Computing in R

Perry Moerland; p.d.moerland@amsterdamumc.nl,
and
Ronald Geskus; r.b.geskus@amsterdamumc.nl.
Dept. of Epidemiology and Data Science,
Amsterdam UMC, Meibergdreef 9, Amsterdam, the Netherlands

.1

Abstract. R is more than a statistical program; it is a programming language for statistical computing. Due to its flexibility and the availability of a large variety of user contributed functionality, it is quickly gaining popularity for performing analyses that go beyond the standard ones. However, for a beginner mastering R can be rather difficult. There is hardly any graphical user interface and the structure of R is quite different from other statistical programs. This course helps you to become familiar with the basics of R. We explain the basic structure and the most important functionalities. There is plenty of time for practicing R with simple computer exercises. And since it is impossible to learn more than only a small fraction during this course, we explain where extra information can be found that may answer specific questions. After the course you will be able to write short programs in R for basic data analyses and for creating publication-quality figures.

Contents

.2

1 Introduction

Course setup

- Course aim: become familiar with the basics of R
- Four days, one morning session per day: 9:00-12:00
- Mix of interactive lectures and computer exercises
- Course website: `https://bioinformatics.amc.nl/education/gs-computing-in-r/`
- Comments and suggestions for improvement are most welcome

.3

Before we start with explaining the basic structure of R, we briefly describe the general use and characteristics of statistical programs. We divide the stages in a statistical analysis into five steps. We ignore the earlier steps, that is, the design of a study and the collection of data, which are generally done without the use of a statistical program. So we suppose that we already have a data set that we want to analyze.

Stages in statistical analysis

1. Importing data into statistical program
2. Inspection of data
    - finding errors, cleaning
    - recoding and transforming
    - description and summarizing of the data

   using spreadsheets, tables and graphics
3. Analysis: estimation, uncertainty (confidence intervals, p-value), predictive value
4. Model validation
   Check the assumptions of the model

5. Reporting of results
   summary, tables, graphics
   export

.4

In this course, we concentrate on steps 2. and 5. Steps 3. and 4. are only touched upon. They are explained in more detail in Graduate School courses like "Advanced Topics in Biostatistics" and "Bioinformatics".

To some extent, all major statistical programs share the following characteristics.

Characteristics of a statistical program

1. Two ways to perform the task
   - Via the menu, graphical user interface (GUI)
   - Writing code in a script (syntax) window

   Actions performed via the menu can also be saved in a script
2. At least five windows
   - Script (syntax). A good editor is really helpful
   - Results (output). Often in *structured markup language* (html, Word, ODF (open document format), LaTeX)
   - Graphics. Can be saved in various formats (pdf, wmf, png)
     Sometimes combined with results window (SPSS)
   - Spreadsheet. To see the complete data set.
   - Help. In program or via web browser.

.5

Statistical programs differ a lot with respect to the amount of actions that can be performed via the GUI. On one extreme is SPSS, which is generally considered a user-friendly program since most types of analyses can be performed via the GUI. On the other extreme is R, which has a very rudimentary GUI. As a consequence, it not easy to master in the beginning. However, everyone who is doing statistical analyses on a regular basis, will experience that use and reuse of code is much more flexible and efficient. In general, the actions that can be performed via the menu are more restricted than what can be done by writing code. And using code makes your research reproducible (see page 31). With respect to performing tasks via written code, R is unsurpassed in extensiveness and flexibility.

However, writing code in R differs considerably from other statistical programs. Yet, there are good reasons why it has its specific structure, and in the end R turns out to be an extremely efficient program for performing statistical analyses.

R is called "a language and environment for statistical computing and graphics". If you are interested to know why, you can read `http://www.r-project.org/about.html`.

R: What is it?

- On `http://www.r-project.org/about.html`: "a language and environment for statistical computing and graphics"

- Free statistical package: no money and open source
- Runs on all major operating systems
- Standard distribution with basic statistical procedures
- Extensions via *packages*
  - Recommended; come installed together with R
  - Thousands more; can be installed from the R website
- Hard to learn(?)
- Very powerful language; exponential growth in popularity

.6

Characteristics of a statistical program: R

1. Two ways to perform the task
   - Via the menu (GUI)
     - Standard R: very few options
     - GUI: Rcmdr, Deducer and others (see links at the end of the handouts).
   - Via scripts. Saved in file with ".R" extension
2. Windows in R
   - Standard R: opens with "Console"
     Can be used for simple calculations; input and output in same window
     Script window can be opened; results still in Console
   - `RStudio`: Many windows (Console, Environment, History, ...)
     Can be customized

.7

R is already installed on the computers in the lecture room. Start the program, via the **Starten - Alle programma's - R - R x64 3.6.1** menu[1]. One window inside the main R window is opened, which is called the **R Console** window. In this course, we describe the basic structure of R and we hope to show you some examples of its efficiency. In the appendices, we give instructions on how to install the program and **RStudio**. We highly recommend to follow the suggestions in the appendix and to have a look at several of the subpages of the R web site.

