Using R for data analysis

2023-05-24

Contents

About this course

This course teaches the basics of R, an open-source and free environment for statistical analyses. In this course we also teach the basics transparent and reproducible research. For this, we teach R Markdown, a tool to make dynamic reports in R.

R is an open-source, free environment for statistical computing and graphics. It provides a large repository of statistical analysis methods, both classic and new. However, R has a steep learning curve, due partly to its using a command-line type of user interface, rather than the usual pull-down menus. The course aims at helping researchers climb this curve, enabling them to perform basic data analysis and graphic displays at the end of the course, as well as giving a platform from which they can deepen their R knowledge later on if necessary.

Teachers

- Renee Menezes, Biostatistics Center, NKI (**coordinator**)
- Renaud Tissier, Biostatistics Center, NKI
- Leyla Azarang, Biostatistics Center, NKI

Goals & Topics

After the course you will be able to:

- understand and write simple R scripts
- use R to perform simple statistical analyses of your own data
- use and adapt R scripts and functions
- generate analysis reports from your own data in html format

We will cover the following topics:

- R expressions
- R data objects (vectors (arrays), data frames (tables), lists) creation and usage
- R Markdown for building reproducible reports [cheat sheet]

- R functions for descriptive statistics and linear model fitting; R formula objects
- histograms, scatter plots, boxplots (in basic R)

Prerequisites

The course assumes no prior programming knowledge. Elementary statistics knowledge is necessary.

Participants must **bring their own laptops** capable of running RStudio.

Before the course **please prepare your laptop**:

1. install R, an open-source, free environment for statistical computing and graphics. You can find instructions for downloading and installing it from one of the CRAN mirrors, for example from the Univ. of Gent or from the Imperial College. A full list of mirrors can be found here.

2. install RStudio. Go to the RStudio download page, select a version of RStudio appropriate for your laptop, download it and then install. Please check whether you can start RStudio.

3. install RMarkdown, a very nice and easy tool to produce reports using RStudio. It is made available as an R package for Rstudio. One easy way to install it is as follows:

i) open RStudio

ii) click on the "File" menu on the top left, and choose "New file">"R Markdown". If RMarkdown is not yet installed on your machine, this will prompt you to install it and any packages required. Just follow the instructions that appear on the screen.

4. download the course materials from the [github repository] (https://github.com/rxmenezes/IntroductionToRCourse/), by looking for the .zip file and dowloading to your laptop. Unpack the zip file to a folder. This will be your course folder.

Materials

The material in your course folder can be assessed in two ways:

- by clicking on the `RcourseNKI.Rproj` file on the root folder, which will open the entire course as an R project within your RStudio. This will be the handiest way during the lectures. The material for each chapter/day of the course is then available via a file with extension .Rmd, as follows: `01-basicR.Rmd`, `02-datatypes1.Rmd`, `03-datatypes2.Rmd` and `04-functions.Rmd` for chapters 1-4, respectively. We will give lectures, and you will follow them, by clicking on each one of these files at a time.

- as a HTML page by opening the file `index.html` from the `_book` folder in any browser. This gives an overview of the course material, and is therefore very useful for later reference.

The materials contain also a `data` folder with the **data files** used in the presentations/tasks. The directory can be also accessed at https://github.com/rxmenezes/IntroductionToRCourse/

Programme

0.0.1 Fifth NKI edition, June 5th, 6th, 8th, and 9th 2023

This course will be given live at room Z4. All course days are in the period 9:00-16:30.

Chapter 1

Basics of R

1.1 Data sets

Throughout the course we will use the two data sets described below.

1.1.1 Pulse

Students in an introductory statistics class (MS212 taught by Professor John Eccleston and Dr Richard Wilson at The University of Queensland) participated in a simple experiment. The students measured their own pulse rate. They were then asked to flip a coin. If the coin came up heads, they were to run in place for one minute. Otherwise they sat without movement for one minute. Then everyone measured their pulse again. The pulse rates and other physiological and lifestyle data are given in the data table.

Variable	Explanation
name	Name of a participant
height	Height (cm)
weight	Weight (kg)
age	Age (years)
gender	Sex (male/female)
smokes	Regular smoker? (yes/no)
alcohol	Regular drinker? (yes/no)
exercise	Frequency of exercise (high/moderate/low)
ran	Whether the student ran or sat between the first and second pulse measurements (ran/sat)
pulse1	First pulse measurement (rate per minute)
pulse2	Second pulse measurement (rate per minute)
year	Year of the class (1993 - 1998)

The pulse data set is available in the data folder as tab-delimited text: `pulse.txt`. It is also available in SPSS-format as `pulse.sav`.

1.1.2 Survey

This data frame contains the responses of 233 Statistics I students at the University of Adelaide to a number of questions. It is a slightly modified version of the `survey` data from the `MASS` pacakge.

Variable	Explanation
name	Name of a participant
gender	Sex (male/female)
span1	Span (distance from tip of thumb to tip of little finger of spread hand) of writing hand (cm)
span2	Span of non-writing hand (cm)
hand	Writing hand of student (left/right)
fold	Fold your arms! which is on top? (right/left/neither)
pulse	Pulse measurement (rate per minute)
clap	Clap your hands! which is on top? (right/left/neither)
exercise	Frequency of exercise (freq/some/none)
smokes	How much the student smokes (heavy/regul/occas/never)
height	Height (cm)
m.i	whether the student expressed height in imperial (feet/inches) or metric (centimetres/metres) units. (metric/imperial)
age	Age of the student (years)

The pulse data set is available in the data folder as tab-delimited text: `survey.txt`.

1.2 Introduction

1.2.1 Why this course?

Modern science requires modern statistical methods:

- Genomics/bioinformatics
- Advanced survival data analysis
- Causal modeling
- …

Statisticians make methods are made available as packages in R.

No need to wait until they are programmed into SPSS.

1.2.2 What is R?

1.2.2.1 *R*: a short history

S: a programming language for statistics by John Chambers (Bell Labs) in the 1970s-80s.

Two implementations:

- S-plus (1988, commercial)
- R (1993, GNU public license) by Robert Gentleman and Ross Ihaka (Univ. of Auckland)

The R Foundation for Statistical Computing. Non-profit.

Huge community (users, package writers).

1.2.2.2 Open source

Free software.

Volunteer work (mostly by academics).

Anyone can see the source code.

Anyone can contribute:

- write code
- report bugs
- write documentation

1.2.2.3 Obtaining R

CRAN: Comprehensive R Archive Network

- Repository for R and packages
- Go to http://www.r-project.org

Free download

- New major version (R 3.2.0) every year
- New minor versions (R 3.2.4) in between

Also on CRAN

- Manuals (don't read them)
- Mailing lists + archives (well indexed on google)

1.2.3 R and RStudio

RStudio: open source integrated development environment to R (2011)

Adds useful features to help write code and organize projects

Not necessary to use R, but highly recommended

RStudio organises input and output in useful windows

1.2.4 Course overview

1.2.4.1 What we teach

This is not a statistics course!

To learn about statistics, follow the *Medical Statistics course* (or more advanced courses).

This course teaches R proficiency:

- The mechanics of R
- How to use other people's R scripts
- How to write your own R scripts
- How to use R packages

Focus: R as a *language* for data analysis.

1.2.4.2 Course structure

- Interactive lectures
- Practice time in between

Hands-on at four levels:

1. *Type with me*
2. Mini-exercises briefly interrupt the lecture
3. Longer exercises to do it on your own
4. Advanced exercises introduce advanced concepts for quick learners (optional)

Eight half-day slots.

1.3 Basics

1.3.1 R as a calculator

1.3.1.1 Calculations

At the prompt > you can do any calculations you like. Press enter to see the result.

```
2*8
```

```
## [1] 16
```

```
4+5
```

```
## [1] 9
```

```
2/8
```

```
## [1] 0.25
```

```
5^2  # ^ = to the power
```

```
## [1] 25
```

Note: decimals always with `.`, never `,`.

1.3.1.2 Parentheses

Use parentheses as much as possible to make sure the calculations are done in the right order.

```
12/2*3
```

```
## [1] 18
```

```
12/(2*3)
```

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

RStudio will automatically insert a closing parenthesis. If you don't like this, change using Tools…Global options…Code…Editing.

1.3.1.3 Arithmetic functions

Useful functions.

```
sqrt(10)              # square root
```

```
## [1] 3.162278
```

```
log(10)               # natural logarithm
```

```
## [1] 2.302585
```

Terminology: the *function* (e.g. `log`) is applied to its *argument* (e.g. `10`). The argument of a function is always between parentheses.

Other useful functions

- `log2` (logartithm base 2)
- `log10` (logarithm base 10)
- `abs` (absolute value)

1.3.1.4 Multiline commands

Use up/down arrow to retrieve previous/next commands. Use Ctrl-R to see command history and type letters to select a line.

Getting a `+` as a prompt means the command is not finished yet. Continue typing or press *Esc*.

```
5*(1+1
)
```

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

1.3.1.5 (*) Integer division and remainders

The remainder of one number after division by another.

```
17 %/% 5    # integer division
```

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

```
17 %% 5     # remainder
```

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

`17 %/% 5` evaluates to 3 because `17 = 3*5+2`, so 5 fits into 17 3 times. `17 %% 5` evaluates to 2 because `17 = 3*5+2`, so 2 is the remainder of 17 when divided by 5.

1.3.2 Variables

1.3.2.1 Variable names

Variables store values or results of calculations. Choose the names of variables freely.

```
x <- 5
my_calculation <- 6 + x
```

To find out what the value of a variable is: type the name.

```
x
```

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

Rstudio has autocomplete (with `tab`). Useful for long variable names.

1.3.2.2 Legal variable names

Note that `_` and `.` are allowed in variable names. Numbers are allowed too.

Not allowed:

- names containing a space
- names containing a one of @#!%$^()-+=!~?,<>}{][‘
- for, while, repeat, if, else, TRUE, FALSE, Inf, NA, NaN (reserved names)
- a name that starts with a number

Variable names in R are case sensitive. Everything else too!

Choose meaningful variable names for readable code.

1.3.2.3 Assignment

Arrow is called "assignment". Also allowed: `=`.

```
x = 5
```

Assignments are needed to store a result. No assignment: printed to screen and lost.

```
x+1
```

```
## [1] 6
```

```
x
```

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

You have asked R what `x+1` is, but `x` did not change. To change the value of a variable, reassign.

```
x <- x + 1
x
```

```
## [1] 6
```

Remember: *no assignment, no change*

Important: variables are stored in memory, not on disk. If you close R, all variables are lost (if save workspace image = no)

Your RStudio has an *environment* tab that lists all the variables you made.

1.3.2.4 (*) The workspace

To list the variable(s) you have defined:

```
ls()
```

```
##  [1] "catLinkTaskSection" "catLoad"            "catReadCsv"
##  [4] "catReadDelim"       "catReadLines"       "catReadTable"
##  [7] "catSlot"            "catTopic"           "my_calculation"
## [10] "params"             "x"
```

Note that this is a function with no arguments.

To remove a variable from memory:

```
rm(x)
```

You only need to remove a variable from memory when:

1. The variable is large and you want to free memory

2. You have accidentaly overwritten one of R's fixed constants

Note that `rm` is definitive and you cannot *undo* it!

Quick task(s):

Solve the task(s), and check your solution(s) here.

1.3.3 Vectors

Vectors are the basic building blocks of the R language.

1.3.3.1 Vector basics

Variables can contain *vectors* of numbers. A vector can be e.g. just any sequence of numbers.

You can make a vector using `c` (combine):

```
x <- c(3, 6, 7, 2)
```

Calculations or functions often work on vectors elementwise. This is helpful to do many calculations simultaneously:

```
x^2
```

```
## [1]  9 36 49  4
```

```
x - 18
```

```
## [1] -15 -12 -11 -16
```

```
sqrt(x)
```

```
## [1] 1.732051 2.449490 2.645751 1.414214
```

Some functions summarize a vector to a single number:

```
sum(x)
```

```
## [1] 18
```

To find out the number of elements in a vector:

```
length(x)
```

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

1.3.3.2 Simple sequences

A simple regular sequence you can make with `:` (colon) operator:

```
y <- 1:10
7:9
```

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

1.3.3.3 Simple selection

To see only part of a vector use square brackets. Combine with `:` to select more than one element:

```
x[1]
```

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

```
x[3:4]
```

```
## [1] 7 2
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

1.4 Projects and scripts

1.4.1 Projects

We are going to read our first data set into R. To structure this, we are going to make a `project`.

1.4.1.1 Creating and opening a new project

Choose File…New project…New directory… and create a new folder on your computer for this R course. Make sure this new folder is NOT within the course folder, to avoid confusion.

RStudio has created a `.Rproj` file for you that you can use to open Rstudio. You end up in the right project immediately.

Projects are a feature of RStudio to:

- organize all files and scripts that you need for a single project
- keep files separate from other projects

From now on in this course, you are going to do exercises within the new R course project you created. This is to allow you to create scripts and make changes, which are discouraged in the course project to avoid problems.

Look top right in the RStudio window to check which project you are working in. There, you can also switch between projects.

1.4.1.2 Creating a folder

In our new project we are going to create a folder that contains our data sets

Choose the files tab and click `new folder` to create a `data` folder, or type at the prompt

```
dir.create('data')
```

From the course project, folder `data`, copy the data sets `pulse.txt` and `survey.txt` into the new data folder you created.

Check in Rstudio that the files are indeed in the right folder.

1.4.1.3 Quick exercises

- Close RStudio and open it again. Check that you are in your new project.
- Create new folders in your new project, called `scripts` and `output`

1.4.2 Reading data

Now we can read data into R.

1.4.2.1 Reading tab-delimited text

The file `pulse.txt` is a tab-delimited text file. We can read it into R with

```
# To get 'pulse.txt' directly from the server, use:
# pulse <- read.table( url( "/tree/master/data/pulse.txt" ), header = TRUE, sep = "\t"
```

```
pulse <- read.delim( "data/pulse.txt" )
```

We added `data/` because the file is in the data subfolder we just created.

Note that we assigned `<-` the result of `read.delim` to `pulse`. We have given our dataset the name `pulse`.

1.4.2.2 View

To check that you've read the data correctly

```
View(pulse)
```

Note the capital V.

Quick task(s):

Solve the task(s), and check your solution(s) here.

1.4.3 Scripts

So far we have been typing at the prompt. What we type at the prompt is executed and lost. Using scripts allows repeating things and make our results reproducible.

1.4.3.1 Making a script

We will open an R script File…New file…R script. An R script is just a text editor. Type some R code into the script

```
2^6
14+15
3-4
x <- log10(100)
y <- x^2
z <- c(3, 5, 2, 8)
z2 <- z+2
x
```

We can `run` (Ctrl+Enter) part of this code by sending it to the prompt. Check what happens if we run when

- The cursor is in a line
- We made a block of several lines
- We made a block of part of a line

If we use (Ctrl+Shift+Enter), the entire content of the R script file is run. We can do the same thing by using `source()` to run the whole script.

We can save a script. By default the file receives the `.R` extension.

1.4.3.2 Using a script

The script should contain the analysis you want to reproduce later.

Type at the prompt only to try things out.

To check that the script works, close RStudio and open again. Run the script.

More advanced way of working with scripts: R Markdown (later in the course)

1.4.3.3 Comments

R ignores everyting in a line after `#`. Use to put human readable text in your scripts (explanation).

Quick task(s):

Solve the task(s), and check your solution(s) here.

```
# Here I calculate x
x <- c(3, 6, 7, 8) *3
```

1.5 Data frames (basics)

So we are now able to read in data sets (tables). A data set in R is called a `data.frame`.

```
pulse <- read.delim( "data/pulse.txt" )
```

```
survey <- read.delim( "data/survey.txt" )
```

R can have many data sets in memory simultaneously. You will always have to specify which data set you are working in.

1.5.1 Exploring

1.5.1.1 Dimensions

Rows in a data.frame are typically subjects; columns are variables.

To find the size of a data.frame

```
pulse <- read.delim( "data/pulse.txt" )
ncol(pulse)
```

```
## [1] 12
```

```
nrow(pulse)
```

```
## [1] 110
```

```
dim(pulse)
```

```
## [1] 110  12
```

1.5.1.2 Showing head and tail

To get a quick impression of a data.frame:

- `head` prints the first 6 rows
- `tail` prints the last 6 rows

If you want more or less than 6, add the number you want as a second argument to the function:

```
head(pulse)
```

```
##            name height weight age gender smokes alcohol exercise ran pulse1
## 1993_A   Bonnie    173     57  18 female     no     yes moderate sat     86
## 1993_B  Melanie    179     58  19 female     no     yes moderate ran     82
## 1993_C Consuelo    167     62  18 female     no     yes     high ran     96
## 1993_D   Travis    195     84  18   male     no     yes     high sat     71
## 1993_E    Lauri    173     64  18 female     no     yes      low sat     90
## 1993_F   George    184     74  22   male     no     yes      low ran     78
##        pulse2 year
## 1993_A     88 1993
## 1993_B    150 1993
## 1993_C    176 1993
## 1993_D     73 1993
## 1993_E     88 1993
## 1993_F    141 1993
```

```
tail(pulse, 3)
```

```
##         name height weight age gender smokes alcohol exercise ran pulse1 pulse2
## 1998_P Chris    182     60  22   male     no     yes      low sat     86     84
## 1998_Q Lewis    170     65  18   male     no     yes     high sat     69     64
## 1998_R  Gene    185     85  19   male     no     yes moderate sat     75     68
##        year
## 1998_P 1998
## 1998_Q 1998
## 1998_R 1998
```

We've already seen `View`.

With `names` (alternatively `colnames`) you find the names of the variables (columns) in the data.frame:

```
names(pulse)
```

```
##  [1] "name"     "height"   "weight"   "age"      "gender"   "smokes"
##  [7] "alcohol"  "exercise" "ran"      "pulse1"   "pulse2"   "year"
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

1.5.2 Extracting

1.5.2.1 Columns

To extract a column of a data.frame use `$`. The result is a vector:

```
pulse$age
```

```
##   [1] 18 19 18 18 18 22 20 18 19 23 20 19 22 18 18 22 19 18 21 19 19 34 20 26 19
##  [26] 18 18 21 19 21 20 19 19 23 19 20 18 19 18 18 20 23 21 19 19 18 26 20 19 22
##  [51] 20 20 20 18 20 20 20 18 19 20 18 20 18 20 21 19 20 21 19 22 23 19 20 19 20
##  [76] 20 20 20 18 19 18 41 21 25 28 21 18 45 19 18 19 19 21 21 23 28 20 20 20 19
## [101] 24 19 20 20 23 19 19 22 18 19
```

Note the autocomplete in RStudio suggests the right column.

