Module 1 (R basics part I)

Contents

HUMAN CAPITAL
NATIONAL COHESION STRATEGY

EUROPEAN UNION
EUROPEAN
SOCIAL FUND

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

1 Introduction to R

1.1 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 ($\neq$ 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.

1.2 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.

The course is divided into 8 modules. Its duration is ca. 16 weeks (2 weeks per module). Here is the course's outline.

Part I: R basics

Module 1.

- Getting started with RStudio
- Classification of R data types. Basic atomic types
- Basic vector operations

Module 2.

- Lists a.k.a. generalized vectors
- Functions (closures)
- Unit testing. Debugging. Exception handling

Module 3.

- Attributes
- Compound types
- Control flow expressions
- Run-time measurement and estimation

Part II: Character string processing

Module 4.

- Strings representation
- Basic string processing and searching tasks
- Regular expressions
- Date and time

Part III: File processing

Module 5.

- Basic operations on files and directories
- Text files and connections
- Common file formats
- Markdown & knitr

Part IV: Advanced R programming

Module 6.

- Environments
- Computing on the language
- Object-oriented programming: S3 & S4
- UNIX-like command line
- Collaborative software development with git
- Writing R packages

Part V: Rcpp – seamless R and C++ integration

Module 7.

- Rcpp intro
- C++ primer
- Rcpp basic types
- R/C API

Module 8.

- Non-basic types
- C++11
- STL
- Rcpp in R packages

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).

And here are the grading rules. There will be 6 homework assignments, a couple of tasks each (60%). Note that each homework must be submitted via courses.ipipan.edu.pl. Moreover, there will be a final exam in the form of a multiple-choice test (40%). At least 50% points is required to pass.

1.3 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.

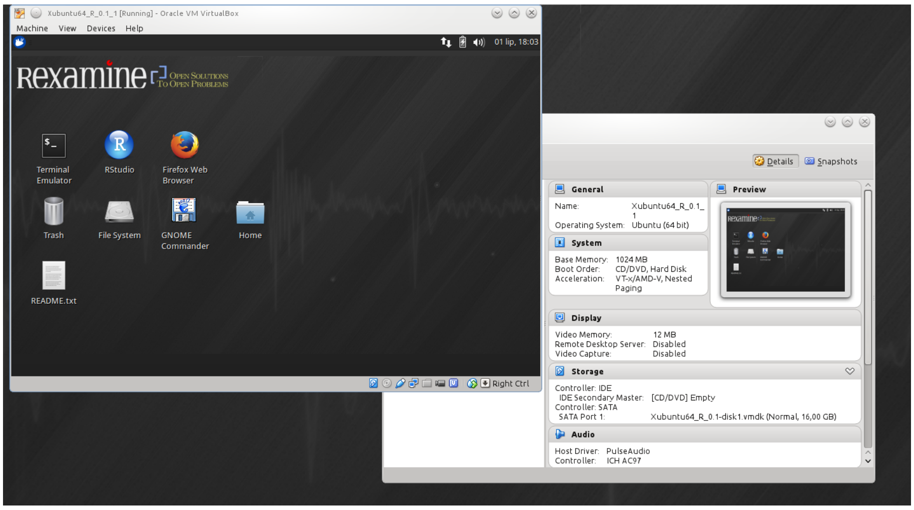
As some of you may find installing all the required tools problematic, I have prepared a virtual appliance that provides everything you need to develop R software/packages (including a working C++11-compliant compiler, a LaTeX distribution, etc.). At least 1GB of free RAM and 8GB of free disk space is required.

1. Install Oracle VirtualBox (runs on Windows, OS X, and Linux)
2. Download the virtual appliance archive (`.ova` file, ca. 3 GB)
3. Import the `.ova` file into VirtualBox
4. Start the virtual machine

You will quickly feel like home with the user-friendly Xubuntu operating system, cf. Fig. 1.

1.4 Bibliography

- Gągolewski M., *Programowanie w języku R*, PWN, 2014 (in Polish).
- *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.