2 Basics

R as a pocket calculator

- First of all, R can be used as a pocket calculator
- Many mathematical operations are pre-defined in R

```
> 2+7
[1] 9
```

[1]You can also use the 32-bit version **R i386 3.6.1**, which is the only option in 32-bit versions of MS Windows

```
 > sqrt(2)
[1] 1.414214

 > cos(pi)
[1] -1

 > log10(10^3)
[1] 3
```

.8

A simple R session

- Now we are ready to type some R code

  ```
  > x <- 2
  > x
  [1] 2
  ```

- The left arrow `<-` denotes an *assignment* statement. This stores a value in object `x`, that can then be used later on.
- Remember: without assignment, it's lost

  ```
  > x^2
  [1] 4
  > x
  [1] 2
  ```

.9

Interacting with the R Console

- Use up/down keys to go back/forth on the command history.

  ```
  y < - x
  ```

  Can easily be corrected using the up key:

  ```
  y <- x
  ```

- Use CTRL+A or HOME to go to the start of a line
- Use CTRL+E or END to go to the end of a line
- Use TAB to complete pre-defined words and filenames
- If for some reason R gets stuck try ESC (Windows) or CTRL+C (Mac, Linux)

.10

Help (I)

If you want to know more about an operator or function just use `help` (or `?`)

```
> help(sqrt)

MathFun package:base
Description:
sqrt(x) computes the (principal) square root of x.
Usage:
sqrt(x)
```

.11

Functions

- `help` is another example of a *function*
- The basic R distribution consists of a large collection of functions
- Functions generate some output given some input
- The inputs are specified via arguments of the function between parentheses `( )`:
  `name_of_function(argument_1)`
- `help(sqrt)`: `sqrt` is argument of function `help`
- The output of a function can be a value written to the Console or assigned to an object, a figure, a help page, ...

.12

Packages

- Functions in R are in general part of a package, such as the **base** package for `sqrt`
- Only the standard packages are loaded when you start R: **base**, **graphics**, **stats**, **utils** . . .
- Other packages are loaded by the `library` command
- `library()` shows the packages installed on your computer
- `help(package=stats)` gives help on all functions defined in **stats**
- Running `help.start()` launches a web browser that allows all (installed) help pages to be browsed with hyperlinks

.13

Help (II)

```
> help(mean)

Description: Generic function for the (trimmed) arithmetic mean.

Usage: mean(x, ...)
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments
x: An R object. Currently there are methods for numeric/logical vec-
tors and date, date-time and time interval objects.
Complex vectors are allowed for trim = 0, only ...

Value
If trim is zero (the default), the arithmetic mean of the values in x is com-
puted, as a numeric or complex vector ...
```

.14

Help (III)

Outline of a help page is always the same:

- Description: what does the function do
- Usage: what arguments does the function expect
- Arguments: description of the individual arguments
- Value: what is the result of a function call

- Details, references, See Also
- Example: `example(mean)`

.15

Everything is an object in R. The most important objects in R are the data, functions to perform analyses and the output of analyses. During this course, we describe these three object types in more detail. We start with the description of the different data structures and how data sets can be read into R.

3 Syntax: data

3.1 Data structures

Vectors (I)

A *vector* is one of the basic data structures in R:

```
> x <- c(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
> x
[1] 10  9  8  7  6  5  4  3  2  1
```

These commands also give a vector of the numbers 10 to 1:

```
> x <- seq(from = 10, to = 1, by = -1)
> x <- seq(10, 1)
> x <- 10:1
```

`c` (short for concatenate) and `seq` are functions as well

.16

Vectors (II)

- Vectors can be *indexed* using square brackets `[ ]`:

```
> x[5] + x[10]
[1] 7
```

- Negative indices exclude elements from a vector:

```
> c(-5, -10)
[1] -5 -10

> x[c(-5, -10)]
[1] 10  9  8  7  5  4  3  2
```