To add a column to a data.frame, use the assignment:

```
pulse$pulse.diff <- pulse$pulse2 - pulse$pulse1
head( pulse )
```

```
##            name height weight age gender smokes alcohol exercise ran pulse1
## 1993_A   Bonnie    173     57  18 female     no     yes moderate sat     86
## 1993_B  Melanie    179     58  19 female     no     yes moderate ran     82
## 1993_C Consuelo    167     62  18 female     no     yes     high ran     96
## 1993_D   Travis    195     84  18   male     no     yes     high sat     71
## 1993_E    Lauri    173     64  18 female     no     yes      low sat     90
## 1993_F   George    184     74  22   male     no     yes      low ran     78
##        pulse2 year pulse.diff
## 1993_A     88 1993          2
## 1993_B    150 1993         68
## 1993_C    176 1993         80
## 1993_D     73 1993          2
## 1993_E     88 1993         -2
## 1993_F    141 1993         63
```

1.5.2.2 Row names

A data.frame always has row names. Note that these names are not themselves a column of the data.frame!

```
rownames(pulse)
```

```
##   [1] "1993_A" "1993_B" "1993_C" "1993_D" "1993_E" "1993_F" "1993_G" "1993_H"
##   [9] "1993_I" "1993_J" "1993_K" "1993_L" "1993_M" "1993_N" "1993_O" "1993_P"
##  [17] "1993_Q" "1993_R" "1993_S" "1993_T" "1993_U" "1993_V" "1993_W" "1993_X"
##  [25] "1993_Y" "1993_Z" "1995_A" "1995_B" "1995_C" "1995_D" "1995_E" "1995_F"
##  [33] "1995_G" "1995_H" "1995_I" "1995_J" "1995_K" "1995_L" "1995_M" "1995_N"
##  [41] "1995_O" "1995_P" "1995_Q" "1995_R" "1995_S" "1995_T" "1995_U" "1995_V"
##  [49] "1996_A" "1996_B" "1996_C" "1996_D" "1996_E" "1996_F" "1996_G" "1996_H"
```

```
##  [57] "1996_I" "1996_J" "1996_K" "1996_L" "1996_M" "1996_N" "1996_O" "1996_P"
##  [65] "1996_Q" "1996_R" "1996_S" "1996_T" "1996_U" "1997_A" "1997_B" "1997_C"
##  [73] "1997_D" "1997_E" "1997_F" "1997_G" "1997_H" "1997_I" "1997_J" "1997_K"
##  [81] "1997_L" "1997_M" "1997_N" "1997_O" "1997_P" "1997_Q" "1997_R" "1997_S"
##  [89] "1997_T" "1997_U" "1997_V" "1997_W" "1998_A" "1998_B" "1998_C" "1998_D"
##  [97] "1998_E" "1998_F" "1998_G" "1998_H" "1998_I" "1998_J" "1998_K" "1998_L"
## [105] "1998_M" "1998_N" "1998_O" "1998_P" "1998_Q" "1998_R"
```

Name of each row must be unique.

1.5.2.3 Elements

An individual entry to a data.frame can be extracted using square brackets `[`, either using the names of row and column (note the quotes) or their indices. Row comes before the comma, column after.

```
pulse["1993_E", "height"]
```

```
## [1] 173
```

```
pulse[5, 2]
```

```
## [1] 173
```

You can also use ranges like with vectors

```
pulse[4:6, 1:5]
```

```
##          name height weight age gender
## 1993_D Travis    195     84  18   male
## 1993_E  Lauri    173     64  18 female
## 1993_F George    184     74  22   male
```

Much more about using square brackets later in the course.

1.5.2.4 (*) Removing a column

To remove a column from a data.frame, assign `NULL` to that column:

```
pulse$pulse.diff <- NULL
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

1.5.3 Example data

R contains many example data sets. To see which, see:

```
data()
```

Example data are immediately accessible in R. For example:

```
BOD
```

```
##   Time demand
## 1    1    8.3
## 2    2   10.3
## 3    3   19.0
## 4    4   16.0
## 5    5   15.6
## 6    7   19.8
```

Some description is always available:

```
?BOD
help("BOD")
```

We will make use of example data from the `MASS` package. Packages are bundles with additional functions and data. To make `MASS` available in your R session, say:

```
library(MASS)
```

More about packages later.

Quick task(s):

Solve the task(s), and check your solution(s) here.

1.6 Basic data types

We will use the `pulse` and `survey` data again for illustration and exercises:

```
pulse <- read.delim( "data/pulse.txt" )
```

```
survey <- read.delim( "data/survey.txt" )
```

1.6.1 Types

Columns in a data.frame can be of different types. Typically:

- numeric (we've seen already)
- character (for text)
- factor (for categorical variables)

1.6.1.1 str and class

To get a quick overview (structure) of the types of data in your data.frame:

```
str(pulse)
```

```
## 'data.frame':    110 obs. of  12 variables:
##  $ name    : chr  "Bonnie" "Melanie" "Consuelo" "Travis" ...
##  $ height  : int  173 179 167 195 173 184 162 169 164 168 ...
##  $ weight  : num  57 58 62 84 64 74 57 55 56 60 ...
##  $ age     : int  18 19 18 18 18 22 20 18 19 23 ...
##  $ gender  : chr  "female" "female" "female" "male" ...
##  $ smokes  : chr  "no" "no" "no" "no" ...
##  $ alcohol : chr  "yes" "yes" "yes" "yes" ...
##  $ exercise: chr  "moderate" "moderate" "high" "high" ...
##  $ ran     : chr  "sat" "ran" "ran" "sat" ...
##  $ pulse1  : int  86 82 96 71 90 78 68 71 68 88 ...
##  $ pulse2  : int  88 150 176 73 88 141 72 77 68 150 ...
##  $ year    : int  1993 1993 1993 1993 1993 1993 1993 1993 1993 1993 ...
```

To learn about the type of a specific column:

```
class(pulse$name)
```

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

1.6.2 Vector classes

1.6.2.1 Numeric data

Numeric data can be `integer` (whole numbers) or `numeric` (continuous data) but you can ignore that distinction if you are not a programmer.

We've seen how to make numeric data with `c` or `:`

1.6.2.1.1 Useful functions for numeric data Summarizing a single variable:

- `mean`
- `median`
- `min`
- `max`
- `range` (two values: min, max)
- `sd` (standard deviation)
- `var` (variance)
- `hist` (histogram)

A six-number summary: range, three quartiles and the mean:

```
summary(pulse$age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   18.00   19.00   20.00   20.56   21.00   45.00
```

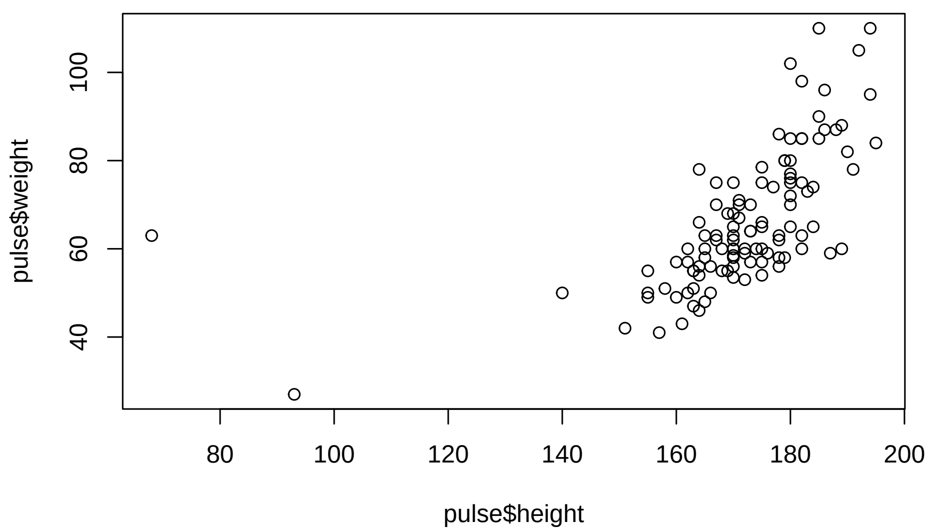
1.6.2.1.2 Relationships between two variables

- `cor` (correlation)
- `plot` (scatterplot)

```
cor(pulse$height, pulse$weight)
```

```
## [1] 0.5796849
```

```
plot(pulse$height, pulse$weight)
```



Quick task(s):

Solve the task(s), and check your solution(s) here.

1.6.3 Character data

Texts in R is called `character`. You recognize is by the quotes around the values.

1.6.3.1 Creating character data

Use either single or double quotes

```r
text <- c('alpha', 'beta', 'gamma')
TEXT <- c("ALPHA", "BETA", "GAMMA")
```

1.6.3.2 Names

Row and column names in a data.frame are always `character`:

```r
rownames(pulse)[1:10]
```

```
##  [1] "1993_A" "1993_B" "1993_C" "1993_D" "1993_E" "1993_F" "1993_G" "1993_H"
##  [9] "1993_I" "1993_J"
```

1.6.3.3 (*) Useful premade character vectors

- `LETTERS` (capitals)
- `letters` (lower case)
- `month.name` (months)

1.6.3.4 (*) Calculating with character vectors

Arithmetics of text

```r
toupper("me")
```

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

```r
paste(LETTERS, letters, sep='_')
```

```
##  [1] "A_a" "B_b" "C_c" "D_d" "E_e" "F_f" "G_g" "H_h" "I_i" "J_j" "K_k" "L_l"
## [13] "M_m" "N_n" "O_o" "P_p" "Q_q" "R_r" "S_s" "T_t" "U_u" "V_v" "W_w" "X_x"
## [25] "Y_y" "Z_z"
```

1.6.4 Factor - categorical data

A categorical variable in R is called `factor`:

- They are internally coded as numbers 1,2,3,...
- The numbers have value labels attached to them (called levels)

1.6.4.1 Making a factor

To make a factor variable, start with a character vector and use `factor`:

```r
fct <- factor(c('A', 'B', 'A', 'B', 'B'))
```

1.6.4.2 Factor basics

You recognize a factor by the `Levels:` line when printing:

```
fct
```

```
## [1] A B A B B
## Levels: A B
```

To get the numeric coding:

```
as.numeric(fct)
```

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

To get the value labels:

```
levels(fct)
```

```
## [1] "A" "B"
```

To get the number of levels (categories):

```
nlevels(fct)
```

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

1.6.4.3 Turning factor (back) into character

```
as.character(fct)
```

```
## [1] "A" "B" "A" "B" "B"
```

1.6.4.4 Table

Factors are best summarized with `table`:

```
table(pulse$exercise)
```

```
##
##     high      low moderate
##       14       37       59
```

`table` can also cross tabulate two (or even more) variables:

```
table(pulse$gender, pulse$exercise)
```

```
##
##          high low moderate
##   female    3  20       28
##   male     11  17       31
```

1.6.4.5 (*) Table of table

It is surprisingly useful to use table twice.

```
table(table(pulse$name))
```

```
##
##   1   2
## 102   4
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

1.7 Functions and help files

1.7.1 Function arguments

1.7.1.1 Functions may have multiple arguments

Example `round`:

```
round(pi,3)
```

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

```
round(pi,5)
```

```
## [1] 3.14159
```

Optional arguments may be left out:

```
round(pi)
```

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

1.7.1.2 Getting help for a function

Functions do calculations for you based on one or more arguments. To find out what arguments a function has and how they work, check the help file of the function. Two ways of getting help for a function

```
help(round)
?round
```

We see that `round` has two arguments and that the default of the second argument is 0.

1.7.1.3 Specifying arguments by name

Function arguments may be either given in the right order or specified explicitly by name.

```
round(pi, digits = 3)
```

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

The latter option is especially useful for functions with many arguments.

1.7.1.4 Default arguments

Look at the help file of `cor`. We see that the argument `method` has a vector of three options as a default. This means that the first mentioned value (`pearson`) is the default and the others (`kendall`, `spearman`) are alternative options.

1.7.1.5 (*) the `...` argument

The argument `...` means that a variable number of arguments may be given. See e.g. `sum` has a `...` argument since it sums all arguments together

```
sum(1,4,5:7,1)
```

```
## [1] 24
```

Arguments appearing after `...` must always be specified by name.

> *Quick task(s)*:
>
> Solve the task(s), and check your solution(s) here.

1.7.2 Other help file aspects

Help files typically explain the type of object that is returned by the function.

Help files also contain examples that can be run.

1.7.2.1 Help search

Finding help if you don't know the function name

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

Usually better: use the R help mailing list: http://www.r-project.org/mail.html. Someone has usually asked the same question you want to ask. The R mailing list is well indexed in google.

Or ask some local expert. At NKI, we recently created "NKI-R users group" on Teams.

Quick task(s):

Solve the task(s), and check your solution(s) here.

Chapter 2

Data types, part 1

2.1 Basic data types (cont.)

2.1.1 Missing values

Numeric, character, logical vectors, factors might contain elements marked as 'missing'.

`NA` is a constant which indicates a missing value.

`NA` values would appear in the course materials and tasks.

2.1.2 Numeric vectors

There are several ways to construct a vector of numbers.

2.1.2.1 Single number (numeric vector with a single element)

```
7
```

```
## [1] 7
```

2.1.2.2 Multiple numbers

```
c( 5, -2.5, NA, 7+3, 1/3 )
```

```
## [1]  5.0000000 -2.5000000         NA 10.0000000  0.3333333
```

2.1.2.3 Sequence of numbers (one by one)

```
3:17
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
```

2.1.2.4 Sequence of numbers

Sequence of numbers with a defined step:

```
seq( 5, 15, 2 )
```

```
## [1]  5  7  9 11 13 15
```

2.1.2.5 Combine (several) vectors of numbers

Multiple vectors can be combined together:

```
v <- 1:9
w <- seq( 10, 90, 10 )
c( 0, v, w, 100 )
```

```
##  [1]   0   1   2   3   4   5   6   7   8   9  10  20  30  40  50  60  70  80  90
## [20] 100
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

2.1.3 Character vectors

There are several ways to construct a vector of texts.

2.1.3.1 Simple single text (character vector with a single element)

With double quotes:

```
"Bioinformatics"
```

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

Or, with single quotes:

```
'Biostatistics'
```

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

2.1.3.2 Multiple texts

```
c( "Bioinformatics", "Biostatistics" )
```

```
## [1] "Bioinformatics" "Biostatistics"
```

2.1.4 Logical vectors

2.1.4.1 Elementary logical values

```
TRUE
```

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

```
FALSE
```

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

```
T
```

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

```
F
```

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

```
c( FALSE, F, TRUE, T )     # vector of logical values
```

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

(*) `NA` is a logical constant and it gets automatically converted to other types when necessary.

2.1.4.2 Logical operators

2.1.4.2.1 Negation Unary negation operator (denoted `!`):

```
!TRUE
```

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

```
!FALSE
```

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

2.1.4.2.2 AND Operator `&` Binary operator AND (denoted `&`) returns `TRUE` result when `all` its arguments are `TRUE`:

```
TRUE & TRUE
```

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

Otherwise returns `FALSE`:

```
FALSE & TRUE
```

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

```
FALSE & FALSE
```

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

2.1.4.2.3 OR operator `|`

Binary operator OR (denoted `|`) returns `TRUE` result when `any` (at least one) of its arguments is `TRUE`:

```
TRUE | TRUE
```

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

```
FALSE | TRUE
```

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

Otherwise returns `FALSE`:

```
FALSE | FALSE
```

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

2.1.4.3 Relational operators

```
1 == 2
```

2.1.4.3.1 Equality operator ==

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

```
"Bioinformatics" == "Biostatistics"
```

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

```
FALSE == F
```

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

```
1 != 2    # unequal
```

2.1.4.3.2 Inequality operators

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

```
"a" != "A"
```

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

```
FALSE != T
```

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

```
1 < 2     # less than
```

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

```
1 > 2     # greater than
```

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

```
2 <= 2    # less or equal
```

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

```
2 >= 2    # greater or equal
```

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

2.1.4.4 Comparison of two vectors

```
v <- c( 0, 1, 2, 3, 4 )
w <- c( 4, 3, 2, 1, 0 )
v == w
```

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

```
v != w
```

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

```
v < w
```

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

```
v <= w
```

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

```
v >= w
```

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

Quick task(s):

Solve the task(s), and check your solution(s) here.

2.1.5 Type conversions

Sometimes a conversion to a vector of certain type might be needed. The family of functions: `as.numeric`, `as.character`, `as.logical` take as an argument a vector of any type and return a vector of the type given in the function name. When the conversion of an element is not possible, `NA` value is used.

```
v <- 101:110
v
```

```
##  [1] 101 102 103 104 105 106 107 108 109 110
```

```
as.character( v )
```

```
##  [1] "101" "102" "103" "104" "105" "106" "107" "108" "109" "110"
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

2.2 Vectors

Vectors are data structures which are able to store multiple elements in a defined order.

There are several ways to access (single/multiple) elements from vectors which are discussed below.

2.2.1 Square brackets operator

2.2.1.1 By numbers

Vector elements are kept at successive, numbered (indexed) positions. R vectors keep the first element at index 1 and the last element at index which can be obtained with the function `length`.

The indices can be used to fetch elements from the corresponding positions with the square bracket operators `[ pos ]`.