General
Name: Xubuntu64_R_0.1_1
Operating System: Ubuntu (64 bit)

System
Base Memory: 1024 MB
Boot Order: CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging

Display
Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage
Controller: IDE
IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
SATA Port 1: Xubuntu64_R_0.1-disk1.vmdk (Normal, 16,00 GB)

Audio
Host Driver: PulseAudio
Controller: ICH AC97

Figure 1: Xubuntu_R, see gagolewski.rexamine.com/resources/vms/

- Matloff N., *The art of R programming*, No Starch Press, 2011.

2 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.

2.1 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/[1]. 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:

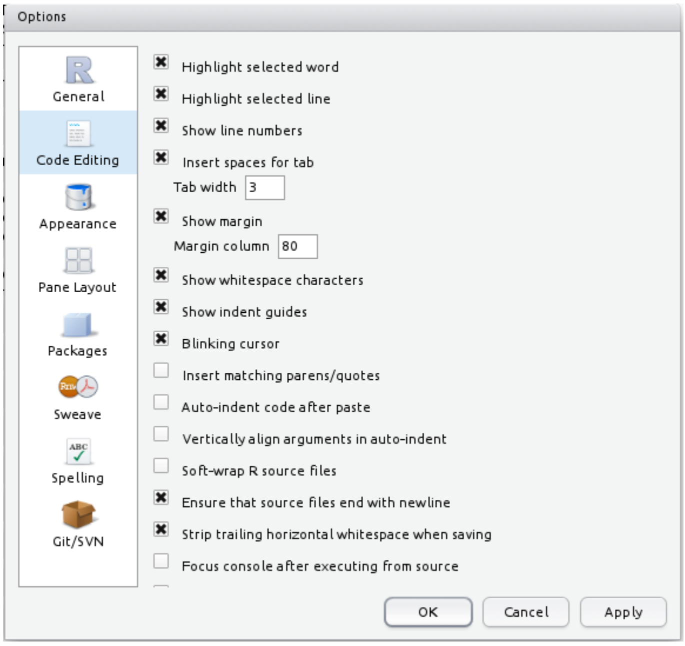
1. A built-in R console,
2. R, LaTeX, HTML, C++ syntax highlighting, R code completion,
3. Easy management of multiple projects,
4. Integrated R documentation,
5. Interactive debugger,
6. Package development tools.

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. 2.

[1] Advanced Linux users/server administrators may also be interested in installing RStudio Server, available at www.rstudio.com/products/rstudio/download-server/.



- ☒ Highlight selected word
- ☒ Highlight selected line
- ☒ Show line numbers
- ☒ Insert spaces for tab
- Tab width 3
- ☒ Show margin
- Margin column 80
- ☒ Show whitespace characters
- ☒ Show indent guides
- ☒ Blinking cursor
- □ Insert matching parens/quotes
- □ Auto-indent code after paste
- □ Vertically align arguments in auto-indent
- □ Soft-wrap R source files
- ☒ Ensure that source files end with newline
- ☒ Strip trailing horizontal whitespace when saving
- □ Focus console after executing from source

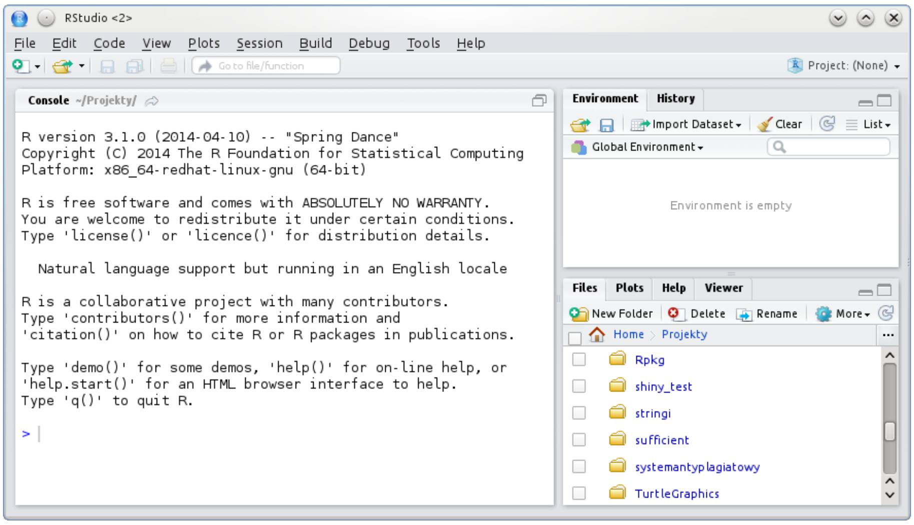
Figure 2: Recommended RStudio code editing options.