- Indices can be used to replace an element of a vector

```
> x[4] <- 12
> x
[1] 10  9  8 12  6  5  4  3  2  1
```

.17

Vectors (III)

- Functions can be applied to vectors:

```
> mean(x)
[1] 6
```

- Many calculations are *vectorized*:

```
> x + 1
[1] 11 10  9 13  7  6  5  4  3  2

> 2*x
[1] 20 18 16 24 12 10  8  6  4  2
```

.18

Matrices (I)

- From one to two dimensions:

```
> help(matrix)
matrix                  package:base
...
Usage:
     matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
     dimnames = NULL)
```

- Note: arguments to a function can be supplied by name or by position

.19

Matrices (II)

- Matrices store data in a table-like structure, with rows and columns:

```
> A <- matrix(data = 1:10, nrow = 2, ncol = 5)
> A <- matrix(1:10, 2, 5)
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

- Indexing is simple (elements):

```
> A[2, 3]
[1] 6
```

- Indices can be used to replace an element of a matrix

```
A[2,3] <- 12
```

.20

Matrices (III)

- Selecting entire row(s)

```
> A[1, ] # Same as A[1,1:5]
[1] 1 3 5 7 9
```

- Selecting entire column(s)

```
> A[, c(1, 5)] # Same as A[1:2, c(1,5)]
      [,1] [,2]
[1,]     1    9
[2,]     2   10
```

- Functions can be applied to matrices:

```
> dim(A[, c(1, 5)])
[1] 2 2
```

- The generalization to any number of dimensions is an `array`

.21

Objects (I)

- Scalars, vectors, matrices are examples of *objects*. You can get an overview of all objects you created until now via `ls` (short for list)

  ```
  > ls()
  [1] "A" "x"
  ```

- Many R functions are defined on any type of data. Examples are:

  ```
  > summary(x)
    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.00    3.25    5.50    6.00    8.75   12.00
  ```

- Try `summary(A)`

.22

Object names

- An object can have almost any name you choose: `patients`, `Data`, `abc`, `sorted.results_file`
- No space
- No special characters such as `@`,$,+ etc.
- _ and . are allowed
- Numbers allowed but not as first character
- Avoid names that are functions in R: `sort`, `c`, `mean`, `t`, `data`, `q`
- Some names are not allowed (reserved for programming constructs): `for`, `if`, `while`...
- Names are case-sensitive: `Data` is not the same as `data`

.23

Modes

- R has several atomic *modes*, the most important ones are:
  - *numeric*:

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

  - *logical*: Boolean values: `TRUE`, `FALSE`

    ```
    > -2 < 2
    [1]  TRUE
    ```

  - *character*:

    ```
    > letters[1:3]
    [1] "a" "b" "c"
    ```

  - You can change the mode of an object

    ```
     > as.character(x)
    [1] "10" "9"  "8"  "12" "6"  "5"  "4"  "3"  "2"
    [10] "1"
    ```

- Modes can be mixed in *lists*, we'll come back to that later

.24

Modes: logical (I)

- Booleans (`TRUE`, `FALSE`) can also be used as an index:

```
> x
[1] 10  9  8 12  6  5  4  3  2  1

> x[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE,
    FALSE, TRUE, FALSE)]
[1] 10  8  6  4  2
```

- Making Booleans by comparing numbers:
  Less/greater: <, >, <=, >=
  Exact equality: ==
  Not equal to: !=

```
> x[x>5]
[1] 10  9  8 12  6
```

- Booleans are converted to integers if a numeric value is required:
  `TRUE` equals 1, `FALSE` equals 0

.25

Modes: logical (II)

You can calculate with Booleans. Main operators are:

- `&`: AND - all must be true
- `|`: OR - at least one must be true
- `!`: NOT - negation

```
> TRUE & FALSE
 [1] FALSE
> TRUE | FALSE
 [1] TRUE
> x>5 & x<8
[1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[9]  FALSE FALSE
```

.26

Naming (I)

- A useful concept in R is access by *names*:

```
 > m <- c(1,2,3,4)
 > names(m) <- c("gene 1","gene 2","gene 3","gene 4")
 > m
gene 1 gene 2 gene 3 gene 4
     1      2      3      4
```

- We can also give names to rows and columns of matrix `A`:

```
> rownames(A) <- c("gene 1", "gene 2")
> colnames(A) <- c("array 1", "array 2", "array 3",
     "array 4", "array 5")
```

.27

Naming (II)

- We can now index by name instead of by number or Boolean:

```
> A
       array 1 array 2 array 3 array 4 array 5
gene 1       1       3       5       7       9
gene 2       2       4      12       8      10

> A["gene 1", ]
array 1 array 2 array 3 array 4 array 5
      1       3       5       7       9
```

- Indexing by name rather than by number makes code more readable: `Data["BRCA1",]` instead of `Data[4137,]`

.28

RStudio

- Open **RStudio** via Starten - Alle programma's - R - RStudio
- A so-called integrated development environment (IDE)
- Editor, Console, Environment, History, Plots, etc in one environment
- Download the Markdown file `CourseMain.Rmd` from `https://bioinformatics.amc.nl/education/gs-computing-in-r/` to execute the R code used during the lecture

.29

Lists (I)

- Something is needed for mixing different modes, for example character and numeric:

```
> c("gene 1", 5)
[1] "gene 1" "5"
```

- This can be done by *lists*:

```
> list(gene = "gene 1", number = 5)
$gene
[1] "gene 1"

$number
[1] 5
```

- `gene` and `number` are called *components*

.30

Lists (II)

- Lists can be indexed in various ways:
  - As vectors, with square brackets. This returns a list:

```
> x <-  list(gene = "gene 1", number = 5)
> x[1]
$gene
[1] "gene 1"
```

  - With double square brackets. This extracts a component:

```
> x[[1]]
[1] "gene 1"
```

  - Or equivalently, by name using the $ operator (if the list is named):

```
> x$gene
[1] "gene 1"
```

.31

Data frames (I)

- A special kind of list is a matrix with mixed modes, *e.g.,* rows correspond to individuals and columns to variables of different modes.
- All elements within a column should be of the same mode
- In R, this is dealt with by a `data.frame`
- External data (of the tab-delimited type, for example) imported via `read.table` is of class `data.frame`:

```
read.table               package:base
Description:
     Reads a file in table format and creates a data frame
     from it, with cases corresponding to lines and
     variables to fields in the file.
```

.32

Constructing a data frame

```
> pclass <- c("1st","2nd","1st")
> survived <- c(1,1,0)
> name <- c("Elisabeth Walton","Hudson Trevor","Helen Loraine")
> age <- c(29.0,0.9167,2.0)
> titanic <- data.frame(pclass,survived,name,age)
> titanic
  pclass survived             name     age
1    1st        1 Elisabeth Walton 29.0000
2    2nd        1    Hudson Trevor  0.9167
3    1st        0    Helen Loraine  2.0000
```

.33

Data frames (II)

- Data frames can be indexed like a matrix

```
> titanic[c(2,3),c("name","age")]
           name    age
2 Hudson Trevor 0.9167
3 Helen Loraine 2.0000
```

- Columns of a data frame can be indexed like a list, with $ and `[[ ]]`

```
titanic$age  # titanic[["age"]] gives the same result
[1] 29.0000  0.9167  2.0000
```

- $ and `[[ ]]` do not work for rows, use `subset` instead (see later)

.34

Data frames (III)

For large data frames, several useful functions exist to get a more compact overview

- `dim` gives the number of rows and columns
- `head` shows the first six rows of a data frame

```
> dim(titanic3)
[1] 1309   17
> head(titanic3[,1:4])
  pclass survived                            name    sex
1    1st        1    Allen, Miss. Elisabeth Walton female
2    1st        1  Allison, Master. Hudson Trevor   male
3    1st        0    Allison, Miss. Helen Loraine female
4    1st        0 Allison, Mr. Hudson Joshua Crei   male
5    1st        0 Allison, Mrs. Hudson J C (Bessi female
6    1st        1             Anderson, Mr. Harry   male
```

.35

Data frames (IV)

- `tail`: similar to `head` but shows the last 6 rows
- `str`: compact display of the internal structure of an R object

```
> str(titanic3[,1:4])
'data.frame':   1309 obs. of  4 variables:
 $ pclass  : Factor w/ 3 levels "1st","2nd","3rd": 1 1 1 1 1 1 1 1 1 1 ...
 $ survived: num  1 1 0 0 0 1 1 0 1 0 ...
 $ name    : chr  "Allen, Miss. Elisabeth Walton" "Allison, Master. Hudson Trevor" "Al-
lison, Miss. Helen Loraine" "Allison, Mr. Hudson Joshua Crei" ...
 $ sex     : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 1 2 ...
```

- `summary`
- `View`: opens a spreadsheet-style data viewer. In RStudio click on the name of an object in the Environment tab.
- `fix`: opens a spreadsheet-style data editor

.36

Recapitulation: objects

You have seen the most important data objects in R:

- *vectors*
- *matrices* are a two-dimensional extension of vectors
- *lists* are a general form of vectors in which the various elements need not be of the same mode
- *data frames* are matrix-like structures, in which the columns can be of different modes
- Indexing of these objects can be done by number, by name, and using Booleans.

.37

The return of the help file

```
> ?mean

Description: Generic function for the (trimmed) arithmetic mean.