Let's take a vector:

```
v <- 101:110
```

To get the first element of `v`:

```
v[1]
```

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

To get the last element of `v`:

```
v[ length( v ) ]
```

```
## [1] 110
```

To get an element at a given index:

```
v[ 5 ]
```

```
## [1] 105
```

Setting a single element:

```
v
```

```
##  [1] 101 102 103 104 105 106 107 108 109 110
```

```
v[ 2 ] <- 22
v
```

```
##  [1] 101  22 103 104 105 106 107 108 109 110
```

Getting multiple elements:

```
v[ c( 2,4,7 ) ]
```

```
## [1]  22 104 107
```

Setting multiple elements:

```
v[ c( 2,4,7 ) ] <- c( 22, 24, 27 )
v
```

```
##  [1] 101  22 103  24 105 106  27 108 109 110
```

2.2.1.2 By names

Elements of the vectors can also be given names, i.e. `named vector`. Let's take a vector:

```
v <- 101:110
v
```

```
##  [1] 101 102 103 104 105 106 107 108 109 110
```

Setting names of vector elements:

```
names( v ) <- LETTERS[ 1:length( v ) ]
v
```

```
##   A   B   C   D   E   F   G   H   I   J
## 101 102 103 104 105 106 107 108 109 110
```

Getting names of vector elements:

```
names( v )
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

Accessing elements (single or multiple) by names:

```
v[ "C" ]
```

```
##   C
## 103
```

```
v[ c( "F", "H" ) ]
```

```
##   F   H
## 106 108
```

2.2.1.3 By condition (logical indices)

We can use a logical vector to select elements of a vector. Consider the vector

```
b <- c(5, 7)
```

and say that we want to select the second element, but not the first. For this we know we can use

```
b[2]
```

```
## [1] 7
```

and it turns out we can also use a logical vector with `TRUE` for the element to be selected, and `FALSE` for the one not to be selected. That is `b[ c(FALSE, TRUE) ]` gives the same selection as above:

```
b[ c(FALSE, TRUE) ]
```

```
## [1] 7
```

This example is trivial. Let us take a longer vector:

```
v <- 101:110
v
```

```
##  [1] 101 102 103 104 105 106 107 108 109 110
```

Now let us construct a logical vector `idx` with the same number of elements as `v`:

```
idx <- c( F, F, F, T, F, T, F, F, T, NA )
```

`v` can now be "filtered through" `idx`, where positions in `idx` are selected or not according to:

- `TRUE` in `idx`, then the element appears in the result;
- `FALSE` in `idx`, then the element is skipped;
- `NA` in `idx`, then the element is substituted with `NA` in the result.

This means that `NA` entries in `idx` will include `NA`s entries in the result, even if the original vector `v` did not involve `NA`s.

In the example above, the result is:

```
v[ idx ]
```

```
## [1] 104 106 109  NA
```

A logical vector like `idx` is typically a result of equality/comparison, or is generated by specialized functions. For example, if we wish to select entries in `v` that have values at least equal to 102 and smaller than 106, we could use:

```
v
```

```
##  [1] 101 102 103 104 105 106 107 108 109 110
```

```
idx <- v >= 102
idx
```

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

```
v[ idx ]
```

```
## [1] 102 103 104 105 106 107 108 109 110
```

We can also combine logical statements into a single one. Say, if we wish to select entries in `v` that have values at least equal to 102 and smaller than 106, we could use:

```
v
```

```
##  [1] 101 102 103 104 105 106 107 108 109 110
```

```
idx <- (v >= 102) & (v < 106)
idx
```

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

```
v[ idx ]
```

```
## [1] 102 103 104 105
```

Indices given by a logical vector are referred to as *logical indices*.

Quick task(s):

Solve the task(s), and check your solution(s) here.

2.2.2 Other useful functions

Other functions which are in general useful are:

- `sort()`: sorts the elements of a vector:

```
v
```

```
##  [1] 101 102 103 104 105 106 107 108 109 110
```

```
sort(v)
```

```
##  [1] 101 102 103 104 105 106 107 108 109 110
```

- `unique()`: returns a vector containing the unique entries of a vector:

```
u <- c(5, 5, 4, 4, 3, 1)
unique(u)
```

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

```
w <- c("Monday", "Tuesday", "Thursday", "Friday", "Monday", "Wednesday", "Thursday")
w
```

```
## [1] "Monday"    "Tuesday"   "Thursday"  "Friday"    "Monday"    "Wednesday"
## [7] "Thursday"
```

```
unique(w)
```

```
## [1] "Monday"    "Tuesday"   "Thursday"  "Friday"    "Wednesday"
```

- `duplicated()`: returns a logical vector indicating which entries are duplicated or not:

```
duplicated(u)
```

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

```
duplicated(w)
```

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

So, in order to select all duplicated entries, the following could be used:

```
u[ duplicated(u) ]
```

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

If instead you wish to select all unique entries, you can use:

```
u[ !duplicated(u) ]
```

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

You can check that this is the same result that you get using `unique`:

```
unique(u)
```

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

Quick task(s):

Solve the task(s), and check your solution(s) here.

2.3 Data frames

2.3.1 What is a data frame

A data frame is a collection of variables represented as vectors of the same length.

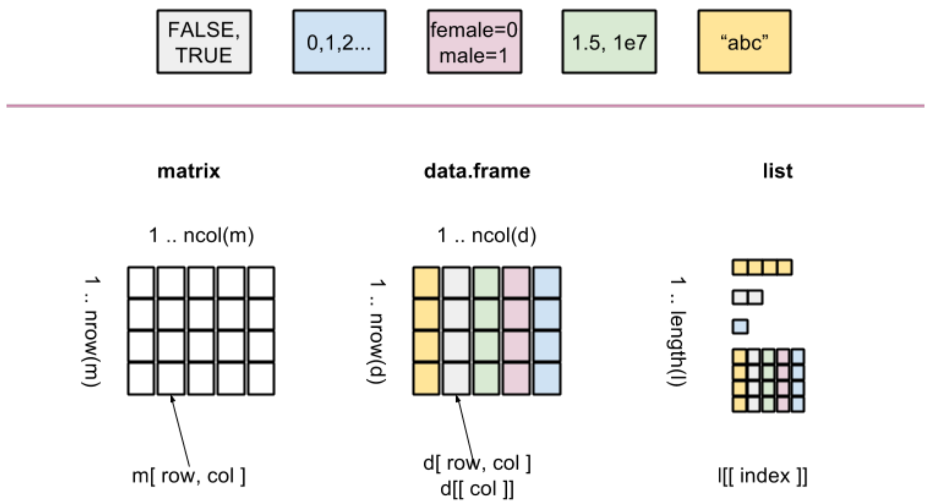


Figure 2.1: Vectors

Therefore, all the vectors (columns) should have unique names. They can be of different types: one column can be character, another a factor, and other columns can be numeric. Of course, entries within a single vector are all of the same type.

Rows represent separate records.

Rows may have names as well (although for new code, better create a separate column with names).

2.3.2 Creating

2.3.2.1 From manually provided vectors

An example data frame built of `character`, `numeric` and `logical` vectors:

```
ids <- c( "A", "B", "D", "E" );
ns <- c( "Amy", "Bob", "Dan", "Eve" )
as <- c( 40, NA, 6, 16 )
ss <- c( TRUE, NA, FALSE, FALSE )

d <- data.frame(
  row.names = ids,
  name = ns,
  age = as,
```

```
  smoker = ss
)
d
```

```
##   name age smoker
## A  Amy  40   TRUE
## B  Bob  NA     NA
## D  Dan   6  FALSE
## E  Eve  16  FALSE
```

The class of `d` is:

```
class( d )
```

```
## [1] "data.frame"
```

Function `str` gives a compact display of an object structure/content:

```
str( d )
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ name  : chr  "Amy" "Bob" "Dan" "Eve"
##  $ age   : num  40 NA 6 16
##  $ smoker: logi  TRUE NA FALSE FALSE
```

2.3.2.2 Import from files

Datasets my come in various formats, e.g. `.tsv`, `.csv`, `.xls(x)`, `.sav`, `.txt` etc. R provides functions to import these formats as data.frame.

Note: always after a file is read the columns must be checked whether their class and values are as expected. Additional conversions might be necessary (e.g. declaring factor levels, order of factor levels).

2.3.2.2.1 TSV file For files stored in tab-separated files (`tsv`) format use:

```
d <- read.table( "data/pulse.tsv", header = TRUE, sep = "\t" )

# To get 'pulse.tsv' directly from the server, use:
# d <- read.table( url( "/tree/master/data/pulse.tsv" ), header = TRUE, sep = "\t" )
str( d )
```

```
## 'data.frame':    110 obs. of  12 variables:
##  $ name    : chr  "Bonnie" "Melanie" "Consuelo" "Travis" ...
##  $ height  : int  173 179 167 195 173 184 162 169 164 168 ...
##  $ weight  : num  57 58 62 84 64 74 57 55 56 60 ...
##  $ age     : int  18 19 18 18 18 22 20 18 19 23 ...
##  $ gender  : chr  "female" "female" "female" "male" ...
##  $ smokes  : chr  "no" "no" "no" "no" ...
```

```
## $ alcohol : chr "yes" "yes" "yes" "yes" ...
## $ exercise: chr "moderate" "moderate" "high" "high" ...
## $ ran : chr "sat" "ran" "ran" "sat" ...
## $ pulse1 : int 86 82 96 71 90 78 68 71 68 88 ...
## $ pulse2 : int 88 150 176 73 88 141 72 77 68 150 ...
## $ year : int 1993 1993 1993 1993 1993 1993 1993 1993 1993 1993 ...
```

2.3.2.2.2 CSV file Data frames (tables) stored in files in comma-separated values (csv) format can be read with:

```
d <- read.csv( "data/pulse.csv" );
```

```
# To get 'pulse.csv' directly from the server, use:
# d <- read.csv( url( "/tree/master/data/pulse.csv" ) )
```

```
str( d )
```

```
## 'data.frame': 110 obs. of 13 variables:
## $ id : chr "1993_A" "1993_B" "1993_C" "1993_D" ...
## $ name : chr "Bonnie" "Melanie" "Consuelo" "Travis" ...
## $ height : int 173 179 167 195 173 184 162 169 164 168 ...
## $ weight : num 57 58 62 84 64 74 57 55 56 60 ...
## $ age : int 18 19 18 18 18 22 20 18 19 23 ...
## $ gender : chr "female" "female" "female" "male" ...
## $ smokes : chr "no" "no" "no" "no" ...
## $ alcohol : chr "yes" "yes" "yes" "yes" ...
## $ exercise: chr "moderate" "moderate" "high" "high" ...
## $ ran : chr "sat" "ran" "ran" "sat" ...
## $ pulse1 : int 86 82 96 71 90 78 68 71 68 88 ...
## $ pulse2 : int 88 150 176 73 88 141 72 77 68 150 ...
## $ year : int 1993 1993 1993 1993 1993 1993 1993 1993 1993 1993 ...
```

2.3.2.2.3 (*) Microsoft Excel file Reading Microsoft Excel files requires an additional library/package, which needs to be installed first. There exists several packages providing reading of Excel files.

```
# install.packages( "readxl" )
library( readxl );
d <- read_excel( "data/pulse.xlsx", sheet = 1 );
str( d );
```

2.3.2.2.3.1 (*) With package readxl

```
## tibble [110 x 13] (S3: tbl_df/tbl/data.frame)
## $ id : chr [1:110] "1993_A" "1993_B" "1993_C" "1993_D" ...
## $ name : chr [1:110] "Bonnie" "Melanie" "Consuelo" "Travis" ...
## $ height : num [1:110] 173 179 167 195 173 184 162 169 164 168 ...
```

```
##  $ weight  : num [1:110] 57 58 62 84 64 74 57 55 56 60 ...
##  $ age     : num [1:110] 18 19 18 18 18 22 20 18 19 23 ...
##  $ gender  : chr [1:110] "female" "female" "female" "male" ...
##  $ smokes  : chr [1:110] "no" "no" "no" "no" ...
##  $ alcohol : chr [1:110] "yes" "yes" "yes" "yes" ...
##  $ exercise: chr [1:110] "moderate" "moderate" "high" "high" ...
##  $ ran     : chr [1:110] "sat" "ran" "ran" "sat" ...
##  $ pulse1  : chr [1:110] "86" "82" "96" "71" ...
##  $ pulse2  : chr [1:110] "88" "150" "176" "73" ...
##  $ year    : num [1:110] 1993 1993 1993 1993 1993 ...
```

2.3.2.2.3.2 (*) With package gdata Please note that `gdata` package requires additional PERL packages to be installed; `readxl` seems to be easier to use.

```
# install.packages( "gdata" )
library( gdata )
d <- read.xls( "data/pulse.xlsx", sheet = 1 )
str( d )
```

2.3.2.3 (*) SPSS files

Reading SPSS files requires an additional library/package `foreign` (normally installed with R distribution).

```
library( foreign )
d <- read.spss( "data/pulse.sav", to.data.frame = TRUE )
str( d )
```

```
## 'data.frame':    110 obs. of  13 variables:
##  $ id      : chr  "1993_A" "1993_B" "1993_C" "1993_D" ...
##  $ name    : chr  "Bonnie     " "Melanie    " "Consuelo   " "Travis     " ...
##  $ height  : num  173 179 167 195 173 184 162 169 164 168 ...
##  $ weight  : num  57 58 62 84 64 74 57 55 56 60 ...
##  $ age     : num  18 19 18 18 18 22 20 18 19 23 ...
##  $ gender  : Factor w/ 2 levels "female","male": 1 1 1 2 1 2 1 1 1 2 ...
##  $ smokes  : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ alcohol : Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 2 2 2 ...
##  $ exercise: Factor w/ 3 levels "low","moderate",..: 2 2 3 3 1 1 2 2 3 2 ...
##  $ ran     : Factor w/ 2 levels "sat","ran": 1 2 2 1 1 2 1 1 1 2 ...
##  $ pulse1  : num  86 82 96 71 90 78 68 71 68 88 ...
##  $ pulse2  : num  88 150 176 73 88 141 72 77 68 150 ...
##  $ year    : Factor w/ 5 levels "1993","1995",..: 1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, "codepage")= int 65001
```

2.3.3 Properties

Let's discuss the data frame `pulse`:

```
pulse <- read.table( "data/pulse.txt", header = TRUE, sep = "\t" )

# To get 'pulse.txt' directly from the server, use:
# pulse <- read.table( url( "/tree/master/data/pulse.txt" ), header = TRUE, sep = "\t" )
```

To shorten output we will use the first 20 rows of the `pulse` data frame:

```
pulse <- head( pulse, 20 )
```

2.3.3.1 Dimensions

There are several functions to get dimensions of a data frame:

- `ncol( pulse )` provides the number of columns:

```
ncol( pulse )

## [1] 12
```

- `nrow( pulse )` provides the number of rows:

```
nrow( pulse )

## [1] 20
```

- `dim( pulse )` returns a vector with two elements: number of rows and number of columns

```
dim( pulse )

## [1] 20 12
```

2.3.3.2 Columns/rows names

`colnames()` is used to get the names of the columns. In data frames, the same result is returned by `names()`:

```
colnames( pulse )

##  [1] "name"     "height"   "weight"   "age"      "gender"   "smokes"
##  [7] "alcohol"  "exercise" "ran"      "pulse1"   "pulse2"   "year"
```

```
names( pulse )

##  [1] "name"     "height"   "weight"   "age"      "gender"   "smokes"
##  [7] "alcohol"  "exercise" "ran"      "pulse1"   "pulse2"   "year"
```

To get names of the rows use:

```
rownames( pulse )
```

```
##  [1] "1993_A" "1993_B" "1993_C" "1993_D" "1993_E" "1993_F" "1993_G" "1993_H"
##  [9] "1993_I" "1993_J" "1993_K" "1993_L" "1993_M" "1993_N" "1993_O" "1993_P"
## [17] "1993_Q" "1993_R" "1993_S" "1993_T"
```

2.3.4 Content

A column of a data frame might be accessed through the `$` operator.

Additionally, the content of a data frame can be accessed with the square bracket `[ ]` (square brackets) operator used in two different ways:

- with one argument `[ col(s) ]` referring to a column
- with two arguments `[ row(s), col(s) ]`

We can illustrate this by creating a vector with the `height` variable of the `pulse` data, using both syntaxes:

```
height1 <- pulse[ "height" ]
height2 <- pulse[, "height" ]
```

Now check the class of the objects created:

```
class(height1)
```

```
## [1] "data.frame"
```

```
class(height2)
```

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

So, by extracting one column of `pulse`using the syntax `[, columnName]`, a vector is created

2.3.4.1 Dollar operator

The `$` method returns a *single* column as a *vector*:

```
pulse$weight
```

```
##  [1]  57  58  62  84  64  74  57  55  56  60  75  58  68  59  72 110  56  70  56
## [20]  50
```

When the column name is valid the returned value is a *vector*:

```
class( pulse$weight )
```

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

When the name is invalid, `NULL` is returned:

```
pulse$wrong
```

```
## NULL
```

Be careful, the $ notation searches for a column name starting with a provided prefix:

```
pulse$we
```

```
##  [1]  57  58  62  84  64  74  57  55  56  60  75  58  68  59  72 110  56  70  56
## [20]  50
```

2.3.4.2 Square brackets operator, single argument

Square brackets notation with a single argument return requested columns as a *data frame* (compare to the dollar operator section):

```
pulse[ 'weight' ]
```

```
##        weight
## 1993_A     57
## 1993_B     58
## 1993_C     62
## 1993_D     84
## 1993_E     64
## 1993_F     74
## 1993_G     57
## 1993_H     55
## 1993_I     56
## 1993_J     60
## 1993_K     75
## 1993_L     58
## 1993_M     68
## 1993_N     59
## 1993_O     72
## 1993_P    110
## 1993_Q     56
## 1993_R     70
## 1993_S     56
## 1993_T     50
```

```
class( pulse[ 'weight' ] )
```

```
## [1] "data.frame"
```

```
pulse[ 3 ]
```

```
##        weight
## 1993_A     57
```

```
## 1993_B     58
## 1993_C     62
## 1993_D     84
## 1993_E     64
## 1993_F     74
## 1993_G     57
## 1993_H     55
## 1993_I     56
## 1993_J     60
## 1993_K     75
## 1993_L     58
## 1993_M     68
## 1993_N     59
## 1993_O     72
## 1993_P    110
## 1993_Q     56
## 1993_R     70
## 1993_S     56
## 1993_T     50
```

```
class( pulse[ 3 ] )
```

```
## [1] "data.frame"
```

Since the returned object is a `data.frame`, multiple columns might be returned (for example in a different order):

```
pulse[ c( 'height', 'weight' ) ]
```

```
##        height weight
## 1993_A    173     57
## 1993_B    179     58
## 1993_C    167     62
## 1993_D    195     84
## 1993_E    173     64
## 1993_F    184     74
## 1993_G    162     57
## 1993_H    169     55
## 1993_I    164     56
## 1993_J    168     60
## 1993_K    170     75
## 1993_L    178     58
## 1993_M    170     68
## 1993_N    187     59
## 1993_O    180     72
## 1993_P    185    110
## 1993_Q    170     56
```

```
## 1993_R    180     70
## 1993_S    166     56
## 1993_T    155     50
```

When a name is invalid, an error is produced:

```
pulse[ 'wrong' ]
```

```
## Error in `[.data.frame`(pulse, "wrong"): undefined columns selected
```

2.3.4.3 Square brackets operator, two arguments

Single brackets notation with two arguments `[ row(s), col(s) ]` might be used to get (multiple) row(s) and (multiple) column(s):

```
pulse[ c( 1, 3 ), c( 'height', 'weight' ) ]
```

```
##        height weight
## 1993_A    173     57
## 1993_C    167     62
```

```
pulse[ c( "1993_C", "1993_A", "wrong" ), c( 'height', 'weight' ) ]
```

```
##        height weight
## 1993_C    167     62
## 1993_A    173     57
## NA         NA     NA
```

Warning: notice the difference of the class of the output when only a single column is requested:

```
pulse[ , c( 'height', 'weight' ) ]
```

```
##        height weight
## 1993_A    173     57
## 1993_B    179     58
## 1993_C    167     62
## 1993_D    195     84
## 1993_E    173     64
## 1993_F    184     74
## 1993_G    162     57
## 1993_H    169     55
## 1993_I    164     56
## 1993_J    168     60
## 1993_K    170     75
## 1993_L    178     58
## 1993_M    170     68
## 1993_N    187     59
## 1993_O    180     72
## 1993_P    185    110
```

```
## 1993_Q    170     56
## 1993_R    180     70
## 1993_S    166     56
## 1993_T    155     50
```

```
class( pulse[ , c( 'height', 'weight' ) ] )
```

```
## [1] "data.frame"
```

```
pulse[ , c( 'weight' ) ]
```

```
##  [1]  57  58  62  84  64  74  57  55  56  60  75  58  68  59  72 110  56  70  56
## [20]  50
```

```
class( pulse[ , c( 'weight' ) ] )
```

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

Empty index field means “all” rows or columns:

```
pulse[ , ]
```

```
##              name height weight age gender smokes alcohol exercise ran pulse1
## 1993_A     Bonnie    173     57  18 female     no     yes moderate sat     86
## 1993_B    Melanie    179     58  19 female     no     yes moderate ran     82
## 1993_C   Consuelo    167     62  18 female     no     yes     high ran     96
## 1993_D     Travis    195     84  18   male     no     yes     high sat     71
## 1993_E      Lauri    173     64  18 female     no     yes      low sat     90
## 1993_F     George    184     74  22   male     no     yes      low ran     78
## 1993_G     Cherry    162     57  20 female     no     yes moderate sat     68
## 1993_H  Francesca    169     55  18 female     no     yes moderate sat     71
## 1993_I      Sonja    164     56  19 female     no     yes     high sat     68
## 1993_J       Troy    168     60  23   male     no     yes moderate ran     88
## 1993_K     Roland    170     75  20   male     no     yes     high ran     76
## 1993_L  Frederick    178     58  19   male     no      no      low sat     74
## 1993_M     Justin    170     68  22   male    yes     yes moderate sat     70
## 1993_N     Ernest    187     59  18   male     no     yes     high sat     78
## 1993_O   Salvador    180     72  18   male     no     yes moderate sat     69
## 1993_P     Mathew    185    110  22   male     no     yes      low sat     77
## 1993_Q     Leslie    170     56  19   male     no      no      low sat     64
## 1993_R    Raymond    180     70  18   male     no     yes moderate ran     80
## 1993_S     Nicole    166     56  21 female    yes      no moderate sat     83
## 1993_T      Maura    155     50  19 female     no      no moderate sat     78
##        pulse2 year
## 1993_A     88 1993
## 1993_B    150 1993
## 1993_C    176 1993
## 1993_D     73 1993
## 1993_E     88 1993
```

