 2L, 1L, 3L)
levels(test) <- c("one", "two", "three", "four")
class(test) <- "factor"
test
```

```
## [1] one   four  three two   one   three
## Levels: one two three four
```

Note that `levels(·)` is the same as `attr(·, "levels")`.

Note that many methods have been overloaded for factors. In particular, `R`'s authors did not want factors to be confused with "standard" vectors:

```
f <- factor(c(1, 3, 1, 2, 5, 2))
is.factor(f)
```

```
## [1] TRUE
```

```
is.vector(f)
```

```
## [1] FALSE
```

```
is.atomic(f)
```

```
## [1] TRUE
```

```
is.integer(f)
```

```
## [1] FALSE
```

```
is.character(f)
```

```
## [1] FALSE
```

We may define any strict linear order on a factor's levels.

```
f <- factor(c("one", "three", "two", "one"),
            levels=c("one", "two", "three"), ordered=TRUE)
sort(f)
```

```
## [1] one   one   two   three
## Levels: one < two < three
```

```
which(f > "one")
```

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

```
f[f > "two"] # not a lexicographic order here
```

```
## [1] three
## Levels: one < two < three
```

```
max(f)
```

```
## [1] three
## Levels: one < two < three
```

Creating contingency tables:

```
table(factor(c("one", "three", "two", "one")))
```

```
##
##   one three   two
##     2     1     1
```

```
table(factor(c("male", "female", "male", "male", "female")),
      factor(c("low", "low", "high", "high", "high")))
```

```
##
##          high low
##   female    1   1
##   male      2   1
```

Dropping unused levels:

```
(f <- factor(c(1, 2, 3, 5, 1), levels = 1:5))
```

```
## [1] 1 2 3 5 1
## Levels: 1 2 3 4 5
```

```
nlevels(f) # length(levels(f))
```

```
## [1] 5
```

```
(f <- droplevels(f))
```

```
## [1] 1 2 3 5 1
## Levels: 1 2 3 5
```

```
nlevels(f)
```

```
## [1] 4
```

By the way, it is easy to change the labels in one call:

```
levels(f) <- c("one", "two", "three", "five")
f
```

```
## [1] one   two   three five  one
## Levels: one two three five
```

Such an operation is not easy in case of ordinary character vectors.

Splitting another vector w.r.t. a qualitative variable:

```
height <- c(164, 182, 173, 194, 159)
gender <- c("m", "f", "m", "m", "f")
split(height, gender)
```

```
## $f
```

```
## [1] 182 159
##
## $m
## [1] 164 173 194
```

```
lapply(split(height, gender), mean) # avg height in each group
```

```
## $f
## [1] 170.5
##
## $m
## [1] 177
```

A somewhat similar operation:

```
height <- c(164, 182, 173, 194, 159)
gender <- c("m", "f", "m", "m", "f")
tapply(height, gender, mean)
```

```
##     f     m
## 170.5 177.0
```

"Discretizing" numeric data:

```
(x <- round(rnorm(10), 1))
```

```
##  [1] -0.6  1.5  1.1  0.0  0.6 -1.8  2.2  0.5 -0.7  0.0
```

```
cut(x, c(-Inf, -1, 0, 1, Inf))
```

```
##  [1] (-1,0]    (1, Inf]  (1, Inf]  (-1,0]    (0,1]     (-Inf,-1] (1, Inf]
##  [8] (0,1]     (-1,0]    (-1,0]
## Levels: (-Inf,-1] (-1,0] (0,1] (1, Inf]
```

```
cut(x, c(-Inf, -1, 0, 1, Inf),
    labels=c("very_small", "small", "large", "very_large"))
```

```
##  [1] small      very_large very_large small      large      very_small
##  [7] very_large large      small      small
## Levels: very_small small large very_large
```

10.3 Matrices and Arrays

`Matrices` are built upon vectors (`atomic` ones or `lists`).

```
x <- 1:6
dim(x) <- c(2, 3) # or attr(x, 'dim') <- c(2, 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
class(x) # vector + dim attr => implicit matrix class
```

```
## [1] "matrix"
```

```
is.matrix(x)
```

```
## [1] TRUE
```

```
is.numeric(x)
```

```
## [1] TRUE
```

Note that a `matrix`'s elements are stored in a column-wise order.

Once again: a `matrix` is nothing more than a vector equipped with the `dim` attribute.