2.2 R in the interactive mode

Type in the R console (see Fig. 3):

```
> 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.



```
R version 3.1.0 (2014-04-10) -- "Spring Dance"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-redhat-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 3: RStudio is ready for use.

You should have obtained the following results:

```
> 2 + 2   # ENTER
## [1] 4
> R.version.string   # ENTER
```

```
## [1] "R version 3.1.0 (2014-04-10)"
```

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
```

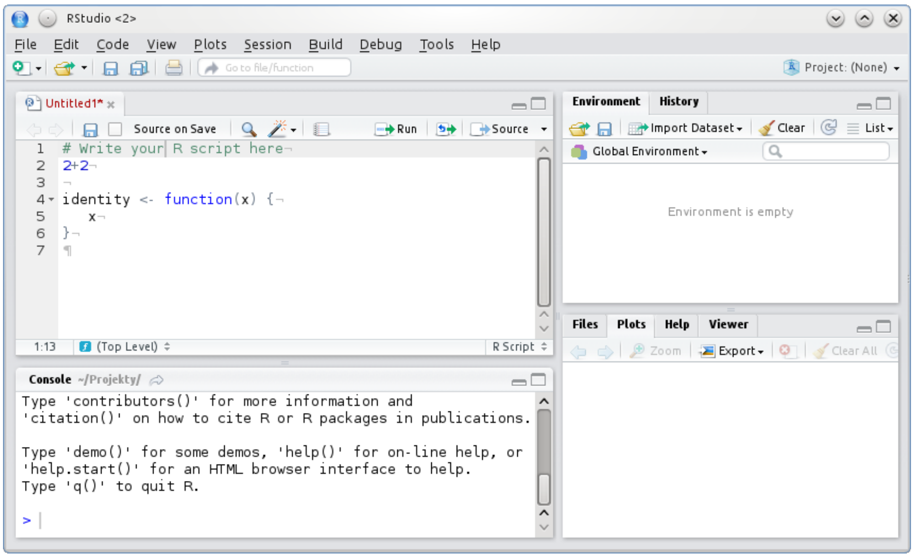
Also note that "+" often appears when there is no matching, `"`, `'`, `}`, or `)`. For example:

```
> 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.

2.3 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. 4.



```
# Write your R script here
2+2

identity <- function(x) {
    x
}
```

```
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 4: 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.

2.4 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:

```
[gagolews@eurydike ~]$ ./test.R
2 + 2 =  4
```

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

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

2.5 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 Fig. 5 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")
```

rep {base} R Documentation

(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.)

Figure 5: Excerpts from the `?rep` man page.

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.

2.6 Keyboard Shortcuts

Many tasks can be done much more effectively without the mouse – You should use your keyboard as often 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.

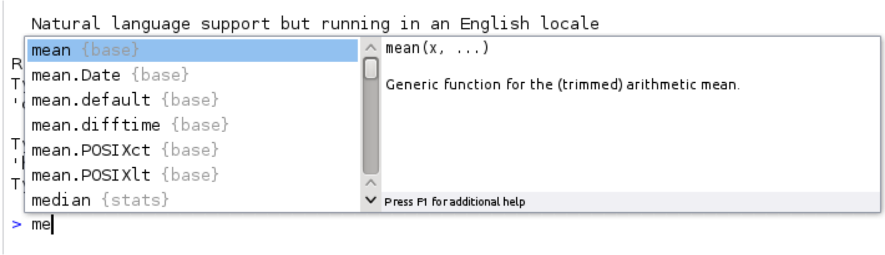
Moving the caret:

- ⟨↑⟩, ⟨↓⟩, ⟨←⟩, ⟨→⟩
- ⟨`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. 6.



```
Natural language support but running in an English locale
mean {base}              mean(x, ...)
mean.Date {base}
mean.default {base}      Generic function for the (trimmed) arithmetic mean.
mean.difftime {base}
mean.POSIXct {base}
mean.POSIXlt {base}
median {stats}           Press F1 for additional help
> me
```

Figure 6: 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, ⟨TAB⟩ 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.