```
## 1993_F    141 1993
## 1993_G     72 1993
## 1993_H     77 1993
## 1993_I     68 1993
## 1993_J    150 1993
## 1993_K     88 1993
## 1993_L     76 1993
## 1993_M     71 1993
## 1993_N     82 1993
## 1993_O     67 1993
## 1993_P     73 1993
## 1993_Q     63 1993
## 1993_R    146 1993
## 1993_S     79 1993
## 1993_T     79 1993
```

```
pulse[ , c( 'height', 'weight' ) ]
```

```
##        height weight
## 1993_A    173     57
## 1993_B    179     58
## 1993_C    167     62
## 1993_D    195     84
## 1993_E    173     64
## 1993_F    184     74
## 1993_G    162     57
## 1993_H    169     55
## 1993_I    164     56
## 1993_J    168     60
## 1993_K    170     75
## 1993_L    178     58
## 1993_M    170     68
## 1993_N    187     59
## 1993_O    180     72
## 1993_P    185    110
## 1993_Q    170     56
## 1993_R    180     70
## 1993_S    166     56
## 1993_T    155     50
```

```
pulse[ c( "1993_C", "1993_A" ), ]
```

```
##            name height weight age gender smokes alcohol exercise ran pulse1
## 1993_C Consuelo    167     62  18 female     no     yes     high ran     96
## 1993_A   Bonnie    173     57  18 female     no     yes moderate sat     86
##        pulse2 year
## 1993_C    176 1993
```

```
## 1993_A     88 1993
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

2.4 Matrices

2.4.1 What is a matrix

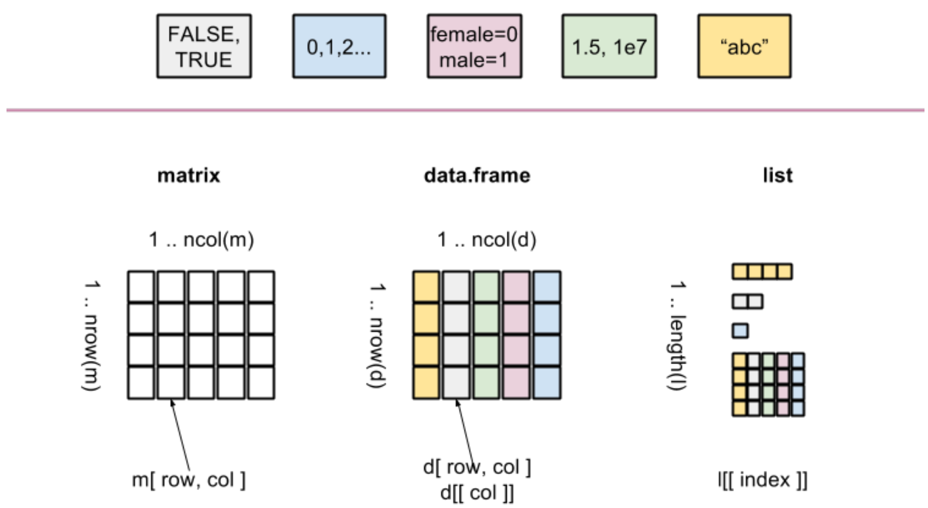


Figure 2.2: Vectors

Matrix is a two-dimensional array of elements of the same type.

Rows and columns are addressed by numerical indices.

Rows and columns might also get names. The names might be used for indexing.

2.4.2 Creation

A matrix can be constructed from a vector. Depending on the arguments, elements are put to the matrix in a different order:

```
m <- matrix( 1:6, nrow = 2 );
m
```

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

```
m <- matrix( 1:6, nrow = 2, byrow = TRUE );
m
```

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

```
m <- matrix( 1:6, ncol = 2 );
m
```

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

The class of `m` is:

```
class( m )
```

```
## [1] "matrix" "array"
```

Function `str` gives a compact display of an object structure/content:

```
str( m )
```

```
##  int [1:3, 1:2] 1 2 3 4 5 6
```

2.4.3 Dimensions

There are several functions to get dimensions of a matrix:

- `ncol( m )` provides the number of columns:

```
ncol( m )
```

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

- `nrow( m )` provides the number of rows:

```
nrow( m )
```

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

- `dim( m )` returns a vector with two elements: number of rows and number of columns

```
dim( m )
```

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

2.4.4 Setting/getting names of the columns and rows

`colnames( m )` and `rownames( m )` are used to set and get the names of matrix columns and rows:

```
m
```

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

```
colnames( m ) <- c( "A", "B" )
m
```

```
##      A B
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
```

```
rownames( m ) <- c( "X", "Y", "Z" )
m
```

```
##   A B
## X 1 4
## Y 2 5
## Z 3 6
```

To get names use:

```
rownames( m )
```

```
## [1] "X" "Y" "Z"
```

```
colnames( m )
```

```
## [1] "A" "B"
```

2.4.5 Getting matrix elements

Single brackets notation with two arguments `[ row(s), col(s) ]` might be used to get specified row(s) and column(s). By default single rows/cols will get reduced to a vector.

```
m[ 3, 1 ]
```

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

```
m[ c( 2, 3 ), 1 ]
```

```
## Y Z
## 2 3
```

```
m[ c( 2, 3 ), c( "B", "A" ) ]
```

```
##   B A
## Y 5 2
## Z 6 3
```

```
m[ c( F, T, T ), c( "B", "A" ) ]
```

```
##   B A
## Y 5 2
## Z 6 3
```

Notice the difference in the output class when only a single element is requested:

```
class( m[ 3, 1 ] )
```

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

```
class( m[ c( 2, 3 ), c( "B", "A" ) ] )
```

```
## [1] "matrix" "array"
```

Dropping of `matrix` dimensionality might be prevented:

```
m[ 3, 1 ]
```

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

```
m[ 3, 1, drop = FALSE ]
```

```
##   A
## Z 3
```

```
class( m[ 3, 1, drop = FALSE ] )
```

```
## [1] "matrix" "array"
```

Empty index field means “all” rows or columns:

```
m[ , c( "B", "A" ) ]
```

```
##   B A
## X 4 1
## Y 5 2
## Z 6 3
```

```
m[ c( "Z", "X", "Y" ), ]
```

```
##   A B
```

```
## Z 3 6
## X 1 4
## Y 2 5
```

```
m[ , ]
```

```
##   A B
## X 1 4
## Y 2 5
## Z 3 6
```

2.4.6 Useful matrix functions

Short summary of matrix operations: http://www.statmethods.net/advstats/matrix.html

2.4.6.1 Row/columns means/sums

```
m
```

```
##   A B
## X 1 4
## Y 2 5
## Z 3 6
```

```
rowMeans( m )
```

```
##   X   Y   Z
## 2.5 3.5 4.5
```

```
rowSums( m )
```

```
## X Y Z
## 5 7 9
```

```
colMeans( m )
```

```
## A B
## 2 5
```

```
colSums( m )
```

```
##  A  B
##  6 15
```

2.4.6.2 (*) Transposition

```
m <- matrix( 1:6, nrow = 3 );
colnames( m ) <- c( "A", "B" )
```

```
rownames( m ) <- c( "X", "Y", "Z" )
m
```

```
##   A B
## X 1 4
## Y 2 5
## Z 3 6
```

```
t(m)
```

```
##   X Y Z
## A 1 2 3
## B 4 5 6
```

2.4.6.3 (*) Matrix multiplication

```
m
```

```
##   A B
## X 1 4
## Y 2 5
## Z 3 6
```

```
t( m )
```

```
##   X Y Z
## A 1 2 3
## B 4 5 6
```

```
m %*% t( m )
```

```
##    X  Y  Z
## X 17 22 27
## Y 22 29 36
## Z 27 36 45
```

2.4.6.4 (*) Element-wise multiplication

```
m
```

```
##   A B
## X 1 4
## Y 2 5
## Z 3 6
```

```
m+1
```

```
##   A B
## X 2 5
## Y 3 6
```

```
## Z 4 7
```

```
m * (m+1)
```

```
##    A  B
## X  2 20
## Y  6 30
## Z 12 42
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

Chapter 3

Data types, part 2

3.1 R scripts and reports (Rmarkdown)

3.1.1 Markdown : R code + Markdown text formating language

With RMarkdown you can combine text, scripts and results. This makes it easy to organize projects.

- RMarkdown (cheat sheet) is a simple language for creating professional reports

An RMarkdown document includes your R code output. Use to:

- share your data analysis with colleagues
- document your data analysis for future generations

The RMarkdown report can be exported in various formats: `html`, `pdf` and `word`

3.1.1.1 Short session

- Opening a new RMarkdown file
  - From RStudio File -> New File -> R Markdown
  - Choose title and output format, choose `html` (default) here and press OK.
- Knitting (compiling R Markdown)
  - New RMarkdown file contains example text and code generated for you.
  - To knit, press the `knit` button in Rstudio, or use the shortcut `CTRL+Shift+k`. It first asks you to save your file. Your file will by default get extension `.Rmd`

Quick exercises : Review the RMarkdown with the html produced. What did RMarkdown do?

3.1.2 Features of R Markdown

3.1.2.1 Markdown examples

Use

- two spaces at the end of a line for new paragraph (newline)
- hash (`#`) for section titles.
- dash (`-`) for bullet lists.
- *italic*, **bold** and ~~strike through~~ for *italic*, **bold**, ~~strike through~~ respectively
- > to highlight

to highlight

3.1.2.2 Including R code in markdown

R code is included in a `chunk`:

```
x <- 5
x + 10
```

```
## [1] 15
```

```
x <- 5
x + 10
```

```
## [1] 15
```

3.1.2.3 Chunk options

There are many chunck options available, most common being `echo`, `eval`. For example the option results='hide' prevents the result of the `x + 10` being included in the compiled report:

```
x <- 5
x + 10
```

```
x <- 5
x + 10
```

If you have a code chunk that generates a lot of output and you cannot switch that off while running it, this may be useful.

3.1.2.4 R Studio and RMarkdown

- RMarkdown is well integrated with RStudio.
- By pressing `knit` a new R session is started and terminated after generating the report.

- This stand-alone nature helps to make sure that your analysis is ***reproducible***.
- The working directory of the markdown R session is the location of the markdown document.
- The knitting R session is independent from the R session in the RStudio console, this means that:
  - All data needed for the generation of the report must be loaded/created by the scripts in the R Markdown document.
  - and conversely no data from R session in the R Studio console is accessible by the R session started for knitting.
- It is possible to run a block of R code from R Markdown document inside the R Studio console.
- etc.

3.1.2.5 Try this!

From now on, work only with RMarkdown for this course.

3.2 Lists

3.2.1 What is a list

A list can be thought of as zero or more (named) cells containing data of *any type*, as well as of various lengths.

Elements may be named.

Elements are accessed through indexing operations.

3.2.2 Creation

```
study <- list(              # here = or <-
  name = "Bob",             # but here only =
  age = 44,                 # never <-
  children = c( "Amy", "Dan", "Eve" )
)
study
```

```
## $name
## [1] "Bob"
##
## $age
## [1] 44
##
## $children
## [1] "Amy" "Dan" "Eve"
```

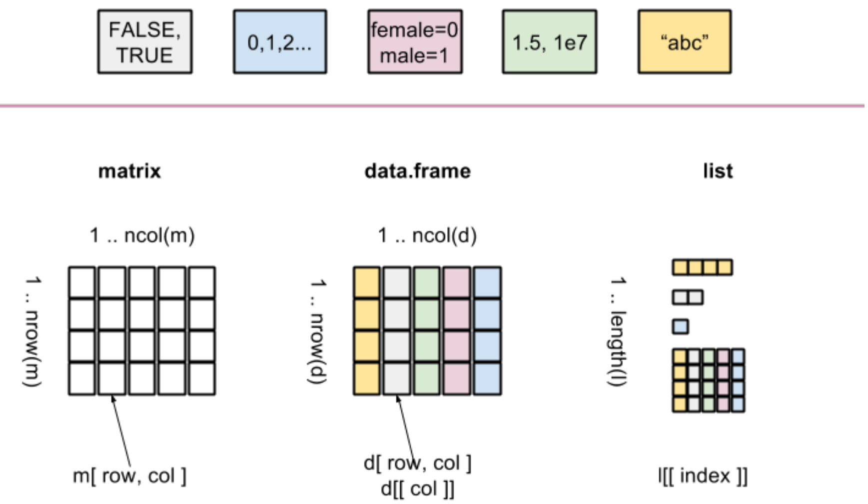


Figure 3.1: Lists

The class of `study`:

```
class( study )
```

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

Function `str` gives a compact display of an object structure/content:

```
str( study )
```

```
## List of 3
##  $ name    : chr "Bob"
##  $ age     : num 44
##  $ children: chr [1:3] "Amy" "Dan" "Eve"
```

3.2.3 Length

Length of the list (number of elements):

```
length( study )
```

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

3.2.4 Getting names of the elements

```
names( study )
```

```
## [1] "name"     "age"      "children"
```

3.2.5 Getting a single element

Single elements can be accessed by their names in the list:

```
study$age
```

```
## [1] 44
```

```
study[[ "age" ]]
```

```
## [1] 44
```

Note the double [[.

The type of the returned element depends on the element:

```
class( study$age )
```

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

```
class( study$children )
```

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

When referring to nonexisting elements, the result is `NULL`

```
study$parents
```

```
## NULL
```

It is also possible to access elements by numerical index:

```
study[[ 2 ]]
```

```
## [1] 44
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

3.2.6 (*) Getting multiple elements as a list

To get (possibly) multiple elements use single brackets notation:

```
study[ c( "age", "children" ) ]
```

```
## $age
## [1] 44
##
## $children
## [1] "Amy" "Dan" "Eve"
```

Note the difference:

```
study[[ "age" ]]
```

```
## [1] 44
```

```
study[ "age" ]
```

```
## $age
## [1] 44
```

```
class( study[[ "age" ]] )
```

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

```
class( study[ "age" ] )
```

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

Accessing nonexisting elements with single brackets:

```
study[ "parents" ]
```

```
## $<NA>
## NULL
```

```
class( study[ "parents" ] )
```

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

```
names( study[ "parents" ] )
```

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

```
is.null( study[ "parents" ] )
```

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

Numerical or logical indices may be also used:

```
study[ c( 3, 1, 1 ) ]
```

```
## $children
## [1] "Amy" "Dan" "Eve"
##
```

```
## $name
## [1] "Bob"
##
## $name
## [1] "Bob"
```

```
study[ c( T, F, T ) ]
```

```
## $name
## [1] "Bob"
##
## $children
## [1] "Amy" "Dan" "Eve"
```

3.2.7 (*) Removing an element

A list element is removed by setting to NULL:

```
str( study )
```

```
## List of 3
##  $ name    : chr "Bob"
##  $ age     : num 44
##  $ children: chr [1:3] "Amy" "Dan" "Eve"
```

```
study[[ "children" ]] <- NULL;    # NULL means "nothing"
str( study )
```

```
## List of 2
##  $ name: chr "Bob"
##  $ age : num 44
```

3.2.8 (*) Adding an element

```
str( study )
```

```
## List of 2
##  $ name: chr "Bob"
##  $ age : num 44
```

```
study$gender <- "male"
str( study )
```

```
## List of 3
##  $ name  : chr "Bob"
##  $ age   : num 44
##  $ gender: chr "male"
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

3.3 Basic statistical tests

```
## pulse <- read.delim("pulse.txt")
## survey <- read.delim("survey.txt")
```

3.3.1 Statistical methods with R

Statistical methods typically have their own pre-defined functions. Example: the t-test.

3.3.1.1 Statistical tests - the t test

Suppose we are interested in finding a difference between `pulse1` and `pulse2` in the pulse data. These are two variables representing measurements per individual before (pulse1) and after (pulse2) an intervention. As such, measurements are paired so we apply a paired t test:

```
t.test(pulse$pulse1, pulse$pulse2, paired=TRUE)
```

```
##
##  Paired t-test
##
## data:  pulse$pulse1 and pulse$pulse2
## t = -7.4726, df = 108, p-value = 2.172e-11
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  -26.70969 -15.51049
## sample estimates:
## mean difference
##       -21.11009
```

3.3.1.2 Accessing the result

The output on the screen is clear. But sometimes we want to extract parts of the output, say to put in a report, without having to include the entire output of the test. To do that, let us first save the result as an R object, and then check what it contains by using names().