Usage: mean(x, ...)
## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments
x: An R object. Currently there are methods for numeric/logical vec-
tors and date, date-time and time interval objects. Complex vectors are  al-
lowed for trim = 0, only.

Value
If trim is zero (the default), the arithmetic mean of the values in x is com-
puted, as a numeric or complex vector ...
```

.38

3.2 Data import and export, external formats

The first thing to do is to import a data set into the statistical program. In plain R and in RStudio, this is all done via commands.

Data import and export: text format

- Data frames in ASCII text format (of the tab-delimited type, for example) can be imported via `read.table`:
- Many arguments (see `help(read.table)`)

  ```
  read.table(file, header = FALSE, sep = "", quote = "\"'",
        dec = ".",row.names, col.names, as.is = !stringsAsFactors,
        na.strings = "NA", colClasses = NA, nrows = -1,
        skip = 0, check.names = TRUE, fill = !blank.lines.skip,
        ...)
  ```

- `read.csv` and `read.delim` are identical to `read.table` apart from other defaults: they are intended for comma-separated and tab-delimited files, respectively.
- Export to ASCII file: `write.table`

.39

Data import of ASCII text format: common problems

- Common problems when reading in tabular data are (especially when you use “Save as - tab-delimited file" from Excel):
  - Additional tabs: between columns or at the end of a row
  - Extra carriage returns at the end of the file
  - Unusual characters such as the # symbol (see option `comment.char`) and " quotes (see option `quote`)
  - Presence of blank fields
  - Regional settings problems: decimal separator
  - Invisible spaces
- Use `dim`, `head` etc to compare the imported data with the original data file
- Be careful when using Excel as an intermediate in manipulating files

.40

We use a data set that gives the survival status of passengers on the Titanic.

Basic data import/export from other formats

- Data formats: sav (SPSS), xls, xlsx (Excel), mdb (Access), dta (STATA), txt, csv
- sav, xls, dta, txt, csv: Imported via a function “`read.`”. E.g. a STATA file `titanic3.dta` can be imported via the commands

```
> library(foreign)
> titanic3 <- read.dta("Exercises/titanic3.dta")
```

- xlsx files: packages **openxlsx** and **readxl** (also xls files)
- SPSS, Stata, and SAS files: package **haven**
- In the latest version of RStudio via the menu **Import Dataset**. See `https://support.rstudio.com/hc/en-us/articles/218611977-Importing-Data-with-RStudio`
- Export to other formats via a function “`write.`” : `write.dta`, `write.foreign`
- See R Data Import/Export Manual under **Help or Help - R Help (RStudio)**
- See `http://r4stats.com/examples/data-import/`

.41

As an example, we show three more ways to import the Titanic data set. The first one is directly from a web site. Although the extension suggests it is an SPSS file, it is in fact a binary R file[2]. With the `ls()` or `objects()` function, it can be seen that indeed the file has been imported.

```
> con <- url("https://biostat.app.vumc.org/wiki/pub/Main/DataSets/titanic3.sav")
> load(con)
> close(con)
> rm(con)
```

We could have done the same in one call, but then the internet connection is not formally closed and this may generate a warning message later during your analyses.

```
> load(url("https://biostat.app.vumc.org/wiki/pub/Main/DataSets/titanic3.sav"))
```

Another option is to import the data set from an Excel file, which can be downloaded from the web site `https://biostat.app.vumc.org/wiki/Main/DataSets`. If this data set has been downloaded into a subfolder "Exercises" of our working directory, we can do:

```
> library(readxl)
> titanic3 <- read_excel("Exercises/titanic3.xls")
```

4 Functions; selections; special data types

4.1 Functions

R basically consists of a large collection of functions, and everything that you do in R boils down to the evaluation of a function. A function contains the code to perform some procedure, such as "perform a Student t-test", "take a logarithm" or "make a scatterplot". Details of the procedure (what are the data, do we want to perform a paired or unpaired t-test) are specified via arguments. Functions generally consist of required arguments and optional arguments. As an example we look at the the logarithmic function, which has two arguments, `x` and `base`.