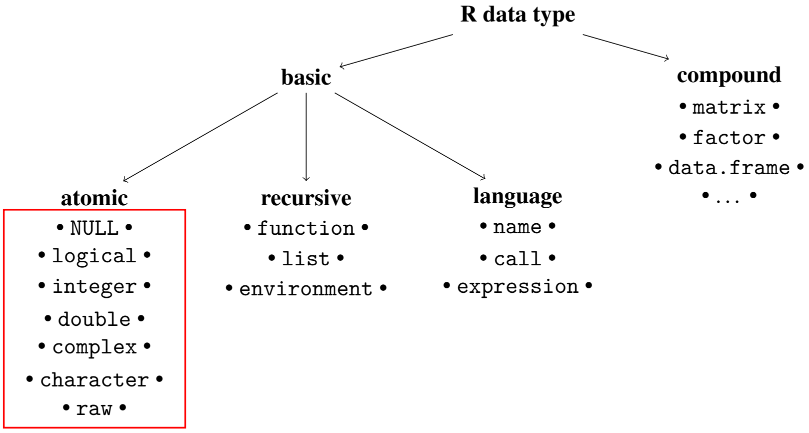
2.7 Summary

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.

3 Atomic types in R

3.1 R data types

Fig. 7 presents one of the possible ways to classify R data types.



Also: S4, externalptr, weakref

Internal (not really interesting): pairlist, promise, special, builtin, char, ..., bytecode

Figure 7: 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.

3.2 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. $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$. Table 1 gives the types of atomic vectors available in R.

Table 1: Atomic vectors in R

Basic type	**Elements**
`logical`	$(\mathbb{B})$; $= \{$`TRUE`, `FALSE`$\}$
`integer`	$(\mathbb{Z})$; $= \{-2\,147\,483\,647, -2\,147\,483\,646, \dots, 2\,147\,483\,647\}$
`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`$, \dots,$ `ff`$\}$

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

3.2.1 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:  object 'false' not found
TypeOf   # wrong
## Error:  object 'TypeOf' not found
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()` **c**ombines a given sequence of values and returns a vector:

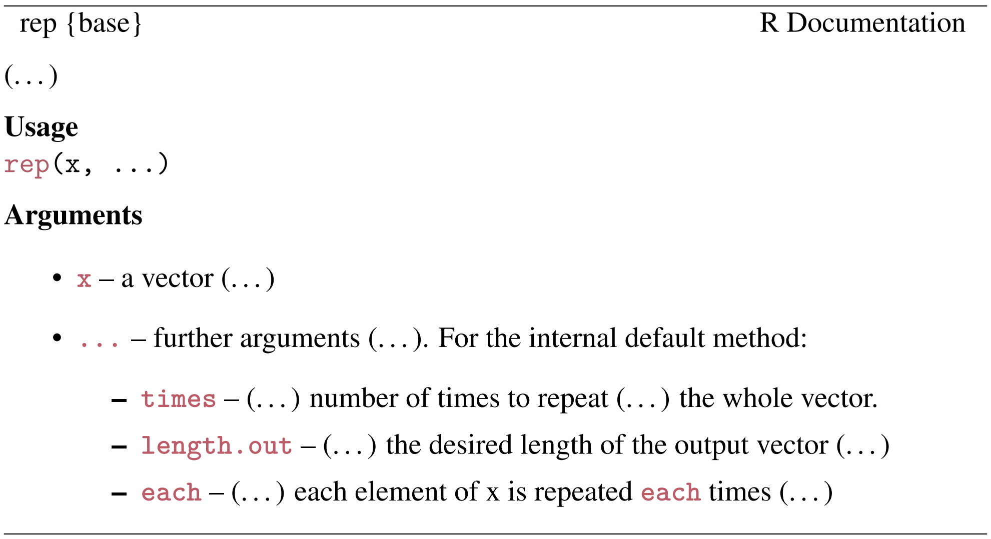
```
c(TRUE, FALSE, TRUE)
## [1]  TRUE FALSE  TRUE
c(FALSE)   # the same as: FALSE
## [1] FALSE
length(c(TRUE, FALSE, TRUE))
## [1] 3
```