```
res <- t.test(pulse$pulse1, pulse$pulse2, paired=TRUE)
names(res)
```

```
##  [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
##  [6] "null.value"  "stderr"      "alternative" "method"      "data.name"
```

```
class(res)
```

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

Note that the resulting object of `t.test` is a special type of object. It contains various components, including statistic and parameter. Each can be assessed by using a $ sign, as in:

```
res$statistic
```

```
##         t
## -7.472648
```

```
res$p.value
```

```
## [1] 2.171529e-11
```

Note in particular that elements in this result involve different types and lengths. Indeed:

```
class(res$p.value)
```

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

```
class(res$alternative)
```

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

```
length(res$p.value)
```

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

```
length(res$conf.int)
```

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

This means that the object resulting from the t.test function is a list too.

3.3.1.3 Wilcoxon test

Similarly to what we did above, we can apply a Wilcoxon test for these paired measurements and save results as an object. This can be done using:

```
res <- wilcox.test(pulse$pulse1, pulse$pulse2, paired=TRUE)
```

Note that the syntax is the same as the one used for the t test.

We can also assess names and the class of the object created by the Wilcoxon test:

```
names(res)
```

```
## [1] "statistic"   "parameter"   "p.value"     "null.value"  "alternative"
## [6] "method"      "data.name"
```

```
class(res)
```

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

Note that the class of the object created by the wilcox.test function is the same as that for the object created by the t.test function.

3.3.1.4 Other statistical tests

Other basic tests available include:

- `chisq.test`
- `fisher.test`
- `binom.test`

Quick task(s):

Solve the task(s), and check your solution(s) here.

3.4 Regression and formula objects

3.4.1 Formula objects

Formula objects are the way to tell R that one variable depends on another.

3.4.1.1 Basics of formula objects

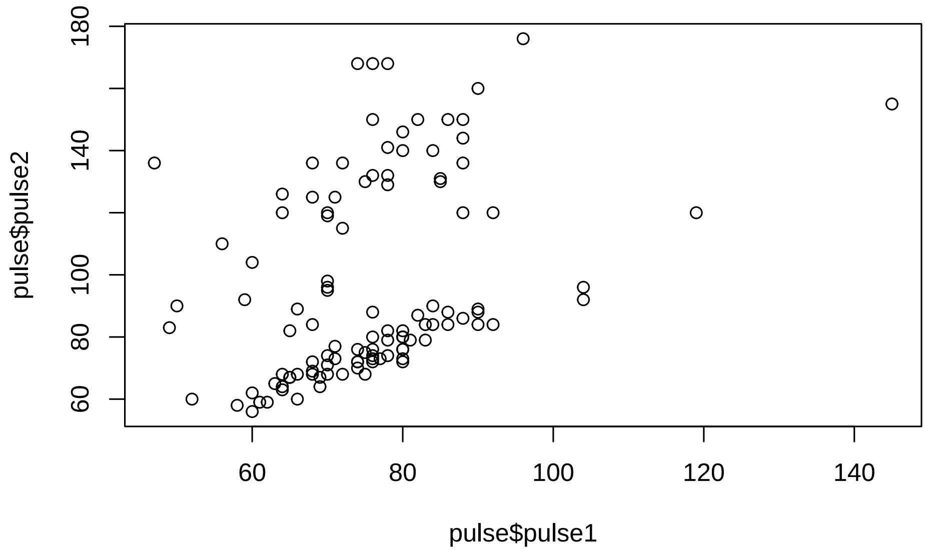
To specify a (statistical) model in which `y` depends on `x`, say

```
y ~ x
```

```
## y ~ x
```

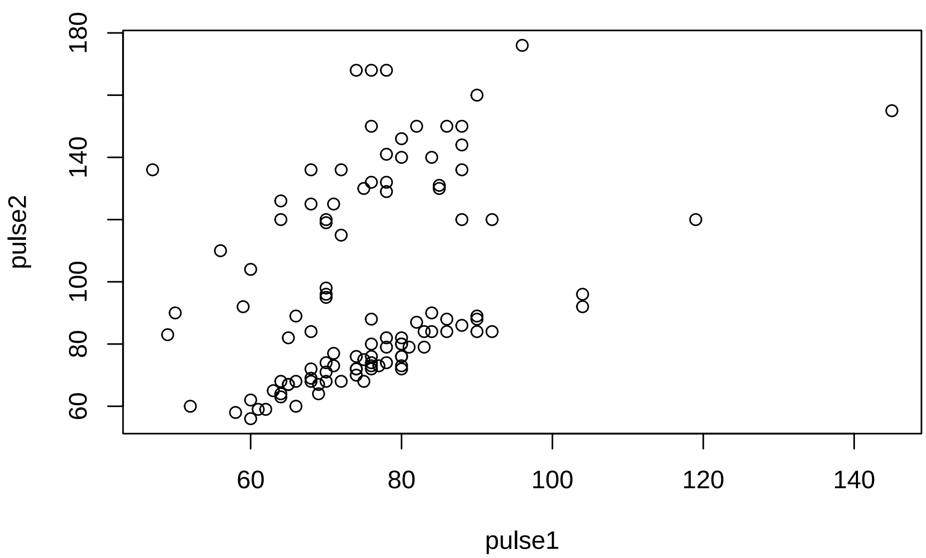
We use a formula for more readable specification of code. For example, when plotting. Instead of

```
plot(pulse$pulse1, pulse$pulse2)
```



we say

```
plot(pulse2 ~ pulse1, data=pulse)
```



Note the reverse order!

If a function allows a formula as input it always has a `data` argument. This gives the data.frame (or other environment) in which the variables in the formula are interpreted.

3.4.1.2 Use of formula objects in statistics

Formula objects can also be used in the syntax of tests and other functions, making them simpler. For example, consider the problem of comparing values of `pulse1` between males and females. We can use an unpaired t test for this, by writing:

```
pulse1.male <- pulse$pulse1[survey$gender == 'male']
pulse1.female <- pulse$pulse1[survey$gender == 'female']
t.test(pulse1.male, pulse1.female)
```

```
##
##  Welch Two Sample t-test
##
## data:  pulse1.male and pulse1.female
## t = 0.25755, df = 99.399, p-value = 0.7973
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -4.469216  5.802550
## sample estimates:
## mean of x mean of y
##  76.00000  75.33333
```

A much simpler way is to use a formula:

```
t.test(pulse1 ~ gender, data = pulse)
```

```
##
##  Welch Two Sample t-test
##
## data:  pulse1 by gender
## t = 1.3234, df = 106.3, p-value = 0.1885
## alternative hypothesis: true difference in means between group female and group male
## 95 percent confidence interval:
##  -1.667268  8.362184
## sample estimates:
## mean in group female   mean in group male
##             77.50000             74.15254
```

Many functions allow (or require!) formula as input.

3.4.1.3 (*) The formula class

A formula is just a R object.

```
class(y ~ x)
```

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

It can be stored in a variable and reused.

```
form <- pulse1 ~ gender
t.test(form, data=pulse)
```

```
##
##  Welch Two Sample t-test
##
## data:  pulse1 by gender
## t = 1.3234, df = 106.3, p-value = 0.1885
## alternative hypothesis: true difference in means between group female and group male is not eq
## 95 percent confidence interval:
##  -1.667268  8.362184
## sample estimates:
## mean in group female   mean in group male
##             77.50000             74.15254
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

3.4.2 Simple linear regression

3.4.2.1 The `lm` function

The command for linear regression is `lm` (for *linear model*). The linear model returns an object of class `lm`.

```
fit <- lm(weight ~ height, data = pulse)
```

The output of `lm` is an object of class `lm`.

```
fit
```

```
##
## Call:
## lm(formula = weight ~ height, data = pulse)
##
## Coefficients:
## (Intercept)       height
##    -27.4398       0.5465
```

```
names(fit)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

Some S3 objects have special functions defined for them. The following functions extract useful information from the `lm` object.

```
summary(fit)
```

```
##
## Call:
## lm(formula = weight ~ height, data = pulse)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -17.549  -8.197  -2.601   5.469  53.277
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -27.43977   12.73858  -2.154   0.0335 *
## height        0.54651    0.07392   7.393 3.23e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.41 on 108 degrees of freedom
## Multiple R-squared:  0.336,  Adjusted R-squared:  0.3299
## F-statistic: 54.66 on 1 and 108 DF,  p-value: 3.231e-11
```

```
coef(fit)
```

```
## (Intercept)      height
## -27.4397668   0.5465124
```

```
residuals(fit)
```

```
##      1993_A      1993_B      1993_C      1993_D      1993_E      1993_F
## -10.1068721 -12.3859463  -1.8277979   4.8698559  -3.1068721   0.8814919
##      1993_G      1993_H      1993_I      1993_J      1993_K      1993_L
##  -4.0952361  -9.9208226  -6.1882608  -4.3743103   9.5326650 -11.8394339
##      1993_M      1993_N      1993_O      1993_P      1993_Q      1993_R
##   2.5326650 -15.7580452   1.0675414  36.3349796  -9.4673350  -0.9324586
##      1993_S      1993_T      1993_U      1993_V      1993_W      1993_X
##  -7.2812855  -7.2696495  -8.1998968   0.9280359  -6.6417484   2.9745167
##      1993_Y      1993_Z      1995_A      1995_B      1995_C      1995_D
##  -9.7464092   4.7070785  -5.4673350  -6.5603597 -15.8510699 -13.8394339
##      1995_E      1995_F      1995_G      1995_H      1995_I      1995_J
##   6.8001032  14.0675414  -3.0022114   3.8117392  -3.1998968  -6.6417484
##      1995_K      1995_L      1995_M      1995_N      1995_O      1995_P
##  16.3349796   3.0791774   0.2652268  -8.2696495  -2.1998968  -6.8394339
##      1995_Q      1995_R      1995_S      1995_T      1995_U      1995_V
##  -8.1185081  -5.4673350  -1.0952361 -16.1882608   3.9861526  12.9745167
##      1996_A      1996_B      1996_C      1996_D      1996_E      1996_F
```

```
##  -7.6533844   6.1722021 -17.3626743   0.4280043  11.1722021   0.9861526
##      1996_G      1996_H      1996_I      1996_J      1996_K      1996_L
##  -9.0254833   2.8931279  12.9745167  -7.9091866 -11.0022114   4.0675414
##      1996_M      1996_N      1996_O      1996_P      1996_Q      1996_R
##   6.0675414  11.6954425  -8.1882608  31.0675414  -7.8394339 -13.2812855
##      1996_S      1996_T      1996_U      1997_A      1997_B      1997_C
## -11.1998968   9.0675414  25.9745167 -13.0836001  12.7884672   5.6024177
##      1997_D      1997_E      1997_F      1997_G      1997_H      1997_I
##   9.6140537 -14.7347732 -13.5603597  -3.1068721 -11.9673350  -6.9673350
##      1997_J      1997_K      1997_L      1997_M      1997_N      1997_O
## -10.6417484   1.0559054  -7.5603597   4.9861526   5.0675414  31.4163683
##      1997_P      1997_Q      1997_R      1997_S      1997_T      1997_U
##  -0.8277979  27.5093930  16.4163683  12.1489301 -11.0952361 -14.1998968
##      1997_V      1997_W      1998_A      1998_B      1998_C      1998_D
##  10.3001032  21.7884672  16.1605661  -7.4673350  -4.7347732  15.8117392
##      1998_E      1998_F      1998_G      1998_H      1998_I      1998_J
##  -5.9324586  -3.4673350  -2.2696495  -2.7347732  -9.3743103  53.2769261
##      1998_K      1998_L      1998_M      1998_N      1998_O      1998_P
##  -2.4673350   9.6140537 -14.6417484   3.6141170 -17.5487237 -12.0254833
##      1998_Q      1998_R
##  -0.4673350  11.3349796
```

```
fitted.values(fit)
```

```
##    1993_A    1993_B    1993_C    1993_D    1993_E    1993_F    1993_G    1993_H
## 67.106872 70.385946 63.827798 79.130144 67.106872 73.118508 61.095236 64.920823
##    1993_I    1993_J    1993_K    1993_L    1993_M    1993_N    1993_O    1993_P
## 62.188261 64.374310 65.467335 69.839434 65.467335 74.758045 70.932459 73.665020
##    1993_Q    1993_R    1993_S    1993_T    1993_U    1993_V    1993_W    1993_X
## 65.467335 70.932459 63.281286 57.269650 68.199897 49.071964 61.641748 72.025483
##    1993_Y    1993_Z    1995_A    1995_B    1995_C    1995_D    1995_E    1995_F
## 68.746409 69.292922 65.467335 66.560360 75.851070 69.839434 68.199897 70.932459
##    1995_G    1995_H    1995_I    1995_J    1995_K    1995_L    1995_M    1995_N
## 60.002211 62.188261 68.199897 61.641748 73.665020 64.920823 62.734773 57.269650
##    1995_O    1995_P    1995_Q    1995_R    1995_S    1995_T    1995_U    1995_V
## 68.199897 69.839434 73.118508 65.467335 61.095236 62.188261 66.013847 72.025483
##    1996_A    1996_B    1996_C    1996_D    1996_E    1996_F    1996_G    1996_H
## 67.653384 63.827798 58.362674 72.571996 63.827798 66.013847 72.025483 67.106872
##    1996_I    1996_J    1996_K    1996_L    1996_M    1996_N    1996_O    1996_P
## 72.025483 58.909187 60.002211 70.932459 70.932459 75.304558 62.188261 70.932459
##    1996_Q    1996_R    1996_S    1996_T    1996_U    1997_A    1997_B    1997_C
## 69.839434 63.281286 68.199897 70.932459 72.025483 55.083600 74.211533 76.397582
##    1997_D    1997_E    1997_F    1997_G    1997_H    1997_I    1997_J    1997_K
## 70.385946 62.734773 66.560360 67.106872 65.467335 65.467335 61.641748 76.944095
##    1997_L    1997_M    1997_N    1997_O    1997_P    1997_Q    1997_R    1997_S
## 66.560360 66.013847 70.932459 78.583632 63.827798 77.490607 78.583632 75.851070
```

```
##    1997_T    1997_U    1997_V    1997_W    1998_A    1998_B    1998_C    1998_D
## 61.095236 68.199897 68.199897 74.211533 69.839434 65.467335 62.734773 62.188261
##    1998_E    1998_F    1998_G    1998_H    1998_I    1998_J    1998_K    1998_L
## 70.932459 65.467335 57.269650 62.734773 64.374310  9.723074 65.467335 70.385946
##    1998_M    1998_N    1998_O    1998_P    1998_Q    1998_R
## 61.641748 23.385883 60.548724 72.025483 65.467335 73.665020
```

Note that `summary(fit)` returns itself an object in which some additional things are calculated.

```
summa <- summary(fit)
class(summa)
```

```
## [1] "summary.lm"
```

```
names(summa)
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

Most useful is the regression table

```
coef(summa)
```

```
##                Estimate  Std. Error   t value     Pr(>|t|)
## (Intercept) -27.4397668 12.73857886 -2.154068 3.345796e-02
## height        0.5465124  0.07392117  7.393178 3.231467e-11
```

and the confidence intervals for all or some regression coefficients

```
confint(fit)
```

```
##                   2.5 %     97.5 %
## (Intercept) -52.6898400 -2.1896935
## height        0.3999878  0.6930369
```

```
confint(fit, "height")
```

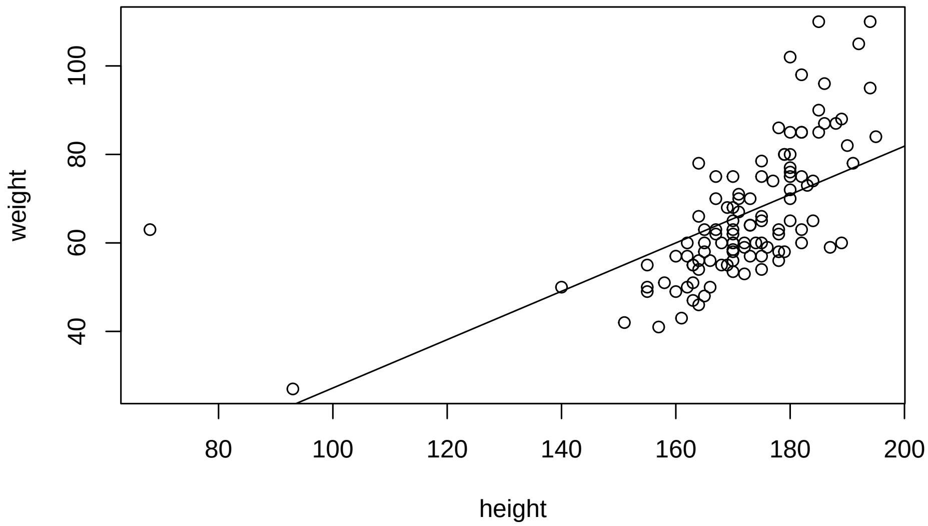
```
##            2.5 %    97.5 %
## height 0.3999878 0.6930369
```

Note that the regression coefficients are only accessible from the `summary` object, whilst the `confint` function yields confidence intervals using the fitted model object.

3.4.2.2 Visualizing a regression

We can easily visualize the regression using the same formula and fit object

```
plot(weight ~ height, data=pulse)
abline(coef(fit))
```



Quick task(s):

Solve the task(s), and check your solution(s) here.

3.4.3 Multiple regression and prediction

3.4.3.1 Multiple terms in a formula

We can have multiple terms in a formula. This way we can do multiple regression

```
fit <- lm(pulse2 ~ pulse1 + height, data=pulse)
fit
```

```
##
## Call:
## lm(formula = pulse2 ~ pulse1 + height, data = pulse)
##
## Coefficients:
## (Intercept)       pulse1       height
##    25.96489      0.86822      0.02984
```

```
summary(fit)
```

```
##
## Call:
## lm(formula = pulse2 ~ pulse1 + height, data = pulse)
##
## Residuals:
```

```
##     Min      1Q  Median      3Q     Max
## -28.89 -23.05 -17.84  28.40  72.89
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 25.96489   39.90679   0.651 0.516688
## pulse1       0.86822    0.22380   3.879 0.000182 ***
## height       0.02984    0.18427   0.162 0.871668
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 29.71 on 106 degrees of freedom
##   (1 observation deleted due to missingness)
## Multiple R-squared:  0.1309, Adjusted R-squared:  0.1145
## F-statistic: 7.981 on 2 and 106 DF,  p-value: 0.0005904
```

3.4.3.2 Predicting

We can use regression for prediction using the predict function. To predict we need two things. First, a fitted model object.

```
fit <- lm(pulse2 ~ pulse1 + height, data=pulse)
```

Second, a data.frame with new data for our covariates

```
new.data <- data.frame(pulse1=c(90, 80), height=c(173, 185))
```

Now we can predict a value for a person with these covariates

```
predict(fit, new.data)
```

```
##        1        2
## 109.2668 100.9426
```

3.4.3.3 (*) Interaction

Specifying interactions between variables in a formula: use `:` or `*`:

- `:` interaction only
- `*` interaction and main effects

Let us say that we want to explain `pulse2` by `pulse1`, `exercise` and `alcohol`, as well as a variable representing an interaction between `exercise` and `alcohol`. Two alternative ways of specifying the same model are

```
lm(pulse2 ~ pulse1 + exercise + alcohol + exercise:alcohol, data=pulse)
```

```
##
## Call:
## lm(formula = pulse2 ~ pulse1 + exercise + alcohol + exercise:alcohol,
```

```
##     data = pulse)
##
## Coefficients:
##                (Intercept)                        pulse1
##                     6.0581                        0.8911
##                exerciselow              exercisemoderate
##                    17.6346                       24.1769
##                 alcoholyes        exerciselow:alcoholyes
##                    28.4487                      -21.3348
## exercisemoderate:alcoholyes
##                   -28.8261
```

```
lm(pulse2 ~ pulse1 + exercise*alcohol, data=pulse)
```

```
##
## Call:
## lm(formula = pulse2 ~ pulse1 + exercise * alcohol, data = pulse)
##
## Coefficients:
##                (Intercept)                        pulse1
##                     6.0581                        0.8911
##                exerciselow              exercisemoderate
##                    17.6346                       24.1769
##                 alcoholyes        exerciselow:alcoholyes
##                    28.4487                      -21.3348
## exercisemoderate:alcoholyes
##                   -28.8261
```

3.4.3.4 (*) The intercept term

We see that R automatically adds an intercept term to the model. You can suppress the intercept too, by adding either `+0` or `-1` to the formula. Suppressing the intercept has different effects if there are factor variables in your model or not.

Suppression of the intercept means regression through the origin

```
lm(weight ~ 0 + height, data=pulse)
```

```
##
## Call:
## lm(formula = weight ~ 0 + height, data = pulse)
##
## Coefficients:
## height
##  0.388
```

Note that this is different if we have factors (see next part)!

Quick task(s):

Solve the task(s), and check your solution(s) here.