```
log(x, base = exp(1))
```

The first argument provides the number of which the logarithm needs to be taken. It always needs to be specified; it is a required argument. The second argument gives the basis of the logarithm. It is optional; as default the natural logarithm is taken. If you want to take the 10-log of 1000, you use `log(1000, base=10)`.

You can leave out the name of the arguments (here `x` and `base`) if there is no ambiguity: `log(1000, 10)` gives the same answer. If we change the order, we need to make this clear: `log(base=10, 1000)` gives the same answer. However, `log(10, 1000)` would give a different answer: the 1000-log of 10. Arguments can be abbreviated if there is no risk of ambiguity.

[2]File types used by a specific program have standard extensions. However, there is no restriction on changing the extension. Here, since it is a binary R file (explained in Section 6), a better extension would have been the default `.RData`

Functions: basic format

- All actions are performed via functions
  - "Basic" functions: `sqrt, mean, help, library`
  - Functions for analysis: `t.test`, `lm`, `plot`
- Input: *required* and *optional* arguments; within parentheses (`sqrt(2)`, `help(seq)`), separated by comma
  - required: need to be supplied
  - optional: have default values

  Beware of sequence of arguments; required ones come first
  e.g. `log(x, base = exp(1))`, `x` required, `base` optional. Argument names can be abbreviated if no risk of ambiguity
- Special "argument" `...`: anything that makes sense, e.g. in `c` and `paste` function
- Output: result of calculations (typically assigned to R object), graphics, help window, . . .
- You can use functions within other functions, e.g. `mean(c(3,6,8))`

.42

A function is evaluated by giving its name, followed by the arguments within parentheses. The code of the function can be obtained by only giving the name of the function. Most of the functions consist themselves of a collection of other functions, grouped together via curly brackets `{ }`. Not only functions can be grouped using curly brackets. Any collection of lines that is placed within curly brackets is seen as a *compound expression*.

Functions: the inside

- Function code can be seen by leaving out the parentheses `( )`
- General structure: `function(args) SOME R CODE`
  with `SOME R CODE` a collection of other functions as *compound expression*
- Compound expressions are placed within "{ " and " }":

```
> z <- {
     x <- 2
     y <- x + 2
 }
> z
[1] 4
```

- A compound expression returns the last value

.43

One function that may be useful when writing a paper that used R for the analyses is the function `citation()`, which gives information on how to cite R in publications.

Functions and packages

- You can write your own functions:

```
> good.morning <- function(work){
    if(work==TRUE) cat("wake up") else
      cat("you can stay in bed")
}
```

Note: here the function is saved in the object `good.morning`
- Can make it into a *package*, i.e. a collection of functions (and data):
  survival, ggplot2, Rcmdr
  sudoku, scuba, engsoccerdata
  See `http://cran.r-project.org/web/packages/`
- R Reference Card 2.0 for overview of most important functions

.44

A summary of the most frequently used functions can be found in the reference card `http://cran.r-project.org/doc/contrib/Baggott-refcard-v2.pdf`. A full reference manual with all R functions in the basic R distribution can be obtained from `http://cran.r-project.org/doc/manuals/fullrefman.pdf`.

4.2 Selections

Most statistical analyses involve making selections, by row and/or by column. We select columns if we choose variables that are to be analysed or summarized. We select rows if we restrict analyses to subgroups, e.g. performing an analysis only for males.

Selection of rows and columns

- Index: `[ ]` (vector) or `[row, col]` (data frame)
  - By *character*: `titanic3[,"sex"]`, `titanic3[,c("age","sex")]`, `islands["Moluccas"]`
  - By *number*: `islands[29]`, `islands[-29]`
  - By *logical*: `islands[names(islands) != "Moluccas"]`
- Columns in data frame can also be selected via $, e.g. `titanic3$sex`
- We can *assign* values to selections or new columns

```
> titanic3[3,"age"] <- 23.4
> my.data$bmi <- my.data$weight/(my.data$height)^2
```

.45

The functions `subset` and `with` can be used for selecting rows and columns from a data frame. Although not strictly necessary, using them often saves some typing and makes the code more readable. As an alternative, many functions allow for row selection via the *subset* argument. Column selection can be done within a function if it uses the *formula* structure. Such functions have a *data* argument that specifies the data frame to be used. Then, for every column that is chosen, only the variable name itself needs to be specified. This is explained in section 9.

Selection of rows via functions

- Via special functions: `head`, `tail`, `subset`
  `subset(my.data, ...)` with ... a logical condition