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
```

Exercise: *Consult* `?rep` *now.*



rep {base} R Documentation

(. . .)

Usage

`rep(x, ...)`

Arguments

- `x` – a vector (. . .)
- `...` – further arguments (. . .). For the internal default method:
  - `times` – (. . .) number of times to repeat (. . .) the whole vector.
  - `length.out` – (. . .) the desired length of the output vector (. . .)
  - `each` – (. . .) each element of x is repeated `each` times (. . .)

Figure 8: Excerpts from the `?rep` man page.

In the `rep()` function man page (cf. Fig. 8) 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
## [12] FALSE
rep(c(TRUE, FALSE), length.out = 8, each = 2)
```

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

3.2.2 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:

```
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:  NAs produced by integer overflow
## [1] NA
```

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

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

```
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 $\varepsilon = 1\mathrm{e} - 8$.

```
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
-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
```

Exercise: *The behavior of the* `seq_along()` *and* `seq_len()` *functions may easily be imitated by calling* `seq()`*. How can we do it?*

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

3.2.3 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"
```

3.2.4 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 2 lists the most interesting control sequences, see also `?Quotes`.

3.2.5 Raw vectors

The vectors of type `raw` consists of single bytes, i.e. integer $\in \{0, 1, \dots, 255\}$. 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.

Table 2: Escape characters in R

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

3.2.6 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 $\neq 0$ 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.

3.3 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 Not *Available*.

```
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:  NAs introduced by coercion
## [1] NA
```

[4] Except in `raw` vectors.

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

A general rule is: operations applied on NAs 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 NaNs (*not a number*).

```
c(sqrt(-1), 0^0)
## Warning:  NaNs produced
## [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 $\not\in$ {NA, NaN, Inf, −Inf}.

```
is.finite(c(1, NA, NaN, Inf, -Inf))
## [1]  TRUE FALSE FALSE FALSE FALSE
```

3.4 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
```

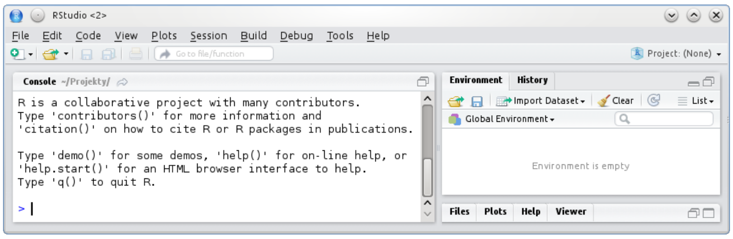
Also, NULLs are used e.g. as *empty* lists' elements, etc.

3.5 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:



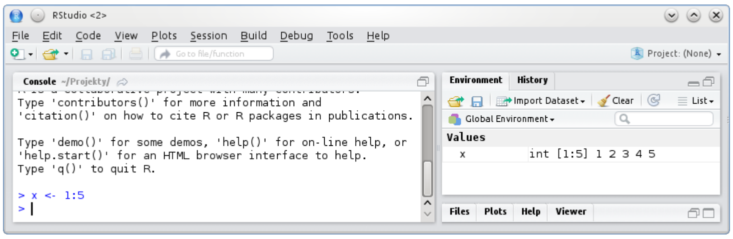
```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Environment is empty

After:



```
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> x <- 1:5
> |
```

```
Values
  x            int [1:5] 1 2 3 4 5
```

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`.

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

```
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
```

3.6 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

3.7 Bibliography

- R Core Team, *An introduction to R*, 2014, Sec. 2
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 2
- Goldberg D., What every computer scientist should know about floating-point arithmetic, *ACM Computing Surveys* 21(1), 1991, pp. 5-48.

4 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.

4.1 Operators

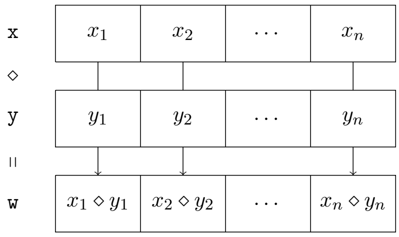
4.1.1 Arithmetic operators

Table 3 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 3: Arithmetic operators in R

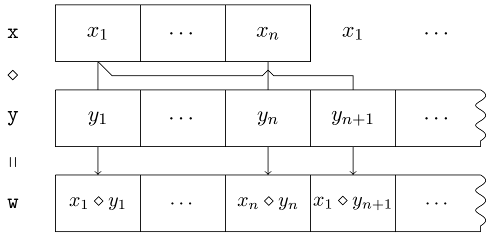
Operation	Meaning
`x + y`	addition
`x - y`	subtraction
`x * y`	multiplication
`x / y`	division
`x ^ y`	exponentiation (also: `x ** y`)
`x %% y`	division remainder (modulo)
`x %/% y`	integer division

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:  longer object length is not a multiple of shorter object length
## [1]  1 -2  3 -4  5 -6  7
```