```
x <- 1:6
dim(x) <- c(2, 3) # or attr(x, 'dim') <- c(2, 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
dim(x) <- c(3, 2)
x
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
as.numeric(x) # drops the dim attribute
```

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

Moreover:

```
x <- matrix(1:6, nrow=2, ncol=3) # == structure(1:6, dim=c(2, 3))
x^2
```

```
##      [,1] [,2] [,3]
## [1,]    1    9   25
## [2,]    4   16   36
```

```
x*c(-1, 2)
```

```
##      [,1] [,2] [,3]
## [1,]   -1   -3   -5
## [2,]    4    8   12
```

```
x*x
```

```
##      [,1] [,2] [,3]
## [1,]    1    9   25
## [2,]    4   16   36
```

Note that all the above are vectorized *elementwise* ops. Here is a character matrix:

```
matrix(letters[1:4], ncol=2) # nrow auto-guessed
```

```
##      [,1] [,2]
## [1,] "a"  "c"
## [2,] "b"  "d"
```

And some list-based matrices:

```
structure(list(mean, sd, var, median), dim = c(2, 2))
```

```
##      [,1] [,2]
## [1,] ?    ?
```

```
## [2,] ?    ?
```

```
structure(list(1:5, c(1, 5.4)), dim = c(1, 2))
```

```
##      [,1]      [,2]
## [1,] Integer,5 Numeric,2
```

Note some printing issues.

An `array` is a generalization of a `matrix`:

```
structure(1:12, dim = c(2, 3, 2))
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

See also: `?array`.

The `dimnames` attribute may be set in order to label row/column names (cf. the `names` attrib for atomic vectors). Generally, it is a list of `dim(·)` character vectors, of lengths `dim(·)[1]`, `dim(·)[2]`, ..., respectively.

```
(x <- matrix(1:6, nrow=2,
             dimnames=list(c("r1", "r2"), c("c1", "c2", "c3"))))
```

```
##    c1 c2 c3
## r1  1  3  5
## r2  2  4  6
```

```
dim(x)
```

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

```
str(dimnames(x))
```

```
## List of 2
##  $ : chr [1:2] "r1" "r2"
##  $ : chr [1:3] "c1" "c2" "c3"
```

"Multidimensional" indices are available for `"["` if dim is set:

```
(x <- matrix(letters[1:6], nrow = 2))
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "c"  "e"
## [2,] "b"  "d"  "f"
```

```
x[1, 2] # 1st row, 2nd column
```

```
## [1] "c"
```

```
x[1, ] # 1st row
```

```
## [1] "a" "c" "e"
```

```
x[1:2, 2:3] # 1st and 2nd row, 2nd and 3rd column [select block]
```

```
##      [,1] [,2]
## [1,] "c"  "e"
## [2,] "d"  "f"
```

Here is a `dimnames`-based subsetting:

```
(x <- matrix(letters[1:6], nrow=2,
             dimnames=list(c("r1", "r2"), c("c1", "c2", "c3"))))
```

```
##    c1  c2  c3
## r1 "a" "c" "e"
## r2 "b" "d" "f"
```

```
x[c("r2", "r1"), c("c3", "c1")]
```

```
##    c3  c1
## r2 "f" "b"
## r1 "e" "a"
```

A `matrix` can also be subsetted with a 2-column numeric `matrix`:

```
(x <- matrix(letters[1:6], nrow=2))
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "c"  "e"
## [2,] "b"  "d"  "f"
```

```
(y <- matrix(c(1, 3, 2, 1, 1, 1), byrow=TRUE, ncol=2))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    1
## [3,]    1    1
```

```
x[y]
```

```
## [1] "e" "b" "a"
```

The "*" operator does an element-wise multiplication. Matrix multiplication is available via "%*%":

```
(a <- matrix(1:4, nrow = 2))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
(b <- matrix(c(1, -1, -1, 1), nrow = 2))
```

```
##      [,1] [,2]
## [1,]    1   -1
## [2,]   -1    1
```

```
a %*% b # compare with a*b
```

```
##      [,1] [,2]
## [1,]   -2    2
## [2,]   -2    2
```

"Expanding" matrices:

```r
(x <- matrix(1:6,nrow = 2))
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
cbind(x, 1:2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    1
## [2,]    2    4    6    2
```

```r
rbind(x, 1:3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    1    2    3
```

```r
rbind(1:5, 11:15)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]   11   12   13   14   15
```

`simplify2array()` tries to "simplify" a given list:

```r
simplify2array(list(1, 2, 3)) # result = atomic vector
```