```
> subset(titanic3, pclass %in% c("1st","2nd"))
```

  (`%in%`—“belongs to”—is another Boolean construct)
- Many functions have a *subset* argument
  Often combined with formula structure

```
> xtabs(~survived, data=titanic3, subset=(sex=="male"))
```

.46

Selection of columns via functions

- Via `with` function:

  ```
  > table(titanic3$sex, titanic3$survived)
  > with(titanic3, table(sex, survived))
  ```

- Many functions have a *data* argument, combined with formula structure

  ```
  > xtabs(~sex+survived, data=titanic3)
  ```

- Via *select* argument of `subset` function
- Don't use "$" for column selection if function has a data argument
  Don't write:

  ```
  > xtabs(~titanic3$sex+titanic3$survived, data=titanic3)
  ```

.47

We explain the basic selection functions with the `juul` data set, which is included in the **ISwR** package that comes with the book "Introductory Statistics with R" by P. Dalgaard. First, we download and install the package. Instead of the using the GUI, we can also use the function `install.packages`. Next, the package is loaded via `library(ISwR)` and we have a look at the description of the `juul` data set via `help(juul)`.

```
> install.packages("ISwR")
> library(ISwR)
> help(juul)
> subset(juul, tanner==2 & age<10)
     age menarche sex igf1 tanner testvol
192 9.50       NA   1   NA      2       2
831 9.82        1   2   NA      2      NA
835 9.89        1   2  229      2      NA
> subset(juul, tanner>=4 & age > 45)
       age menarche sex igf1 tanner testvol
1325 47.37        2   2  144      5      NA
1326 48.01        2   2  154      5      NA
1329 51.07        2   2  187      5      NA
1334 58.95        2   2  218      5      NA
1335 60.99        2   2  226      5      NA
> subset(juul, tanner %in% c(1,5) & (age==0.25 | age>50))
       age menarche sex igf1 tanner testvol
13    0.25       NA   1   90      1      NA
14    0.25       NA   1  141      1      NA
628   0.25       NA   2   51      1      NA
1329 51.07        2   2  187      5      NA
1334 58.95        2   2  218      5      NA
1335 60.99        2   2  226      5      NA
> xtabs(~sex+tanner, data=juul, subset=(menarche==1))
   tanner
sex   1   2   3   4   5
  2 221  43  32  14   2
```

4.3 Some special data types

4.3.1 Missing data

Missing data

- Special value: NA (short for "not available")
- The function `is.na` checks for missingness

```
> table(is.na(titanic3$age))
FALSE  TRUE
 1046   263
```

- Within functions, missings are often excluded by default, *but not always*
  - `quantile`, `mean` give error if there are missings; specify argument `na.rm=TRUE`
  - `table` excludes missings, include them via argument `useNA="always"`

.48

We give an example using the data set `juul`.

```
> with(juul, table(sex, menarche))
     1   2
 1   0   0
 2 369 335
> with(juul, table(sex, menarche, useNA="always"))
        1   2 <NA>
 1      0   0  621
 2    369 335    9
 <NA>   0   0    5
```

4.3.2 Factors

In R, there are two ways to represent a categorical variable. We can store it as a vector of character values. But if we want to use it in statistical analyses, it is better to store it as a **factor**. Factors are interpreted as categorical variables of mode "character". However, they have a different format, allowing for more flexibility. Internally, values are stored as numbers $1, 2, \ldots$, with a link between each number and the categorical value it represents. In other statistical programs, the same distinction exists[3]. Often, factors and categorical variables are treated similarly in R.

Factors are useful if we want to define the reference category when fitting statistical models. Mostly, effects of categorical variables are quantified with respect to some reference value of the variable. The default choice (e.g. "first" or "last" category as reference) differs per statistical program, but can be changed within the program as well. In R, the default choice is to choose the "first" category, i.e. the one coded as the number 1, as the reference value. By default, when creating factors or reading a data set from another format, the "first" category is the one that comes first in alphabetical order. However, R allows for manipulation of the choice of reference value within factors. If your data preparation

[3]In SPSS, variables of mode character are of **Type** "String", whereas factors are of **Type** "Numeric". The latter have a label defined for the different values in the **Value** column.

has been done in another program, you are advised to check whether the correct labels are allocated to each variable, and whether the coding of missing values is still correct.

Note that the `summary` function for a `data.frame` does not summarize character vectors.

Factors: what are they?

- Categorical variable with "levels"