3.5 Factors (advanced)

3.5.1 Factors revisited

We have seen an introduction to factors in the section 'Basic data types'. Remember that they are variables that define categories. We can find out the category names involved using `levels` and tabulate factors:

```
levels(pulse$exercise)
```

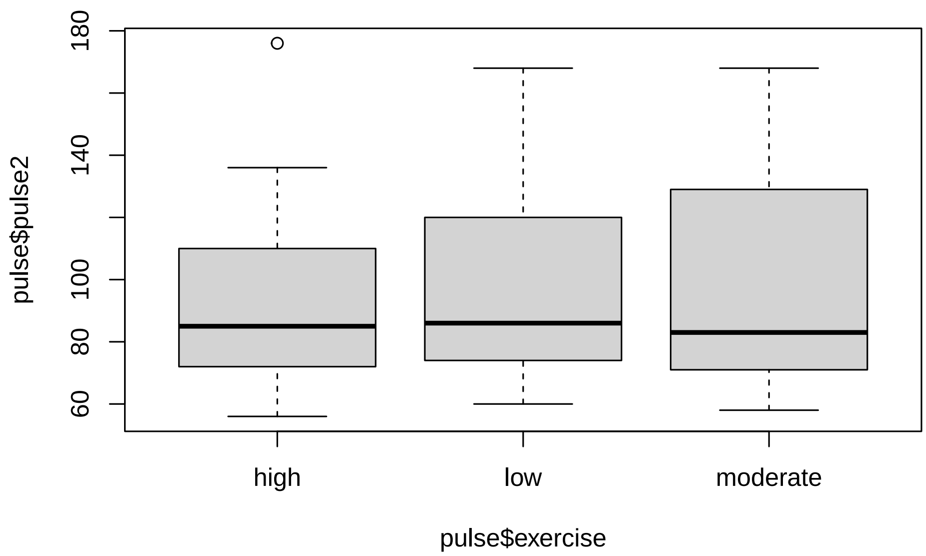
```
## NULL
```

```
table(pulse$exercise)
```

```
##
##     high      low moderate
##       14       37       59
```

Note that the category names given by its levels come typically in alphabetical order. In the example above, this order does not correspond to the intrinsic order of the categories, in which the extremes are given by `high` and `low`, with `intermediate` in the middle, rather than the last one. This is not a big problem for a table, but it is not ideal for a graph. Indeed, a boxplot of `pulse2` according to the groups defined by `exercise` looks like:

```
boxplot(pulse$pulse2 ~ pulse$exercise)
```



So we would like to re-order the factor levels so that they correspond to the intrinsic order of the categories.

3.5.1.1 Reordering a factor

To change the order of the category levels, we create the factor again by giving its levels in the correct order:

```
pulse$exercise <- factor(pulse$exercise, levels=c('high', 'moderate', 'low'))
```

We can check that the re-ordering has worked:

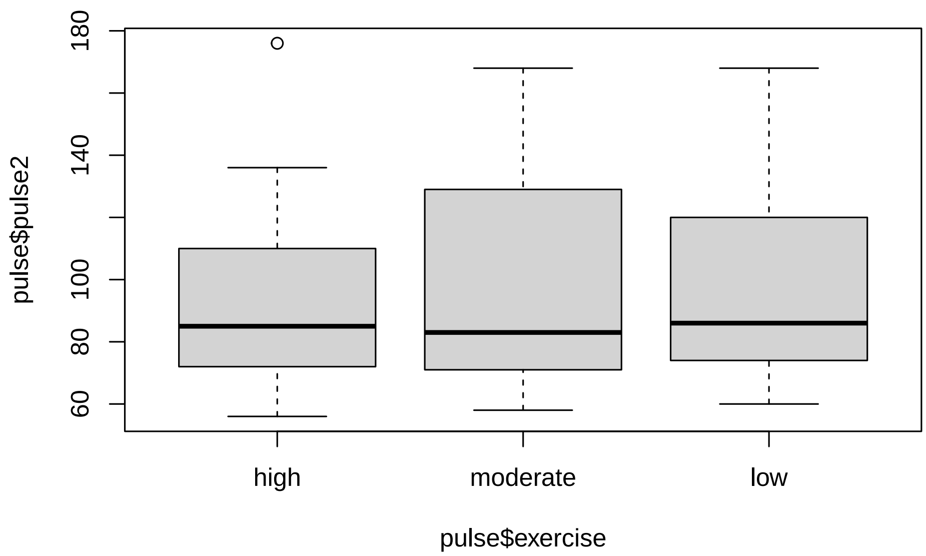
```
levels(pulse$exercise)
```

```
## [1] "high"     "moderate" "low"
```

```
table(pulse$exercise)
```

```
##
##     high moderate      low
##       14       59       37
```

```
boxplot(pulse$pulse2 ~ pulse$exercise)
```



3.5.1.2 Changing factor labels

If you want to change the category labels only, without re-ordering them, assign new values to the `levels` of a factor.

```
pulse$exercise2 <- pulse$exercise
levels(pulse$exercise2) <- c("H", "I", "L")
table(pulse$exercise2)
```

```
##
##  H  I  L
## 14 59 37
```

You are advised to check that the correspondence between observations in `exercise` and `exercise2`, for example by tabulating the old and new factors:

```
table(pulse$exercise, pulse$exercise2)
```

```
##
##             H  I  L
##   high     14  0  0
##   moderate  0 59  0
##   low       0  0 37
```

We can use similar code to merge categories of a factor. For example, to merge the categories `H` and `I`, assign to them the same label:

```
pulse$exercise3 <- pulse$exercise
levels(pulse$exercise3) <- c("H.I", "H.I" , "L")
```

3.5.1.3 Turning a continuous variable into categories

Use `cut` to categorise a continuous variable and turn into a factor variable. Note that when calling `cut` the break points between categories need to be given, including the maximum and minimum values of the original variable.

```
pulse$height4 <- cut(pulse$height, c(min(pulse$height)-1, 160, 170, 180, max(pulse$height)))
class(pulse$height4)
```

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

```
table(pulse$height4)
```

```
##
##  (67,160] (160,170] (170,180] (180,195]
##        11        38        37        24
```

You may want to change the labels to something prettier.

```
levels(pulse$height4) <- c('-160', '160-170', '170-180', '180+')
table(pulse$height4)
```

```
##
##    -160 160-170 170-180    180+
##      11      38      37      24
```

3.5.1.4 (*) Combining factors

The `:` operator can be used to make a new factor with all combinations of two (or more) factors

```
#pulse$smokes:pulse$alcohol
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

3.5.2 ANOVA and regression with factors

An ANOVA analysis can be run in R by using the results of a regression model fit, such as from `lm`.

3.5.2.1 Regression model fit

Say you fit a regression model of `pulse2` on `exercise`, which already had its categories reordered:

```
table(pulse$exercise)
```

```
##
##     high moderate      low
##       14       59       37
```

```
fit <- lm(pulse2 ~ exercise, data=pulse)
summary(fit)
```

```
##
## Call:
## lm(formula = pulse2 ~ exercise, data = pulse)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -39.45 -24.45 -12.97  27.55  82.36
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)        93.643      8.510  11.004   <2e-16 ***
## exercisemoderate    3.805      9.481   0.401    0.689
## exerciselow         3.330      9.991   0.333    0.740
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 31.84 on 106 degrees of freedom
##   (1 observation deleted due to missingness)
## Multiple R-squared:  0.001533,	Adjusted R-squared:  -0.01731
## F-statistic: 0.08139 on 2 and 106 DF,  p-value: 0.9219
```

In the results, `exercisemoderate` represents the effect of `exercise='moderate'` versus the reference category `exercise='high'`. By default, the first level of the factor is taken as the reference category, and this is often the first level in alphabetical order.

The model fit above yields tests per category of `exercise`, compared with the reference category. However, a test for the effect of the entire variable `exercise` is not directly available. This can be obtained with ANOVA (Analysis of Variance).

3.5.2.2 The ANOVA table and F-test

The ANOVA table can be obtained by using the function `anova` and the model fit `fit`:

```
anova(fit)
```

```
## Analysis of Variance Table
##
```

```
## Response: pulse2
##            Df Sum Sq Mean Sq F value Pr(>F)
## exercise    2    165   82.51  0.0814 0.9219
## Residuals 106 107461 1013.78
```

We can also compare two model fits using ANOVA. Say that we want to check if the above model fit improves by including `gender` in the model. Then we fit a model with both `exercise` and `gender`, and compare this new fit with the above one:

```
fit2 <- lm(pulse2 ~ exercise + gender, data=pulse)
anova(fit, fit2)
```

```
## Analysis of Variance Table
##
## Model 1: pulse2 ~ exercise
## Model 2: pulse2 ~ exercise + gender
##   Res.Df    RSS Df Sum of Sq     F Pr(>F)
## 1    106 107461
## 2    105 107117  1    343.83 0.337 0.5628
```

Note that `lm` by default removes any subjects which have missing values in at least one of the covariates. This means that the number of subjects in `fit` and `fit0` may be different and error returned. In that case remove subjects with missing values manually (or do imputation or something more fancy).

3.5.2.3 (*) Model fit without a reference category

When fitting a regression model, we can do without a reference category in a model fit by suppressing the intercept:

```
lm(pulse2 ~ 0 + gender, data=pulse)
```

```
##
## Call:
## lm(formula = pulse2 ~ 0 + gender, data = pulse)
##
## Coefficients:
## genderfemale    gendermale
##        98.92         95.00
```

The coefficient now represents the mean in the group (male or female), instead of a comparison between males and females (such a comparison is called a contrast).

Note that the reference category is only suppressed for the first factor in the formula:

```
lm(pulse2 ~ 0 + gender + ran, data=pulse)
```

```
##
```

```
## Call:
## lm(formula = pulse2 ~ 0 + gender + ran, data = pulse)
##
## Coefficients:
## genderfemale    gendermale        ransat
##       127.99        125.80        -51.92
```

Note that suppressing the intercept has a different effect for explanatory factors and for continuous explanatory variables.

3.5.3 (*) Generalized linear models and survival

Regression models can be run using the function `glm` (generalized linear model), which has very similar syntax to `lm`. Amongst useful models are logistic models.

3.5.3.1 (*) Logistic regression and ANOVA

For logistic regression, use `glm` with slot `family=binomial`.

```
## fit <- glm(alcohol ~ gender + smokes + exercise, family=binomial, data=pulse)
summary(fit)
```

```
##
## Call:
## lm(formula = pulse2 ~ exercise, data = pulse)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -39.45 -24.45 -12.97  27.55  82.36
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)        93.643      8.510  11.004   <2e-16 ***
## exercisemoderate    3.805      9.481   0.401    0.689
## exerciselow         3.330      9.991   0.333    0.740
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 31.84 on 106 degrees of freedom
##   (1 observation deleted due to missingness)
## Multiple R-squared:  0.001533,   Adjusted R-squared:  -0.01731
## F-statistic: 0.08139 on 2 and 106 DF,  p-value: 0.9219
```

When using `anova` in `glm`, the default is not to give a p-value. If you want it, explicitly ask for one. In case of the logistic model, the adequate way to compute the ANOVA p-value is via the likelihood ratio test (LRT):

```
anova(fit, test='LRT')
```

```
## Analysis of Variance Table
##
## Response: pulse2
##            Df Sum Sq Mean Sq F value Pr(>F)
## exercise    2    165   82.51  0.0814 0.9219
## Residuals 106 107461 1013.78
```

3.5.3.2 (*) Survival analysis

Survival analysis methods are available in the `survival` package, which is installed automatically with the base package. The syntax is similar to `lm` and `glm`, except that the response has to be a `Surv` object, built from two separate variables giving time and event.

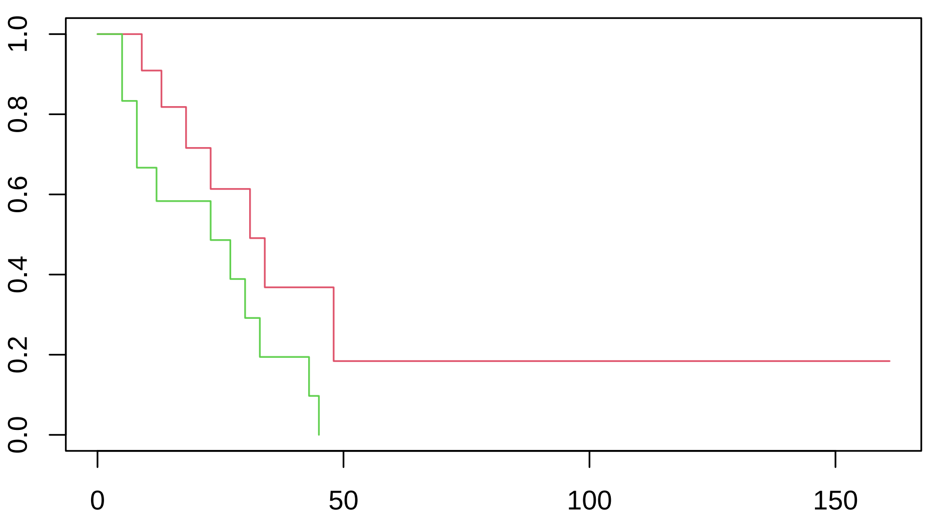
We do not have survival times in the `pulse` data, so we use the `aml` data from the `survival` package.

```
library(survival)
## ?aml
with(aml, Surv(time, status))
```

```
##  [1]   9   13   13+  18   23   28+  31   34   45+  48  161+   5    5    8    8
## [16]  12   16+  23   27   30   33   43   45
```

To draw Kaplan-Meier curves, use `survfit`:

```
fit <- survfit(Surv(time, status) ~ x, data=aml)
plot(fit, col=2:3)
```



A log-rank test can be computed using:

```
survdiff(Surv(time, status) ~ x, data=aml)
```

```
## Call:
## survdiff(formula = Surv(time, status) ~ x, data = aml)
##
##                   N Observed Expected (O-E)^2/E (O-E)^2/V
## x=Maintained     11        7    10.69      1.27       3.4
## x=Nonmaintained 12       11     7.31      1.86       3.4
##
##  Chisq= 3.4  on 1 degrees of freedom, p= 0.07
```

A Cox model can be fitted as follows:

```
coxph(Surv(time, status) ~ x, data=aml)
```

```
## Call:
## coxph(formula = Surv(time, status) ~ x, data = aml)
##
##                   coef exp(coef) se(coef)     z      p
## xNonmaintained 0.9155    2.4981   0.5119 1.788 0.0737
##
## Likelihood ratio test=3.38  on 1 df, p=0.06581
## n= 23, number of events= 18
```

As before, each object can be stored separately. In particular, the Surv object can be saved and the entire analysis run using it.

The result of coxph can be stored as an object and manipulated in most ways like a `lm` or `glm` object. In particular we can use `anova` to compare different model fits.

Quick task(s):

Solve the task(s), and check your solution(s) here.

Chapter 4

Functions

4.1 User-defined Functions

Functions are constructs that encapsulate series of statements for convenience so you do not repeat the same statements all over again when needed.

Function construct:

```
# (1) default return value is the last statment
functionName <- function(...) {
  statement
  ...
  <value>
}

# (2) return(...)
functionName <- function(...) {
  statement
  ...
  return(<value>)
}
```

Example:

The following function `add_one` increments its argument by 1:

```
add_one <- function(x) {
    x + 1
}

add_one(2)
```

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

```
add_one(-1)
```

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

Now let's update our function to calculate $x^2 + 1$:

```
square_add_one <- function(x) {
    result <- x^2    # choose a variable for temporary result
    result + 1
}

square_add_one <- function(x) {
    result <- x^2    # choose a variable for temporary result
    res1 <- result + 1
    res2 <- log(x)
    list(sq1 = res1, logx = res2)
}
square_add_one(2) # 2^2 + 1 = 5
```

```
## $sq1
## [1] 5
##
## $logx
## [1] 0.6931472
```

```
square_add_one(-1) # (-1)^2 + 1 = 2
```

```
## Warning in log(x): NaNs produced
```

```
## $sq1
## [1] 2
##
## $logx
## [1] NaN
```

The following versions of this functions are all equivalent:

```
# (v1) One operation per line.
square_add_one <- function(x) {
    result <- x^2
    result <- result + 1
    result    # The last statement is returned as the value.
}
# (v2) No additional 'result' variable needed
square_add_one <- function(x) {
    x^2 + 1
}
```

Multiple arguments Functions may take as many arguments as needed. Recall the function `add_one` defined above, now we would like to have a more `generic` function to increment by a value other than a constant `1` :

```
increment <- function(x,i) {
  # increment x by i
  x + i
}

increment(2,1)              # <=> add_one(2)
```

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

```
increment(11,2)             # increment 11 by 2
```

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

You may use the argument names explicitly:

```
increment( x = 14, i = 7 ) # increment 14 by 7
```

```
## [1] 21
```

Now let's write a function which is not already part of the base R, e.g. `odd`. This function will take as an argument a whole number and returns a logical `TRUE` if the number is an odd number and FALSE otherwise.

A whole number is odd when it is not integer divisible by 2. We can do this by taking it's remainder of integer division (`%%` operator) by 2 and see whether it is non-zero:

```
odd <- function(x) {
  x %% 2 != 0
}

odd(31)
```

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

```
odd( x = 6 )
```

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

Observations:

- User-defined functions lead to a more structured code.
- Variables declared inside a function cease to exist once the function terminates.
- Arguments are given in the same order as they are declared in the function except when the argument names are provided explicitly.

Quick task(s):

Solve the task(s), and check your solution(s) here.

4.2 (*) Control flow constructs

`conditional statements` are constructs controlling the program sequence being executed. Based on the conditions we impose into the program, we can influence the outcome.

4.2.1 Sequential execution

```
statement_1
statement_2; statement_3
```

4.2.2 Conditional execution: if

```
# Only if 'condition' variable evaluates to TRUE the
# statement_T is executed.
#
# condition is TRUE  =>  statement_A ; statement_T ; statement_B
# condition is FALSE =>  statement_A ; statement_B
statement_A
if ( condition ) {
  statement_T
}
statement_B
```

4.2.3 Conditional execution: if/else

```
# condition is TRUE  =>  statement_A ; statement_T ; statement_B
# condition is FALSE =>  statement_A ; statement_F ; statement_B
statement_A
if ( condition ) {
  statement_T
} else {
  statement_F
}
statement_B
```

Example:

Let's take the following excerpt from the World Health Organization (WHO):

An adult is a person older than 19 years of age unless national law defines a person as being an adult at an earlier age. An adolescent is a person aged 10 to

19 years inclusive. A child is a person 19 years or younger unless national law defines a person to be an adult at an earlier age. However, in these guidelines when a person falls into the 10 to 19 age category they are referred to as an adolescent (see adolescent definition). An infant is a child younger than one year of age.

Now we would like to write a function, given the age in years, to evaluate the appropriate age group label for us. For this we need conditional statements.

Important First we enumerate the age group categories:

classification	bmi	notation
adult	>19	(19, 100)
adolescent	>=10 and <=19	[10,19]
child	>=1 and <=9	[1,9]
infant	<1	(0,1)

Here is a template of the `AgeGroup` function:

```
#
# Template
#
AgeGroup <- function(x) {
  # x : is age in year
}

AgeGroup(0)   #  "infant"
AgeGroup(10)  #  "adolescent"
AgeGroup(9)   #  "child"
AgeGroup(20)  #  "adult"
```

Scenario 1 : if

Let's first assume that everybody is a child:

```
AgeGroup <- function(x) {
  # x : is age in year
  theLabel <- "child"
  theLabel
}

AgeGroup(0)   #  "infant"
```

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

```
AgeGroup(10)  #  "adolescent"
```

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

```
AgeGroup(9)   #  "child"
```

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

```r
AgeGroup(20)  #  "adult"
```

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

Next we add a single conditional to capture `adult` label:

```r
# child and adult
AgeGroup <- function(x) {
  # x : is age in year
  theLabel <- "child"
  if ( x > 19 ) {
    theLabel <- "adult"
  }
  theLabel
}

AgeGroup(0)   #  "infant"
```

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

```r
AgeGroup(10)  #  "adolescent"
```

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

```r
AgeGroup(9)   #  "child"
```

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