When needed, operands' type coercion is applied automatically:

```
typeof(c(1L, 2L) + c(1, 2))  # integer + double
## [1] "double"
c(TRUE, FALSE) * 1:6  # TRUE -> 1, FALSE -> 0
## [1] 1 0 3 0 5 0
c("1", "2") + c("2", "3")  # use as.numeric() here explicitly
## Error:  non-numeric argument to binary operator
```

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 NAs 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
```

4.1.2 Logical operators

Table 4 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 4: 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)  # recycling
## [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
```

4.1.3 Comparison operators

Table 5 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
```

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]:

```
"b" <= c("a", "b", "c")
## [1] FALSE  TRUE  TRUE
```

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

Table 5: 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

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

4.2 Selecting and modifying subsets of vectors

4.2.1 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:

- positive integer quantities,
- negative integer quantities,
- logical values,
- character strings (see Module 2).

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:  only 0’s may be mixed with negative subscripts
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] 358 808 467 892 942 140 572 896 591 506
x[x > 500]  # select only elements > 500
## [1] 808 892 942 572 896 591 506
x[x >= 250 & x <= 750]
## [1] 358 467 572 591 506
```

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"
```

4.2.2 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”:

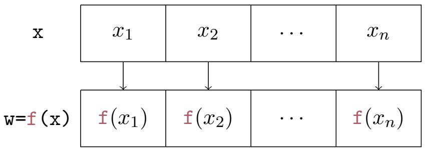
```
(x <- 1:6)
## [1] 1 2 3 4 5 6
x[length(x)+1] <- 7; x
## [1] 1 2 3 4 5 6 7
x[10] <- 10; x
##  [1]  1  2  3  4  5  6  7 NA NA 10
```

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).

4.3 Built-in functions

4.3.1 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:

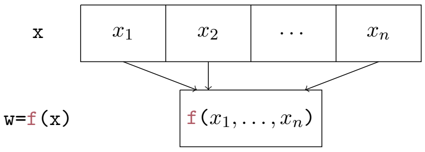
```
abs(c(1, -1, 2, -2, 0))  # vectorized
## [1] 1 1 2 2 0
round(3.141592, 3)
## [1] 3.142
```

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

```
round(3.141592)  # equiv. round(3.141592, 0) if `digits` omitted
## [1] 3
```

4.3.2 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.