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

```r
simplify2array(list(1:2, 3:4, 5:6)) # result = matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
simplify2array(list(1, 2:3)) # cannot simplify
```

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

Thus, `sapply(·)` is a convenient substitute for `simplify2array(lapply(·))`:

```r
sapply(list(1:10, 11:20, 21:30), mean)
```

```
## [1]  5.5 15.5 25.5
```

```r
sapply(list(1:10, 11:20, 21:30), range)
```

```
##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]   10   20   30
```

On the other hand, `apply()` applies a given operation on each row/column:

```r
(x <- matrix(1:6, nrow = 2))
```

```
##      [,1] [,2] [,3]
```

```
## [1,]    1    3    5
## [2,]    2    4    6
```

```
apply(x, 1, sum) # each row
```

```
## [1]  9 12
```

```
apply(x, 2, mean) # each column
```

```
## [1] 1.5 3.5 5.5
```

See also: `?rowSums`, `?colSums`, `?rowMeans`, `?colMeans`.

The `outer()` function applies a given binary operation on each pair of elements in two given vectors:

```
outer(c(TRUE, FALSE, NA), c(TRUE, FALSE, NA), "|")
```

```
##       [,1]  [,2] [,3]
## [1,] TRUE  TRUE TRUE
## [2,] TRUE FALSE   NA
## [3,] TRUE    NA   NA
```

```
outer(c("a", "b"), 1:3, paste, sep = "")
```

```
##      [,1] [,2] [,3]
## [1,] "a1" "a2" "a3"
## [2,] "b1" "b2" "b3"
```

For more information on `matrix` operations, read more: `?t`, `?diag`, `?upper.tri`, `?lower.tri`, `?isSymmetric`, `?maxCol`, `?aperm`, `?norm`, `?dist`, `?det`, `?eigen`, `?qr`, `?svd`, `?chol`, `?kappa`, `?solve`, `?lsfit`. See also: the Matrix package (e.g. sparse and band matrices, etc.) and igraph (graphs), relations (binary relations),...

10.4 Data frames

A `data frame` is a list of vectors, each having the same length. In the practice of data analysis it is very often used to represent *tabular data*.

```
data.frame(gender=c("m", "f", "m", "f"), height=c(185, 158, 191, 174))
```

```
##   gender height
## 1      m    185
## 2      f    158
## 3      m    191
## 4      f    174
```

Some basic type information:

```
df <- data.frame(gender=c("m", "f", "m", "f"),
height=c(185, 158, 191, 174))
class(df)
```

```
## [1] "data.frame"
```

```
typeof(df)
```

```
## [1] "list"
```

```
is.list(df)
```

```
## [1] TRUE
```

```
is.data.frame(df)
```

```
## [1] TRUE
```

```
str(unclass(df))
```

```
## List of 2
##  $ gender: Factor w/ 2 levels "f","m": 2 1 2 1
##  $ height: num [1:4] 185 158 191 174
##  - attr(*, "row.names")= int [1:4] 1 2 3 4
```

Here is how to create a `data frame` manually:

```
df <- list(c("m", "f", "m", "f"), c(185, 158, 191, 174))
names(df) <- c("gender", "height")
attr(df, "row.names") <- c("1", "2", "3", "4")
class(df) <- "data.frame"
df
```

```
##   gender height
## 1      m    185
## 2      f    158
## 3      m    191
## 4      f    174
```

This enables us to create very fancy `data frames`:

```
(x <- structure(list(
  list(matrix(1:4, ncol=2),
       matrix(11:14, ncol=2)
       )
  ),
  class="data.frame",
  names="c1",
  row.names=c("r1", "r2")
  )
 )
```

```
##                c1
## r1     1, 2, 3, 4
## r2 11, 12, 13, 14
```

```
x[["c1"]][[1]]
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Let us consider the following data set:

```
(survey <- data.frame(
  gender = c("m", "m", "f", "m", "f", "f"),
  coffee = c(FALSE, TRUE, TRUE, FALSE, TRUE, FALSE),
  time = c(23, 25, 31, 46, 24, 38),
  weight = c(69, 71, 58, 98, 63, 41)
  )
 )
```

```
##   gender coffee time weight
## 1      m  FALSE   23     69
```

```
## 2      m   TRUE   25     71
## 3      f   TRUE   31     58
## 4      m  FALSE   46     98
## 5      f   TRUE   24     63
## 6      f  FALSE   38     41
```

Each `data frame` is a named `list`: here is how we may extract the last column:

```
survey$weight
```

```
## [1] 69 71 58 98 63 41
```

```
survey[["weight"]]
```

```
## [1] 69 71 58 98 63 41
```

```
survey[[4]]
```

```
## [1] 69 71 58 98 63 41
```

On the other hand, here is how we may subset a `data frame`:

```
survey[c(1, 3)]
```

```
##   gender time
## 1      m   23
## 2      m   25
## 3      f   31
## 4      m   46
## 5      f   24
## 6      f   38
```

We can do that even though the dim attribute is not set explicitly:

```
attr(survey, "dim")
```

```
## NULL
```

The `dim()` function returns a sensible value:

```
dim(survey)
```

```
## [1] 6 4
```

Thus, a `data frame` may sometimes imitate a `matrix`'s behavior. In particular, a two-dimensional indexing operator is available:

```
survey[1, ] # 1st row
```

```
##   gender coffee time weight
## 1      m  FALSE   23     69
```

```
survey[, 4] # == survey[[4]] != survey[4]
```

```
## [1] 69 71 58 98 63 41
```

```
survey[1:2, 3:4]
```

```
##   time weight
## 1   23     69
## 2   25     71
```

In fact, a `data frame` may be easily converted to a `matrix`.

```
as.matrix(survey)
```

```
##      gender coffee  time weight
## [1,] "m"    "FALSE" "23" "69"
## [2,] "m"    "TRUE"  "25" "71"
## [3,] "f"    "TRUE"  "31" "58"
## [4,] "m"    "FALSE" "46" "98"
## [5,] "f"    "TRUE"  "24" "63"
## [6,] "f"    "FALSE" "38" "41"
```

Note that coercion occurred.

By the way, when a `data frame` is created, character data are automatically converted to `factors`s:

```
class(survey$gender)
```

```
## [1] "factor"
```

See, however, the `stringsAsFactors` argument of `data.frame()` and `stringsAsFactors` option (`?options`).

All row names should be unique:

```
row.names(survey) <- c("Frank", "Avishai", "Ella", "John", "Ella", "Elis")
```

```
## Error: (przetworzone z ostrzeżenia) non-unique value when setting 'row.names': 'Ella'
```

```
row.names(survey) <- c("Frank", "Avishai", "Ella", "John", "Nina", "Elis")
```

Thus, row names serve as row identifiers:

```
survey["Ella", ]
```

```
##      gender coffee time weight
## Ella      f   TRUE   31     58
```

It is recommended that column names are syntactically valid names, see `?make.names`:

```
test <- data.frame(good.name=1:2, "1bad.name"=3:4)
test # auto-fix
```

```
##   good.name X1bad.name
## 1         1          3
## 2         2          4
```

```
test$good.name
```

```
## [1] 1 2
```

```
names(test)[2] <- "1bad.name"
test$"1bad.name" # test$1bad.name won't work
```

```
## [1] 3 4
```

Filtering:

```
survey[survey$gender == "m" & survey$weight > 70, ]
```

```
##         gender coffee time weight
## Avishai      m   TRUE   25     71
## John         m  FALSE   46     98
```

```
subset(survey, gender == "m" & weight > 70)
```

```
##         gender coffee time weight
## Avishai      m   TRUE   25     71
## John         m  FALSE   46     98
```

```
subset(survey, gender == "f", select = -weight)
```

```
##      gender coffee time
## Ella      f   TRUE   31
## Nina      f   TRUE   24
## Elis      f  FALSE   38
```

See also: `?with`, `?within`, `?transform`.

"Expanding" `data frames`:

```
cbind(survey, height = c(184, 159, 173, 162, 195, 178))
```

```
##         gender coffee time weight height
## Frank        m  FALSE   23     69    184
## Avishai      m   TRUE   25     71    159
## Ella         f   TRUE   31     58    173
## John         m  FALSE   46     98    162
## Nina         f   TRUE   24     63    195
## Elis         f  FALSE   38     41    178
```

```
rbind(survey, data.frame(gender = "m", coffee = TRUE, time = 41, weight = 98, row.names = "Steinar"))
```

```
##         gender coffee time weight
## Frank        m  FALSE   23     69
## Avishai      m   TRUE   25     71
## Ella         f   TRUE   31     58
## John         m  FALSE   46     98
## Nina         f   TRUE   24     63
## Elis         f  FALSE   38     41
## Steinar      m   TRUE   41     98
```

Applying various operations:

```
sapply(survey, class) # class of each column (it's a list...)
```

```
##    gender    coffee      time    weight
##  "factor" "logical" "numeric" "numeric"
```

```
tapply(survey$time, survey$gender, mean) # avg time / each gender
```

```
##        f        m
## 31.00000 31.33333
```

See also: `?aggregate`, `?by`, `?ave`.

The `order()` function uses a stable sorting algorithm. Thus, it is possible to order rows of a `data frame` w.r.t. multiple criteria.