```r
AgeGroup(20)  #  "adult"
```

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

Finally we add the conditional statement to capture the labels `adolescent` and `infant` in the same way:

```r
# version final
AgeGroup <- function(x) {
  # x : is age in year
  theLabel <- "child"
  if ( x > 19 ) {
    theLabel <- "adult"
  }
  if ( x >= 10  & x <= 19 ) {
    theLabel <- "adolescent"
  }
  if ( x < 1 ) {
    theLabel <- "infant"
  }
  theLabel
```

```
}

AgeGroup(0)   #  "infant"
```

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

```
AgeGroup(10)  #  "adolescent"
```

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

```
AgeGroup(9)   #  "child"
```

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

```
AgeGroup(20)  #  "adult"
```

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

Scenario 2 : if/else

`if` conditional can be extended with `else` part. This is a more concise way to capture labels:

```
AgeGroup <- function(x) {
  # x : is age in year
  # labels `adult` and `child`
  if ( x > 19 ) {
    theLabel <- "adult"
  } else {
    theLabel <- "child"
  }
  theLabel
}

AgeGroup(0)   #  "infant"
```

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

```
AgeGroup(10)  #  "adolescent"
```

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

```
AgeGroup(9)   #  "child"
```

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

```
AgeGroup(20)  #  "adult"
```

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

Now as before we add other conditionals to capture `adolescent` and `infant` labels:

```
AgeGroup <- function(x) {
  # x : is age in year
  # labels `adult` and `child`
  if ( x > 19 ) {
    theLabel <- "adult"
  } else {
    theLabel <- "child"
  }
  # label `adolescent`
  if ( x >= 10  & x <= 19 ) {
    theLabel <- "adolescent"
  }
  # label `infant`
  if ( x < 1 ) {
    theLabel <- "infant"
  }
  theLabel
}

AgeGroup(0)    #  "infant"
```

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

```
AgeGroup(10)   #  "adolescent"
```

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

```
AgeGroup(9)    #  "child"
```

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

```
AgeGroup(20)   #  "adult"
```

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

Scenario 2 : if/else/if . . .

It is also possible to have cascading if/else construct with multiple conditionals:

```
AgeGroup <- function(x) {
  # x : is age in year
  if ( x > 19 ) {
    theLabel <- "adult"
  } else if ( x >= 10  & x <= 19 ) {
    theLabel <- "adolescent"
  } else if ( x < 1 ) {
    theLabel <- "infant"
  } else {
    theLabel <- "child"
```

```
  }
  theLabel
}

AgeGroup(0)   #  "infant"
```

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

```
AgeGroup(10)  #  "adolescent"
```

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

```
AgeGroup(9)   #  "child"
```

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

```
AgeGroup(20)  #  "adult"
```

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

Quick task(s):

Solve the task(s), and check your solution(s) here.

4.3 apply family: apply, lapply, sapply, tapply

The apply family functions as the name suggests are a mechanism to apply a function to a sequence of predefined arguments.

4.3.1 apply(X, MARGIN, FUN, ...)

`X` is a matrix (or data.frame) and the `MARGIN` is either 1 or 2 corresponding to row and column respectively. It applies the function `FUN` to each row or column depending on MARGIN value. The returned value is an array structure, i.e either a vector or a matrix, depending on the function `FUN` value.

If `FUN` returns a single value and not a vector then the return value is a vector:

```
# apply function to columns (MARGIN=2)
apply(pulse[,c("height","weight","age")],2, mean)
```

```
##    height    weight       age
## 171.58182  66.33182  20.56364
```

Applying mean to the columns pulse1 and pulse2 results into NA:

```
apply(pulse[,c("pulse1","pulse2")],2, mean)
```

```
## pulse1 pulse2
##     NA     NA
```

This is cause by missing values (NA's) in pulse1 and pulse2. Note that it the function mean's behavior that causes this:

```
mean( c(3,4,1,5,10) )
```

```
## [1] 4.6
```

```
mean( c(3,4,1,5,10,NA) )
```

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

Quiz How can we get around NA's and produce mean values for pulse measurements? (Hint: ?mean)

...

4.3.2 sapply(X, FUN, ...)

This is an easy way of running any function for a range of values.

```
sapply(1:4, odd)  # <=>   unlist( lapply(1:4,odd) )
```

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

This is in fact a simplified, user-friendly `lapply()`, which we will see next.

Quick task(s):

Solve the task(s), and check your solution(s) here.

4.3.3 lapply(X, FUN, ...) : apply a function to a list/vector

`X` is the sequence (vector/list) of elements on which the function `FUN` is applied to each element. The return value is always a list.

```
lapply(1:2, add_one)              # vector as input
```

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

```
lapply(list(1,2), square_add_one)  # list as input
```

```
## [[1]]
## [[1]]$sq1
## [1] 2
##
## [[1]]$logx
## [1] 0
##
##
## [[2]]
## [[2]]$sq1
## [1] 5
##
## [[2]]$logx
## [1] 0.6931472
```

Note that only the function name, e.g. `add_one`, is given without it's argument, `lapply` is aware of the function's argument and will instantiate the function first with `1` and then `2`.

Quiz Rewrite `lapply(1:2, add_one)` by the function `increment`.

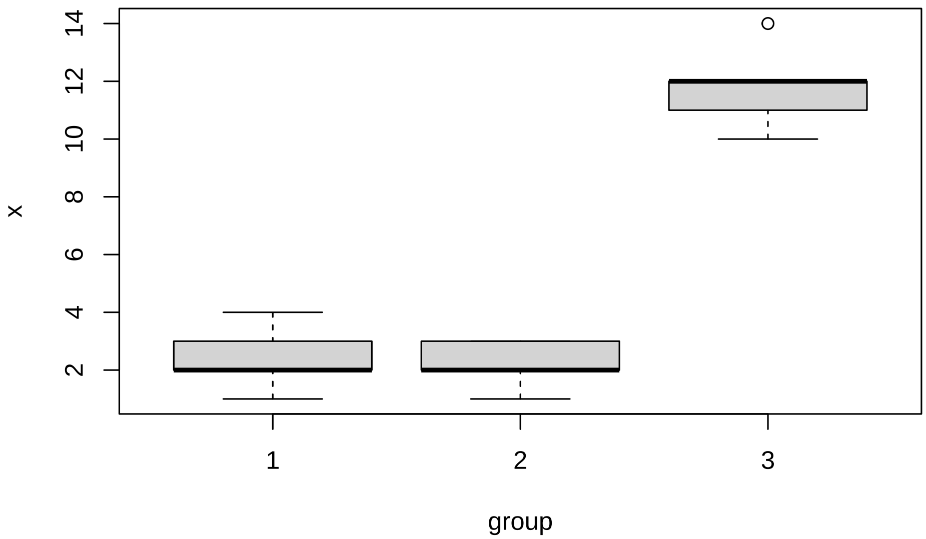
4.3.4 tapply(X, INDEX, FUN, ...) : apply a function to a vector, according to groups of INDEX

Consider the data in the vector `x` and the grouping variable `group`:

```
x <- c(1, 2, 2, 4,  3, 1, 3, 2, 2, 3, 10, 12, 11, 14, 12)
group <- factor(rep(1:3, each = 5))
```

We can display the data by using a boxplot:

```
boxplot(x ~ group)
```



To compute the means of values of `x` within groups defined by `group` we use:

```
tapply(x, INDEX = group, FUN = mean)
```

```
##    1    2    3
##  2.4  2.2 11.8
```

Anonymous functions Are functions used in R expressions without being declared with a name:

```
# odd: test whether a number is odd (named function)
lapply(1:4,odd)
```

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

```
# odd: test whether a number is odd (anonymous function)
lapply(1:4, function(x) { x %% 2 != 0 } )
```

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

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

4.4 Type checking

R language is not a *strictly typed* language. This means the programmer is responsible for making sure that variables involved in an assignment are of the same type. For example :

```
apple <- "123"  # character
pear  <- 123    # numeric
apple == pear   # ?
```

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

```
apple +  pear   # ?
```

```
## Error in apple + pear: non-numeric argument to binary operator
```

here the last two operations should both produce an error because `apple` and `pear` are not of the same type, however R is more liberal towards logical comparison and produces a result even though the variables compared are not of the same type. The mechanism is called `type coercion` and the full detail is beyond the scope of this course but in short what happens here is that first `pear` is converted from numeric to character to make the comparison possible and then a lexicographical comparison (dictionary order) is carried out, i.e. :

```
"123" == "123" # character comparison
```

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

4.4.1 is.* family functions

These are functions created to check a characteristic in/of the data. Some examples are:

```
# type checkinh
is.numeric("a")
```

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

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

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

```
is.character("b")
```

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

```
is.character( data.frame() )
```

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

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

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

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

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

```
# missing values
is.na(NA)
```

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

```
is.na( c(3,NA,0,-1,NA) )
```

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

All these functions return a logical objetc as response. Note that both:

```
is.na(3)
```

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

and

```
is.na(NA)
```

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

return an object with a single entry, whilst

```
is.na(c(3,NA,0,-1,NA))
```

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

returns an object which has one logical value for each entry in the input object. This function not only works with vectors as input, but also with matrices. Consider for example the following matrix, which involves some `NA` entries:

```
mat <- matrix(1:9, nrow = 3, ncol = 3)
mat[1, 2] <- mat[3, 2] <- NA
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1   NA    7
## [2,]    2    5    8
## [3,]    3   NA    9
```

Then if we apply `is.na()` to this object, we get:

```
is.na(mat)
```

```
##       [,1]  [,2]  [,3]
## [1,] FALSE  TRUE FALSE
## [2,] FALSE FALSE FALSE
## [3,] FALSE  TRUE FALSE
```

which is an object with the same dimensions as the input object `mat`.

Quiz Compute the number of `NA` entries per row and per column of `mat`.

4.4.2 (*) stop(...)/warning(...)

When to issue:

- **Warnings** : possibility of recovery and no potential harm to the end result
- **Errors** : no possibility for recovery and potential harm to the end result

```
stop("your error message !")
```

```
## Error in eval(expr, envir, enclos): your error message !
```

```
warning("your warning message !")
```

```
## Warning: your warning message !
```

Example:

Recall the function `AgeGroup` which takes age as argument. The function as is now covers all positive numbers but there is a flaw in the function, it can not handle negative input.

```
AgeGroup(-10)
```

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

which is an error. The function needs to be modified to prevent the erroneous answer as follows:

```
AgeGroup <- function(x) {
  # x : is age in year
  # error on x<0
  if (x<0)
    stop("invalid age !")
  # x >= 0
  if ( x > 19 ) {
    theLabel <- "adult"
  } else if ( x >= 10  & x <= 19 ) {
    theLabel <- "adolescent"
  } else if ( x < 1 ) {
```

```
    theLabel <- "infant"
  } else {
    theLabel <- "child"
  }
  theLabel
}

AgeGroup(-10)
```

```
## Error in AgeGroup(-10): invalid age !
```

Quiz What about `AgeGroup(1000)` ?

Quick task(s):

Solve the task(s), and check your solution(s) here.

4.5 (*) R programming

4.5.1 General coding conventions

- Coding conventions: to improve *readability* and *maintenance*
  - Identifiers: function names : AgeGroup or age_group variable names : theLabel
  - Line Length: maximum 70-80 characters
  - Indentation
  - Curly Brace
    - first on same line, last on own line
    - else: Surround else with braces
  - Assignment: use <-, not =
  - Semicolons: don't use them
  - General layout and ordering (library, functions, main)
- Use existing R functions if possible.
- Write generic parts as functions for reuse
- Inline documentation

4.5.2 Finding, installing and loading packages

Package

- Is a bundle of function(s), possibly with data and binary code.
- R comes with packages already installed, e.g. **base**, **utils**, **stats**, **methods**, etc.
- External packages can be installed and loaded into the workspace

Main sources

- Comprehensive R Archive Network (CRAN)
- Bioconductor (genomic data)

Links

- http://cran.r-project.org
- http://www.bioconductor.org
- https://www.tidyverse.org

Appendix

Character string processing & Pattern matching

Learning goals

- Character string manipulation : string concatenation,splitting, etc.
- Pattern matching and replacement

Quotes and escape characters (characters with special meaning)

`\n`	newline	`\v`	vertical tab
`\r`	carriage return	`\\`	backslash \
`\t`	tab	`\'`	ASCII apostrophe '
`\b`	backspace	`\"`	ASCII quotation mark "
`\a`	alert (bell)	`` \` ``	ASCII grave accent (backtick) `
`\f`	form feed	`\\.`	dot (escaped twice ; special meaning in RE)

Character string

Character string or simply `character`, as it is called in R, is a sequence of characters and a character vector is vector of character strings.

```
( cs <- "This is a character string" )
```

```
## [1] "This is a character string"
```

```
( cv <- c(cs, "and this is another !") )
```

```
## [1] "This is a character string" "and this is another !"
```

Single and double quotes can be used interchangeably, however, double quotes are preferred.

```
"string"
```

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

```
'string'
```

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

```
"'strings'"
```

```
## [1] "'strings'"
```

```
'"string"'
```

```
## [1] "\"string\""
```

Formatting the output with `print` and `cat` functions. The function `print` does more formatting than `cat`. On the other hand `cat` interprets escape characters such as whitespaces [\t\n\r,...], note that the behavior will differ depending on the platform. The function `cat` is useful and often used to print progress and/or debugging information in functions.

```
print(1:50)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

```
cat(1:50)
```

```
## 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

```
print("separate\nlines")
```

```
## [1] "separate\nlines"
```

```
cat("separate\nlines")
```

```
## separate
## lines
```

```
cat("column1\tcolumn2")
```

```
## column1  column2
```

Some useful function:

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

```
month.name
```

```
##  [1] "January"   "February"  "March"     "April"     "May"       "June"
```

```
##  [7] "July"      "August"    "September" "October"   "November"  "December"
```

```
month.abb
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

paste(…, sep = ” “, collapse = NULL):

Concatenates one or more vectors into a character vector.

```
paste("Bonnie","@", "lumc.nl")
```

```
## [1] "Bonnie @ lumc.nl"
```

```
paste("Travis","@", "lumc.nl", sep="")
```

```
## [1] "Travis@lumc.nl"
```

```
paste("Travis", "lumc.nl", sep="@")
```

```
## [1] "Travis@lumc.nl"
```

```
paste("Travis", 1,"@", "lumc.nl", sep="") # convert numeric to character
```

```
## [1] "Travis1@lumc.nl"
```

```
paste("Travis","@","lumc.nl",";","Bonnie","@","lumc.nl", sep="")
```

```
## [1] "Travis@lumc.nl;Bonnie@lumc.nl"
```

```
paste(c("Bonnie","Travis"),"@", "lumc.nl", sep="") # recycling
```

```
## [1] "Bonnie@lumc.nl" "Travis@lumc.nl"
```

Recycling occurs in expressions involving multiple vectors of different sizes. The rule is that smaller vectors are recycled, partially if necessary, as often as possible to match the size of the largest vector.

```
paste(c("Bonnie","Travis"),"@", "lumc.nl", sep="")
```

```
## [1] "Bonnie@lumc.nl" "Travis@lumc.nl"
```

```
paste(c("Bonnie","Travis"),"@", "lumc.nl", sep="", collapse = ";")
```

```
## [1] "Bonnie@lumc.nl;Travis@lumc.nl"
```

```
paste(c("Bonnie","Travis"), "lumc.nl", sep="@", collapse = ";")
```

```
## [1] "Bonnie@lumc.nl;Travis@lumc.nl"
```

Pulse data set

```
head(pulse)
```

```
##            name height weight age gender smokes alcohol exercise ran pulse1
## 1993_A   Bonnie    173     57  18 female     no     yes moderate sat     86
## 1993_B  Melanie    179     58  19 female     no     yes moderate ran     82
## 1993_C Consuelo    167     62  18 female     no     yes     high ran     96
## 1993_D   Travis    195     84  18   male     no     yes     high sat     71
## 1993_E    Lauri    173     64  18 female     no     yes      low sat     90
## 1993_F   George    184     74  22   male     no     yes      low ran     78
##        pulse2 year exercise2 exercise3 height4
## 1993_A     88 1993         I       H.I 170-180
## 1993_B    150 1993         I       H.I 170-180
## 1993_C    176 1993         H       H.I 160-170
## 1993_D     73 1993         H       H.I    180+
## 1993_E     88 1993         L         L 170-180
## 1993_F    141 1993         L         L    180+
```

```
allNames <- as.vector(pulse$name)
head(allNames)
```

```
## [1] "Bonnie"   "Melanie"  "Consuelo" "Travis"   "Lauri"    "George"
```

```
length(allNames)
```

```
## [1] 110
```

```
( pulseNames <- sample(allNames, size = 5) ) # select randomly 5 names from allNames
```

```
## [1] "Ernest"  "Adeline" "John"    "Crystal" "Erik"
```

```
domains <- c("lumc.nl", "leidenuniv.nl", "vumc.nl")
emails  <- paste(pulseNames,domains, sep="@")
emails
```

```
## [1] "Ernest@lumc.nl"        "Adeline@leidenuniv.nl" "John@vumc.nl"
## [4] "Crystal@lumc.nl"       "Erik@leidenuniv.nl"
```

tolower, toupper, nchar

```
toupper(emails) # convert to uppercase
```

```
## [1] "ERNEST@LUMC.NL"        "ADELINE@LEIDENUNIV.NL" "JOHN@VUMC.NL"
## [4] "CRYSTAL@LUMC.NL"       "ERIK@LEIDENUNIV.NL"
```

```
tolower(emails) # convert to lowercase
```

```
## [1] "ernest@lumc.nl"        "adeline@leidenuniv.nl" "john@vumc.nl"
## [4] "crystal@lumc.nl"       "erik@leidenuniv.nl"
```

```
nchar(emails)   # nr. of characters in string
```

```
## [1] 14 21 12 15 18
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

Split character string : strsplit(x, split, ...):

The `strsplit` function splits each element of the character vector `x` by substring `split` into substrings and returns a list of character vectors as the result.

```
strsplit("Bonnie@lumc.nl", "@")
```

```
## [[1]]
## [1] "Bonnie"  "lumc.nl"
```

```
unlist( strsplit("Bonnie@lumc.nl", "@") )
```

```
## [1] "Bonnie"  "lumc.nl"
```

```
strsplit(c("Bonnie@lumc.nl","Melanie@lumc.nl"), "@")
```

```
## [[1]]
## [1] "Bonnie"  "lumc.nl"
##
## [[2]]
## [1] "Melanie" "lumc.nl"
```

```
strsplit("Bonnie@lumc.nl", "n")
```

```
## [[1]]
## [1] "Bo"       ""         "ie@lumc." "l"
```

```
strsplit("Bonnie@lumc.nl", "") # useful
```

```
## [[1]]
##  [1] "B" "o" "n" "n" "i" "e" "@" "l" "u" "m" "c" "." "n" "l"
```

```
userDomains <- strsplit(emails,"@")
userDomains
```

```
## [[1]]
## [1] "Ernest"  "lumc.nl"
##
## [[2]]
## [1] "Adeline"       "leidenuniv.nl"
##
## [[3]]
## [1] "John"    "vumc.nl"
##
```

```
## [[4]]
## [1] "Crystal" "lumc.nl"
##
## [[5]]
## [1] "Erik"          "leidenuniv.nl"
```

rbind(. . .), cbind(. . .) : combine by rows or columns

```
letters[1:3]
```

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

```
LETTERS[1:3]
```

```
[1] "A" "B" "C"
```

```
rbind(letters[1:3], LETTERS[1:3])
```

```
     [,1] [,2] [,3]
[1,] "a"  "b"  "c"
[2,] "A"  "B"  "C"
```

```
cbind(letters[1:3], LETTERS[1:3])
```

```
     [,1] [,2]
[1,] "a"  "A"
[2,] "b"  "B"
[3,] "c"  "C"
```

strsplit(x, split, . . .)