```
x <- c(1, 5, 4, NA, 3)
sum(x)
## [1] NA
sum(x[!is.na(x)])  # cf. sum(x, na.rm=TRUE)
## [1] 13
sum(is.na(x))  # how many missing values?
## [1] 1
any(is.na(x))
## [1] TRUE
all(x == 1:5)
## [1] FALSE
```

Exercise: *Let* x *be a numeric vector (e.g. "*x <- round(rnorm(20, 0, 1), 2)*").*

- *Print all values* $\in [-2,-1] \cup [1,2]$.
- *Print the number of non-negative values.*

- *Calculate the value that is closest to 0.*
- *Calculate the value that is most distant to 2.*
- *Print the fractional parts of each number.*
- *Apply a linear transformation of elements in* `x` *to obtain values* $\in [0, 1]$.
- *Create a character vector* `y`*;* y_i *should be* $\in$*{"negative", "zero", "positive"}, depending on* x_i.

Exercise: *Let* `x`, `y` *– numeric vectors of length* n*. Calculate the Pearson correlation coefficient,*

$$r(\mathtt{x}, \mathtt{y}) = \frac{1}{n-1} \sum_{i=1}^{n} \frac{x_i - \bar{\mathtt{x}}}{s_{\mathtt{x}}} \frac{y_i - \bar{\mathtt{y}}}{s_{\mathtt{y}}},$$

where $\bar{\mathtt{x}}, \bar{\mathtt{y}}$ *– arithmetic means, and* $s_{\mathtt{x}}, s_{\mathtt{y}}$ *– standard deviations, of* `x` *and* `y`*, respectively.*

Test your code with:

- `x <- rnorm(20, 0, 1); y <- 10*x+2`,
- `x <- rnorm(20, 0, 1); y <- -4*x+1`,
- `x <- rnorm(2000, 0, 1); y <- rnorm(2000, 5, 2)`.

Note that $r(\mathtt{x}, \mathtt{y}) \in [-1, 1]$.

Exercise: *The arcsine function may be approximated with the Taylor series:* $\arcsin x \simeq \sum_{n=0}^{m} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1}$, *where* $x \in (-1, 1)$ *and* $m \in \mathbb{N}$.

Given `x` *and* $m = 10, 100, 1000$*, calculate the difference between your approximation and the value returned by* `asin()`.

Exercise: *Let's take two numeric vectors of length* n*:* `x` *(increasing) and* `p` *(with non-negative elements and such that* $\sum_{i=1}^{n} p_i = 1$*). The pair* $(\mathtt{x}, \mathtt{p})$ *defines a discrete probability distribution of a random variable* X*, with* $\Pr(X = x_i) = p_i$.

The expected value of X *is given by* $\mathbb{E}\,X = \sum_{i=1}^{n} x_i\, p_i$*, variance by* $\mathrm{Var}\,X = \mathbb{E}\,X^2 - (\mathbb{E}\,X)^2$*, where* $\mathbb{E}\,X^2 = \sum_{i=1}^{n} x_i^2\, p_i$*, and standard deviation by* $\sigma_X = \sqrt{\mathrm{Var}\,X}$.

Moreover, the cumulative distribution function of X *is given by* $F(y) = \Pr(X \le y) = \sum_{i: x_i \le y} p_i$.

Jose takes parts in a municipal lottery. Here are the possible outcomes together with respective probabilities:

x_i	*0*	*10*	*25*	*100*
p_i	*80%*	*15%*	*4%*	*1%*

Calculate the expected prize and its standard deviation.

Moreover, calculate the c.d.f. of this RV at -10, 10, 50, *and* 200.

4.3.3 Other functions

Cumulative sum, product, minimum, and maximum:

```
x <- c(4, 2, 3, 5, 1)
cumsum(x)
## [1]  4  6  9 14 15
```

```
cumprod(x)
## [1]   4   8  24 120 120
```

```
cummin(x)
## [1] 4 2 2 2 1
```

```
cummax(x)
## [1] 4 4 4 5 5
```

Iterated difference:

```
diff(x)  # lag=1 by default, see ?diff
## [1] -2  1  2 -4
```

`which` family:

```
x <- c(40, 20, 30, 50, 20)
which(x > 3)
## [1] 1 2 3 4 5
```

```
which.min(x)  # like which(x == min(x))[1], but faster
## [1] 2
```

```
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
```

```
order(x)  # ordering permutation
## [1] 2 5 3 1 4
```

```
x[order(x)]  # same as sort(x)
## [1] 20 20 30 40 50
```

```
sample(x)  # random permutation
## [1] 20 20 30 50 40
```

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

Exercise: *Let* `x` *be a numeric vector of even length. Calculate its minimum, maximum, and median without calling* `median()`, `min()`, `max()`, *or* `quantile()`.

Exercise: *Let* `x` *be a numeric vector of odd length* n. *Also, let* `k` $\le \frac{n-1}{2}$. *Calculate the* k*-trimmed mean, i.e. the arithmetic mean of all but* `k` *smallest and* `k` *largest elements in* `x`.

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()`.

Exercise: *Let* `x`, `y` *– vectors of positive integers. Using the* `tabulate()` *function, write a chunk of code that gives the same result as* `intersect(x,y)`.

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.

4.4 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.

4.5 Bibliography

- R Core Team, *An Introduction to R*, 2014, Sec. 2
- Gagolewski M., *Programowanie w jezyku R*, PWN, 2014 (in Polish), Chap. 3
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 2