```
survey[order(survey$gender, survey$time), ]
```

```
##         gender coffee time weight
## Nina         f   TRUE   24     63
## Ella         f   TRUE   31     58
## Elis         f  FALSE   38     41
## Frank        m  FALSE   23     69
## Avishai      m   TRUE   25     71
```

```
## John         m  FALSE   46     98
```

Note that within each gender, the observations are sorted w.r.t. time. If the algorithm was not stable, sorting w.r.t. gender could break the established relative order w.r.t. time.

Here is a list of "hot" `data frame` processing-related packages:

- data.table
- reshape2
- plyr2 and dplyr
- magrittr

I recommended that you take a look at their features.

10.5 Time series

`Time series` are objects of class ts. Try to find out yourself how are they represented.

10.6 Summary

`Compound` types are extensions of basic types. A `matrix` is based on an ordinary vector. A factor is "something between" a `character` and an `integer` vector. A `data frame` is a special kind of a `list`.

10.7 Bibliography

- R Core Team, *{An introduction to R}*, 2014, Sec. 4, 5, 6.3
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 3, 5, 6

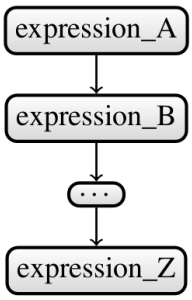
11 Controlling program flow

11.1 Introduction

Up to now, our functions followed the following scheme:

```
f <- function(...) {
  expression_A
  expression_B
  # ...
  expression_Z
}
```

Here is the corresponding program control flow diagram:



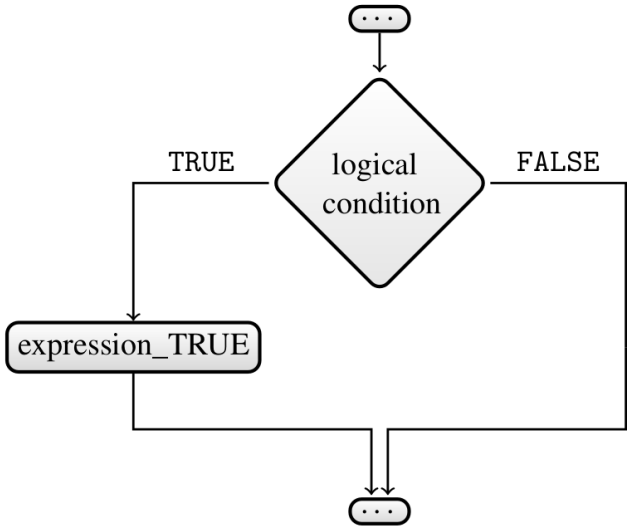
In this case, each expression is evaluated exactly once. As we may sometimes need a different control flow scheme, in this section we will discuss the `R` control-flow expressions:

- Conditional execution: `if`
- Repetitive execution: `for, while, repeat`

11.2 Conditional execution

11.2.1 `if..else`: Syntax

Here is the control flow diagram for the if expression:



Syntax:

```
if (logical_condition) expression_TRUE
```

```
## Error in eval(expr, envir, enclos): nie znaleziono obiektu 'logical_condition'
```

`logical_condition` is a single `R` expression such that `as.logical(logical_condition)` $\in$ `{TRUE, FALSE}` (a single logical value, not `NA`).

Note that `expression_TRUE` will be evaluated if and only if `as.logical(logical_condition)` gives `TRUE`. `expression_TRUE` is a single `R` expression. If you want to evaluate more expressions conditionally, group them with `"{...}"`.

An example:

```
sum2 <- function(x) {
  if (!is.numeric(x)) # gives either TRUE or FALSE
    x <- as.numeric(x) # will throw an error if it's impossible
  sum(x) # return value
  }
sum2(1:4) # already numeric
```

```
## [1] 10
```

```
sum2(c("1", "2", "3")) # coercion
```

```
## [1] 6
```

```
sum2(mean)
```

```
## Error in as.numeric(x): cannot coerce type 'closure' to vector of type 'double'
```

Other examples:

```
if (TRUE) cat("!")
```

```
## !
```

```
if (FALSE) cat("!") # nothing done
if (NA) cat("!")
```

```
## Error in if (NA) cat("!"): brakuje wartości tam, gdzie wymagane jest TRUE/FALSE
```

```
if (5) cat("!") # coercion
```

```
## !
```

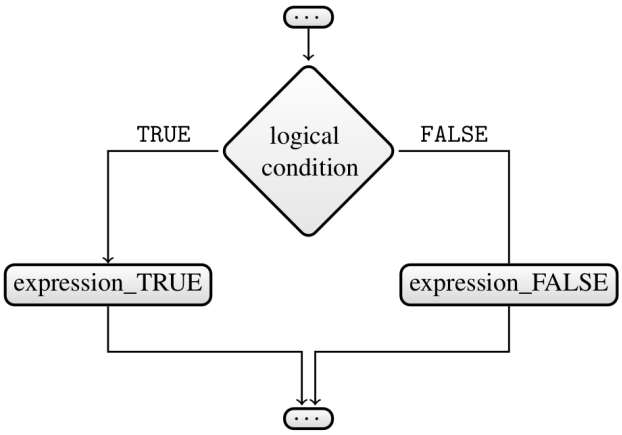
```
if (c(TRUE, FALSE)) cat("!") # warning != error
```

```
## Error in if (c(TRUE, FALSE)) cat("!"): (przetworzone z ostrzeżenia) warunek posiada długość > 1 i ty
```

```
if (c(FALSE, NA, TRUE)) cat("!") # warning != error
```

```
## Error in if (c(FALSE, NA, TRUE)) cat("!"): (przetworzone z ostrzeżenia) warunek posiada długość > 1
```

And here is a control flow diagram of the if..else expression:



Syntax:

```
if (logical_condition) expression_TRUE else expression_FALSE
```

```
## Error in eval(expr, envir, enclos): nie znaleziono obiektu 'logical_condition'
```

`expression_TRUE` will be evaluated if and only if `as.logical(logical_condition)` gives `TRUE`. Other- wise, `expression_FALSE` is executed.

Example – nested ifs:

```
sgn <- function(x) {
  stopifnot(is.numeric(x), length(x) == 1, is.finite(x))
  if (x > 0)
    cat("positive\n")
  else { # if not positive, then either negative or zero
    if (x < 0)
```

```
      cat("negative\n")
    else
      cat("zero\n")
    }
}
sgn(5)
```

```
## positive
```

```
sgn(0)
```

```
## zero
```

The above is of course equivalent to:

```
sgn <- function(x) {
  stopifnot(is.numeric(x), length(x) == 1, is.finite(x))
  if (x > 0)
    cat("positive\n")
  else if (x < 0)
    cat("negative\n")
  else
    cat("zero\n")
  }
sgn(5)
```

```
## positive
```

```
sgn(0)
```

```
## zero
```

Note that the `R` parser in some cases will not interpret the following as we wish to:

```
if (TRUE) print(TRUE)
else print(FALSE)
```

```
## Error: <text>:2:1: nieoczekiwane 'else'
## 1: if (TRUE) print(TRUE)
## 2: else
##    ^
```

This is because the parser cannot predict whether else follows after `if`. Possible fixups:

```
if (TRUE) print(TRUE) else print(FALSE) # one-liner
```

```
## [1] TRUE
```

or:

```
{ # as a grouped expression (e.g. within a function)
  if (TRUE) print(TRUE)
  else print(FALSE)
}
```

```
## [1] TRUE
```

or:

```
if (TRUE) {
  print(TRUE)
```

```
  } else { # no newline before `else`
    print(FALSE)
}
```

```
## [1] TRUE
```

11.2.2 `if..else`: Return value

`if..else` is just a syntactic sugar, equivalent to calling some `R` function.

```
if (TRUE) print(TRUE) else print(FALSE)
```

```
## [1] TRUE
```

```
"if"(TRUE, print(TRUE), print(FALSE))
```

```
## [1] TRUE
```

Each `R` function returns some value. What does `if..else` result in? It turns out that the return value of the `if..else` expression is determined by either `expression_TRUE` or `expression_FALSE`.