```
userDomains <- strsplit(emails,"@")
userDomains
```

```
## [[1]]
## [1] "Ernest"  "lumc.nl"
##
## [[2]]
## [1] "Adeline"       "leidenuniv.nl"
##
## [[3]]
## [1] "John"    "vumc.nl"
##
## [[4]]
## [1] "Crystal" "lumc.nl"
##
## [[5]]
## [1] "Erik"          "leidenuniv.nl"
```

```
rbind(userDomains[[1]],userDomains[[2]],
      userDomains[[3]],userDomains[[4]],
      userDomains[[5]])
```

```
##      [,1]      [,2]
## [1,] "Ernest"  "lumc.nl"
## [2,] "Adeline" "leidenuniv.nl"
## [3,] "John"    "vumc.nl"
## [4,] "Crystal" "lumc.nl"
## [5,] "Erik"    "leidenuniv.nl"
```

do.call(what, args, . . .) : execute function on list of arguments

```
do.call(rbind, userDomains) # <=>  rbind(userDomains[[1]],userDomains[[2]],...)
```

```
     [,1]      [,2]
[1,] "Ernest"  "lumc.nl"
[2,] "Adeline" "leidenuniv.nl"
[3,] "John"    "vumc.nl"
[4,] "Crystal" "lumc.nl"
[5,] "Erik"    "leidenuniv.nl"
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

---

grep(pattern, x, ignore.case = FALSE, value = FALSE, . . .)

Search for matches of `pattern` in strings of character vector `x`.

```
emails
```

```
[1] "Ernest@lumc.nl"        "Adeline@leidenuniv.nl" "John@vumc.nl"
[4] "Crystal@lumc.nl"       "Erik@leidenuniv.nl"
```

```
grep("Dona", emails)
```

```
integer(0)
```

```
grep("Dona", emails, value = TRUE)
```

```
character(0)
```

```
grep("dona", emails, value = TRUE) # case-sensitive
```

```
character(0)
```

```
grep("dona", emails, value = TRUE, ignore.case = TRUE)
```

```
character(0)
```

```
grep("dona", emails, value = TRUE, ignore.case = TRUE, invert = TRUE)
```

```
[1] "Ernest@lumc.nl"        "Adeline@leidenuniv.nl" "John@vumc.nl"
[4] "Crystal@lumc.nl"       "Erik@leidenuniv.nl"
```

grepl(pattern, x, ignore.case = FALSE, . . .)

Same functionality as `grep` except it returns a logical vector of matches found.

```
emails
```

```
[1] "Ernest@lumc.nl"        "Adeline@leidenuniv.nl" "John@vumc.nl"
[4] "Crystal@lumc.nl"       "Erik@leidenuniv.nl"
```

```
foundSubject <- grepl("Dona", emails)
foundSubject
```

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

```
emails[foundSubject]       # value
```

```
character(0)
```

```
emails[ ! foundSubject ]   # invert
```

```
[1] "Ernest@lumc.nl"        "Adeline@leidenuniv.nl" "John@vumc.nl"
[4] "Crystal@lumc.nl"       "Erik@leidenuniv.nl"
```

substr(x, start, stop) : extract/replace substrings

```
substr(x = "abc",start = 1,stop = 1)
```

```
[1] "a"
```

```
substr("abc",1,nchar("abc"))
```

```
[1] "abc"
```

```
(abcs <- rep("abc",3))
```

```
[1] "abc" "abc" "abc"
```

```
substr(abcs,1,1:nchar("abc"))
```

```
[1] "a"   "ab"  "abc"
```

```r
(e <- head(emails))
```

```
[1] "Ernest@lumc.nl"        "Adeline@leidenuniv.nl" "John@vumc.nl"
[4] "Crystal@lumc.nl"       "Erik@leidenuniv.nl"
```

```r
substr(e,1,1)
```

```
[1] "E" "A" "J" "C" "E"
```

```r
tolower(substr(e,1,1))
```

```
[1] "e" "a" "j" "c" "e"
```

```r
substr(e,1,1) <- tolower(substr(e,1,1)) # replace first character with its lowercase
e
```

```
[1] "ernest@lumc.nl"        "adeline@leidenuniv.nl" "john@vumc.nl"
[4] "crystal@lumc.nl"       "erik@leidenuniv.nl"
```

gsub/sub(pattern, replacement, x, ignore.case = FALSE, ...)

```r
sub("@", "(at)", emails)
```

```
[1] "Ernest(at)lumc.nl"        "Adeline(at)leidenuniv.nl"
[3] "John(at)vumc.nl"          "Crystal(at)lumc.nl"
[5] "Erik(at)leidenuniv.nl"
```

```r
sub("\\.", "\n", "git.lumc.nl")   # first occurence only
```

```
[1] "git\nlumc.nl"
```

```r
gsub("\\.", "\n", "git.lumc.nl") # global: apply to all occurrences
```

```
[1] "git\nlumc\nnl"
```

```r
cat( gsub("\\.", "\n", "git.lumc.nl") )
```

```
git
lumc
nl
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

Extra exercises

Quick task(s):

Solve the task(s), and check your solution(s) here.

S3 and S4 classes

Learning goals

- General understanding of data objects, in particular objects from S3 and S4 classes.

- How to recognize and access S3 and S4 classes

Objects

- Object is a piece of data with a type (class)
- Basic types and precedence
  - NULL < raw < `logical < integer < double < character` < list < expression
  - `is.<basic type>`, `as.<basic type>` functions for type test and conversion respectively.
  - `typeof` function
- Object may be associated to methods/functions (Object Orientation : S3/S4)
  - S3 : ad hoc
    - most objects in `base` and `stats` and R core
  - S4 : formal/strict
    - e.g. `Bioconductor`

S3

Most objects in `base` and `stats` are of S3 class and are almost always based on `list`, but not necessarily.

```
res <- lm(extra ~ group, data=sleep)
res
```

```
##
## Call:
## lm(formula = extra ~ group, data = sleep)
##
## Coefficients:
```

```
## (Intercept)      group2
##        0.75        1.58
```

```
class(res)
```

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

You will recognize a S3 class via an explicit attribute `class`:

```
attributes(res)
```

```
## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
##
## $class
## [1] "lm"
```

There are many generic functions in R such as `print`, `plot`, `summary` etc. that behave differently based on the class of an object:

```
methods(plot)
```

```
##  [1] plot.aareg*         plot.acf*           plot.correspondence*
##  [4] plot.cox.zph*       plot.data.frame*    plot.decomposed.ts*
##  [7] plot.default*       plot.dendrogram*    plot.density*
## [10] plot.ecdf*          plot.factor*        plot.formula*
## [13] plot.function*      plot.hclust*        plot.histogram*
## [16] plot.HoltWinters*   plot.isoreg*        plot.lda*
## [19] plot.lm*            plot.mca*           plot.medpolish*
## [22] plot.mlm*           plot.ppr*           plot.prcomp*
## [25] plot.princomp*      plot.profile*       plot.profile.nls*
## [28] plot.raster*        plot.ridgelm*       plot.shingle*
## [31] plot.spec*          plot.spline*        plot.stepfun*
## [34] plot.stl*           plot.Surv*          plot.survfit*
## [37] plot.table*         plot.trellis*       plot.ts*
## [40] plot.tskernel*      plot.TukeyHSD*      plot.xyVector*
## see '?methods' for accessing help and source code
```

S4

S4 objects are more structured and more strict than S3 objects. They are not so popular with packages on CRAN, but very popular for packages on Bioconductor. Let's look at an example from Bioconductor

```
source("https://bioconductor.org/biocLite.R")
biocLite(c("graph", "Rgraphviz"))
```

The packages graph and Rgraphviz are for working with graphs and visualizing them.

```
#install.packages("BiocManager")
#BiocManager::install("Rgraphviz")
library(graph) ; library(Rgraphviz)
```

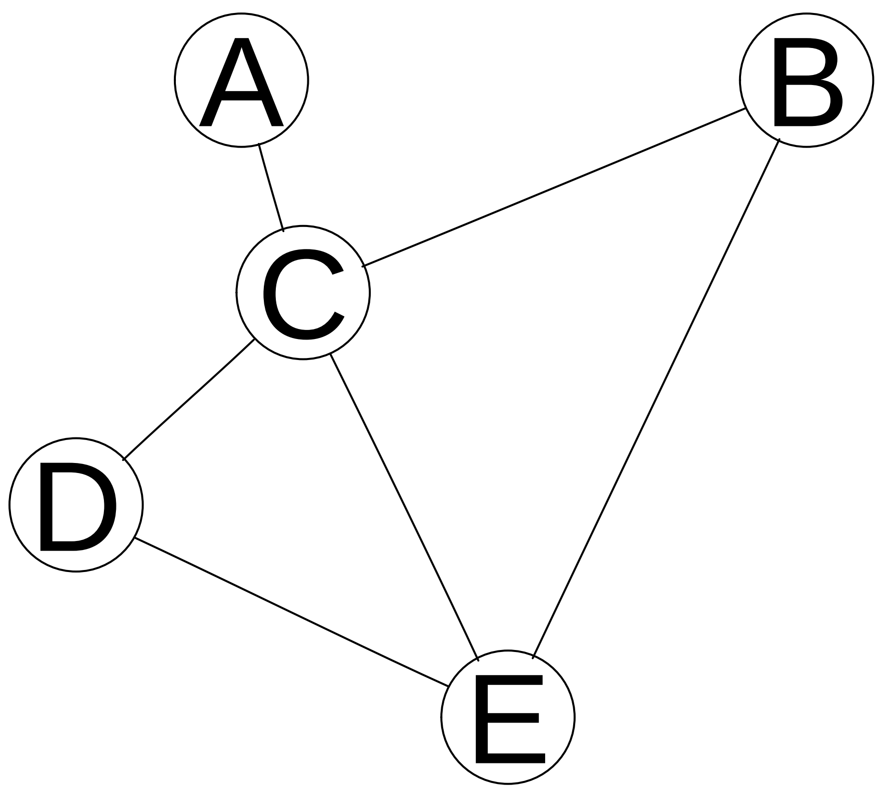
```
g1 <- randomEGraph(LETTERS[1:5], edges=6)
g1
```

```
## A graphNEL graph with undirected edges
## Number of Nodes = 5
## Number of Edges = 6
```

```
class(g1)
```

```
## [1] "graphNEL"
## attr(,"package")
## [1] "graph"
```

```
plot(g1)
```



Note the attr(,“package”) at the bottom. This shows that it is an S4, not an S3 object. To check explicitly

```
isS4(g1)
```

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

There is no isS3. Things you may expect to work for these objects from S3 don't:

```
names(g1)
```

```
## NULL
```

```
getSlots('graphNEL') # class name
```

```
##        nodes        edgeL      edgeData      nodeData    renderInfo     graphData
##     "vector"       "list"    "attrData"    "attrData" "renderInfo"        "list"
```

With S4 object you can directly access the contents with `@` but you are not meant to:

```
g1@nodes
```

```
## [1] "A" "B" "C" "D" "E"
```

There is always a help file for an S4 object which lists all the methods you can use:

```
class?graphNEL
?graphNEL
```

Quick task(s):

Solve the task(s), and check your solution(s) here.

Making errors the right way

There are two types of programming errors: the annoying ones and the dangerous ones.

Annoying errors

These are errors that yield an error message. Such errors can be avoided by including flags in your script, to check that all objects are as they are expected, their classes correspond to what they should be, etc. For example, when reading data into R, it should always be checked what was read, its dimensions, and the variable types. One such useful summary is given by `str()`:

```
pul <- read.table("data/pulse.txt")
str(pul)
```

```
## 'data.frame':    110 obs. of  12 variables:
##  $ name    : chr  "Bonnie" "Melanie" "Consuelo" "Travis" ...
##  $ height  : int  173 179 167 195 173 184 162 169 164 168 ...
##  $ weight  : num  57 58 62 84 64 74 57 55 56 60 ...
##  $ age     : int  18 19 18 18 18 22 20 18 19 23 ...
##  $ gender  : chr  "female" "female" "female" "male" ...
##  $ smokes  : chr  "no" "no" "no" "no" ...
##  $ alcohol : chr  "yes" "yes" "yes" "yes" ...
##  $ exercise: chr  "moderate" "moderate" "high" "high" ...
##  $ ran     : chr  "sat" "ran" "ran" "sat" ...
##  $ pulse1  : int  86 82 96 71 90 78 68 71 68 88 ...
##  $ pulse2  : int  88 150 176 73 88 141 72 77 68 150 ...
##  $ year    : int  1993 1993 1993 1993 1993 1993 1993 1993 1993 1993 ...
```

It is also a good idea to make graphs (or tables, if more appropriate) of variables at different stages, to check if values are as expected. In fact, any summary is helpful. Here we could compute the mean for some numeric columns, the 2nd and 3rd ones, in `pul`:

```
apply(pul[, 2:3], 2, mean)
```

```
##    height    weight
## 171.58182  66.33182
```

If you have different objects with same row or column names, it is a good idea to check that this property is preserved as well.

How to correct (annoying) errors

All the above help us check that things are as they should, but it does not prevent errors from happening. To find the source or an error, it is useful to think "like R", and to follow what R does step-by-step. This may require splitting some expressions to enable intermediate checking.

Say we want to use a t test to compare the two vectors of pulse, `pulse1` ans `pulse2`, from the `pul` data. We know we can do this by:

```
t.test(pul["pulse1"], pul["pulse2"])
```

```
##
##  Welch Two Sample t-test
##
## data:  pul["pulse1"] and pul["pulse2"]
## t = -6.4341, df = 145.16, p-value = 1.693e-09
## alternative hypothesis: true difference in means is not equal to 0
```

```
## 95 percent confidence interval:
##  -27.59474 -14.62544
## sample estimates:
## mean of x mean of y
##  75.68807  96.79817
```

Although the t test works, there is some doubt that the normality assumption holds, so we also run the Wilcoxon test with:

```
wilcox.test(pul["pulse1"], pul["pulse2"])
```

```
## Error in wilcox.test.default(pul["pulse1"], pul["pulse2"]): 'x' must be numeric
```

but this gives an error, complaining that `'x' must be numeric`. We know that the data was read in correctly from the `str()` result, so we check the class of the objects involved in the function call:

```
class( pul["pulse1"] )
```

```
## [1] "data.frame"
```

Indeed, this sort of selection preserves the object class as `data.frame`, which is not numeric. If we do instead:

```
class( pul[, "pulse1"] )
```

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

we get a numeric object. So now we try this as argument in the `wilcox.test()`:

```
wilcox.test(pul[, "pulse1"], pul[, "pulse2"])
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  pul[, "pulse1"] and pul[, "pulse2"]
## W = 3740, p-value = 2.278e-06
## alternative hypothesis: true location shift is not equal to 0
```

and this works.

It is strange as we had used the same syntax as used in the `t.test()` call. If we look at the help files of these two functions, we notice that the `t.test()` accepts data.frames as arguments, whilst the `wilcox.test()` does not.

Dangerous errors

These are errors that do not yield any error message. Such errors can remain silently in your code for ages (or many lines) before they are discovered.

We return to the example where we wish to compare the heights of students in Amsterdam and of students in Rio. We decide to not use the syntax with square brackets, due to the confusion between selecting columns that are numeric and selecting objects that are data.frames. Instead, we will select the columns using the $ sign, which is commonly done with data.frames. So we use:

```
wilcox.test(pul$pulse1, pul$puse2)
```

```
##
##  Wilcoxon signed rank test with continuity correction
##
## data:  pul$pulse1
## V = 5995, p-value < 2.2e-16
## alternative hypothesis: true location is not equal to 0
```

This seems OK, although the output is slightly different from the one we had above. The syntax used is very clear, though, so there should not be any differences. Just to check the syntax, we now apply the same test pretending it involves paired data (this is possible since the two vectors of heights have the same length).

```
wilcox.test(pul$pulse1, pul$puse2, paired=TRUE)
```

```
## Error in wilcox.test.default(pul$pulse1, pul$puse2, paired = TRUE): 'y' is missing :
```

Now we get an error message, suggesting that 'y' is missing and mentioning the paired test, but in fact we have the data to use stated clearly. So we decide to check this once more by using another syntax, now selecting the columns according to their number:

```
wilcox.test(pul[, 2], pul[, 3])
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  pul[, 2] and pul[, 3]
## W = 12052, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
```

This now yields the same output as when we used `wilcox.test(pul[, "pulse1"], pul[, "pulse2"])`, but not as `wilcox.test(pul$pulse1, pul$puse2)`. What could be going wrong here?

If we check carefully, the variable (column) names in `pul` are:

```
colnames(pul)
```

```
##  [1] "name"     "height"   "weight"   "age"      "gender"   "smokes"
##  [7] "alcohol"  "exercise" "ran"      "pulse1"   "pulse2"   "year"
```

So, what does the statement `pulse$puse2` represent? Check:

```
pulse$puse2
```

```
## NULL
```

It turns out that `y=NULL` is a valid input for the `wilcox.test` function. However, it was not what we wanted to do.

How to avoid (dangerous) errors

We all make mistakes, and this does not stop when we are programming/scripting. In fact, this happens more often than we wish when programming, as we are required to write steps precisely but we may overlook internal working differences in functions.

The difference between a beginner and an experienced R user is that the beginner will take longer to sort the error out (typically - I am willing to be proven wrong!). This is partly because a beginner goes through phases like panic (due to the red error message appearing on the screen), followed by frustration (why did this script did not just run? why is this happening to me?), and finally by closing R down and opening Excel instead.

A more experienced R user will, when encountering an error or an unexpected result, first read the error message. This may sometimes give clues as to what is going wrong - although, agreed, not always. Then it is important to try to go back a few steps on the script and check if the objects created were as expected: check `class()` and object size (via `dim()`, `length()` and `str()` are here useful). If these fail to uncover anything unexpected, check the values contained in objects. Here it is useful to perform checks in small parts of the data, when handling large objects. With data.frames, this can be done via `head()` and `tail()`. Also check for the existence of NAs creeping into the objects, via the use of `sum()` onto a `is.na(object)`, possibly in combination with apply to yield row or column summaries. Plots are of course also very useful - both scatterplots as well as heatmaps can greatly help uncover strange values/patterns in the data. Retracing steps should continue until objects are found to contain the expected values/have the expected format.

If in the few steps preceding the strange results nothing unexpected is found, then one way to continue searching for the problem source is by "peeling" of the code: this involves splitting the code into ever simpler operations, or re-writing them as we did in the wilcox.test example, and checking if these actions change the output.

What is important for any beginner in R to understand is that error messages are there to help us - by reading them carefully and performing checks we improve our understanding of how specific R functions work, and how they do not work!

Useful links

- R for Data Science
- RStudio Cheat Sheets
- LUMC Git course
- A curated list of R tutorials for Data Science, NLP and Machine Learning
- Great R packages for data import, wrangling and visualization
- ggplot2: great R package for beautiful plots - cheat sheet