```
x <- if (runif(1) > 0.5) "head" else "tail"
print(x)
```

```
## [1] "head"
```

Moreover,

```
if (logical_condition) expression_TRUE
```

is equivalent to

```
if (logical_condition) expression_TRUE else invisible(NULL)
```

An example:

```
sgn <- function(x) {
  stopifnot(is.numeric(x), length(x) == 1)
  if (x > 0) "positive"
  else if (x < 0) "negative"
  else "zero"
  # here, retval of if == retval of the function
}
unlist(lapply(c(1, 0, -2, 4), sgn))
```

```
## [1] "positive" "zero"     "negative" "positive"
```

11.2.3 Specifying the logical condition

Recall that `&` and `|` are element-wise vector operations denoting logical conjunction and alternative, respectively. The `&&` and `||` operators may be applied only to vectors of length one. They only evaluate their second argument if necessary, e.g. `TRUE ||` whatever, `FALSE &&` whatever.

```
FALSE || {cat("!"); TRUE}
```

```
## !
## [1] TRUE
```

```
TRUE || {cat("!"); TRUE}
```

```
## [1] TRUE
```

Also, when specifying the `logical_condition`, `all()` and `any()` are your friends. For example, `stopifnot(cond)` is equivalent to:

```
if (any(is.na(cond)) || !all(cond)) stop("error message")
```

11.2.4 return()

`return()` may be used only within a function. It stops the function's evaluation immediately and returns a given value. A note to C/C++ programmers: `return()` is a function; parentheses are required.

An example: quick sort (Hoare, 1960). It is a **recursive** sorting algorithm.

- A sequence of length 1 is already sorted.
- In order to sort a sequence `x` of length > 1, we:
  - Pick a *pivot* element v from `x`.
  - Return a sequence consisting of **sorted** elements $< v$, elements $= v$, **sorted** elements $> v$.

By the way, `options("expressions")` determines the maximal number of nested fun calls. Exceeding this limit results in `Error: evaluation nested too deeply: infinite recursion`.

```
qs <- function(x) {
  stopifnot(is.atomic(x), is.vector(x))
  if (length(x) <= 1) # already sorted
    return(x)
  pivot <- sample(x, 1) # random element of x
  c(qs(x[x<pivot]), x[x==pivot], qs(x[x>pivot]))
}
qs(c(5, 1, 4, 2, 3))
```

```
## [1] 1 2 3 4 5
```

```
qs(c("a", "e", "d", "c", "b"))
```

```
## [1] "a" "b" "c" "d" "e"
```

11.2.5 ifelse()

The `ifelse()` function is a vectorized version of `if..else`. We can use it as follows:

```
ifelse(test, values_TRUE, values_FALSE)
```

Some examples:

```
x <- c(5, 3, 1, 2, 4)
ifelse(x < 3, -x, x^2)
```

```
## [1] 25  9 -1 -2 16
```

```
ifelse(x > 3, NA, x)
```

```
## [1] NA  3  1  2 NA
```

Missing values in test give missing values in the result:

```r
x <- c(-1, 0, NA, 1)
ifelse(x < 0, -x, x) # NAs handled correctly
```

```
## [1]  1  0 NA  1
```

Possible pitfalls:

```r
x <- c(-1, 0, 1, 2)
ifelse(x >= 0, sqrt(x), NA) # sqrt is evaluated on -1 anyway
```

```
## Error in sqrt(x): (przetworzone z ostrzeżenia) wyprodukowano wartości NaN
```

11.3 Repetitive execution

`R` loop expressions allow for repetitive execution of expressions, possibly on different input data. There are three loop expressions in `R`:

- `while`
- `repeat`
- `for`

Each `R` loop returns `invisible(NULL)`.

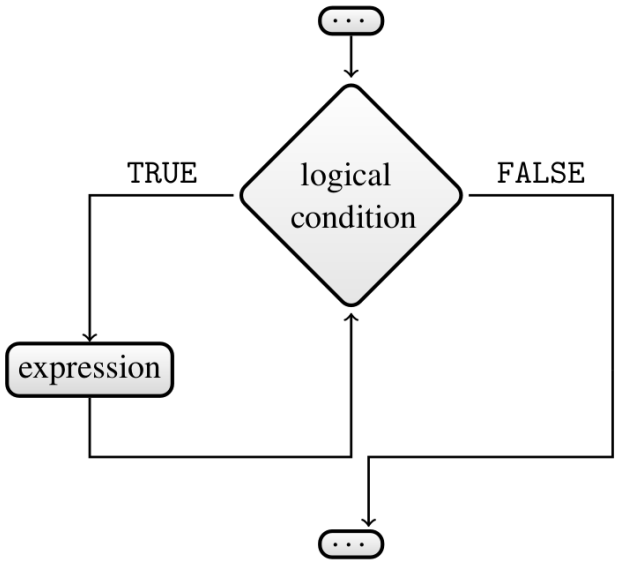
11.3.1 `while`

The `while` loop – syntax:

```r
while (logical_condition) expression
```

The `while` loop evaluates an expression as long as `as.logical(logical_condition)` is `TRUE`. In order to avoid an infinite loop, the `logical_condition` should e.g. depend on a variable that is modified by the expression.

Here is the corresponding program control flow diagram:



For example, this is a possible implementation of `sum()`:

```r
sum_while <- function(x) {
  x <- as.numeric(x) # coercion not possible => error
  result <- 0
  i <- 1
  n <- length(x)
  while (i <= n) { # logical_condition depends on i
    result <- result + x[i]
    i <- i + 1 # i changes here
  }
  result # return value
}
sum_while(1:5)
```

```
## [1] 15
```

```r
sum_while(c(1, 2, NA, 4, 5))
```

```
## [1] NA
```

11.3.2 `break` and `next`

The `break` expression immediately breaks out an (innermost) `R` loop.

```r
i <- 0
while (TRUE) { # infinite loop?
  j <- 0
  while (TRUE) {
    j <- j+1
    if (j > i) break
    print(c(i, j))
  }
  i <- i+1
  if (i > 3) break
}
```

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

The next expression immediately advances to the next loop iteration.

```r
i <- 0
while (i < 6) {
  i <- i+1
  if (i %% 2 == 0) next
  print(i)
}
```

```
## [1] 1
## [1] 3
## [1] 5
```

Of course the two above examples could be rewritten in a much simpler way.

11.3.3 `repeat`

`repeat` – syntax:

```
repeat expression
```

is equivalent to

```
while (TRUE) expression
```

Thus, we need an explicit call to e.g. `break` or `return()` in the expression.

11.3.4 `for`

`for` – syntax:

```
for (name in vector) expression
```

This is roughly equivalent to:

1. `name <- vector[[1]]; expression`
2. `name <- vector[[2]]; expression`
3. `...`
4. `name <- vector[[length(vector)]]; expression`

Here is our 2nd implementation of `sum()`:

```
sum_for1 <- function(x) {
  x <- as.numeric(x) # coercion not possible => error
  result <- 0
  for (elem in x)
    result <- result+elem
  result # return value
}
sum_for1(1:5)
```

```
## [1] 15
```

```
sum_for1(c(1, 2, NA, 4, 5))
```

```
## [1] NA
```

And our 3rd implementation of `sum()`:

```
sum_for2 <- function(x) {
  x <- as.numeric(x) # coercion not possible => error
  result <- 0
  for (i in seq_along(x))
    result <- result+x[i]
  result # return value
}
sum_for2(1:5)
```

```
## [1] 15
```

See `?seq_along`. Why would `1:length(x)` be wrong here? Try calling `sum_for2(numeric(0))`.

Our implementation of `lapply()`:

```
lapply_for <- function(x, f, ...) {
  stopifnot(is.vector(x))
  f <- match.fun(f) # see ?match.fun
  result <- vector("list", length(x)) # preallocate
  for (i in seq_along(x)) result[[i]] <- f(x[[i]], ...)
  result
}
lapply_for(list(1:5, 2:6), "*", 2)
```

```
## [[1]]
## [1]  2  4  6  8 10
##
## [[2]]
## [1]  4  6  8 10 12
```

11.4 Summary

R loops are very slow:

```
x <- runif(100000)
microbenchmark::microbenchmark(unit="relative",
                               sum_while(x), sum_for1(x), sum_for2(x), sum(x))
```

```
## Unit: relative
##          expr      min       lq     mean   median       uq      max neval  cld
##  sum_while(x) 86.86624 78.59631 75.29144 75.65617 73.51862 39.09382   100    d
##   sum_for1(x) 26.55078 24.20433 23.15223 23.11012 22.55084 12.46391   100  b
##   sum_for2(x) 45.39767 41.17023 39.31792 39.37820 38.29419 19.88453   100   c
##        sum(x)  1.00000  1.00000  1.00000  1.00000  1.00000  1.00000   100 a
```

We should rather rely on vectorized R functions. But it does not mean that we must always avoid loops at any cost – some tasks cannot be implemented without them.

11.5 Bibliography

- R Core Team, *{An introduction to R}*, 2014, Sec. 9
- R Core Team, *Writing R extensions*, 2014, Sec. 3, 4