```
> DiseaseState <- factor(c("Cancer", "Cancer", "Normal"))
> DiseaseState
[1] Cancer Cancer Normal
Levels: Cancer Normal
> levels(DiseaseState)
[1] "Cancer" "Normal"
```

- Ordering: default is alphabetical/numeric
- Internally represented as integers $1, 2, \ldots$

```
> as.numeric(DiseaseState)
[1] 1 1 2
```

.49

Factors: how to create?

- By default, character columns are converted into factor if data are read from other statistical programs. Numeric codings (e.g. 999) are not converted by default.
- Create or manipulate via `factor` function
  - Required argument `x`: vector with values
  - Optional argument `levels`: vector of unique values in `x`; sequence determines ordering. Compare

    ```
    > table(factor(DiseaseState))
    > table(factor(DiseaseState, levels=c("Normal","Cancer")))
    ```

  - Optional argument `labels`: labels given to levels.
    Default: same as levels
- Useful in statistical models. Standard in R: first group is reference group. Choice of reference group changed via `relevel`:

```
> relevel(DiseaseState, "Normal")
```

.50

4.3.3 Dates

Dates are basically numeric values, but represented in character format. They are internally stored as a numeric value that gives the number of time units with respect to some time origin. For example, in SPSS dates are stored as the number of seconds from midnight, October 14, 1582 (the beginning of the Gregorian calendar). In R, dates are stored as the number of days since January 1st, 1970. R is flexible with respect to the formats that are used to represent dates. The help file for `strptime` and the reference card give further information. Some further explanation and examples on the use of dates in R is given in the R News Journal: *Grothendieck G, Petzoldt T (2004). "Date and Time Classes in R." R News, 4(1), 29-32.* at `http://www.r-project.org/doc/Rnews/Rnews_2004-1.pdf`. Also note that the package **lubridate** helps in manipulating dates, see `http://www.jstatsoft.org/v40/i03/paper`.

Dates

- Numeric value (units since time origin) with character representation
- Origin: SPSS: October 14, 1582 (seconds); R: January 1st, 1970 (days); STATA: January 1st, 1960 (days)
- SPSS files read into R via `read.spss` in `foreign` package need to be converted

  ```
  > my.data$date <- as.Date(my.data$date+ISOdate(1582,10,14) )
  ```

  The `haven` package makes the conversion automatically
- R is very flexible in conversion between textual date representations
- `as.Date`: create date variable
  `format`: change display format

.51

```
> as.Date("15 April 1912", "%d %b %Y")
[1] "1912-04-15"
> julian(as.Date("15 April 1912", "%d %b %Y"))
[1] -21080
attr(,"origin")
[1] "1970-01-01"
> titanic3$dob <- as.Date("15 April 1912", "%d %b %Y")+ (-titanic3$age*365.25)
> as.Date("15041912", "%d%m%Y")
[1] "1912-04-15"
> as.Date("150412", "%d%m%y")
[1] "2012-04-15"
> format(as.Date("1912April15", "%Y%b%d"),"%A %B %d, %Y")
[1] "Monday April 15, 1912"
> library(lubridate)
> wday(as.Date("1912April15", "%Y%b%d"),label=TRUE)
[1] Mon
Levels: Sun < Mon < Tues < Wed < Thurs < Fri < Sat
```

5 Graphics

R has versatile tools for graphics. There are typically three steps to producing useful graphics:

1. Creating the basic plot
2. Enhancing the plot with labels, legends, colors etc.
3. Exporting the plot from R for use elsewhere

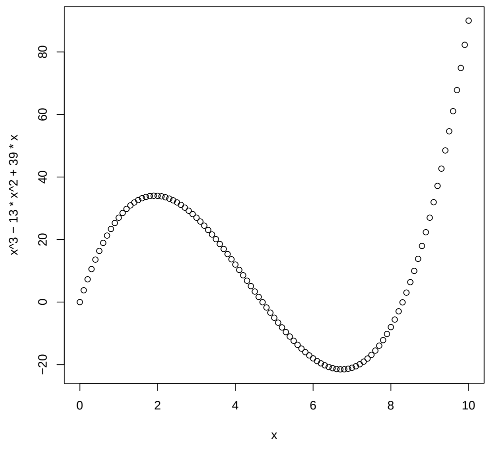
.52

5.1 Basic graphics

Basic plot (I)

It is straightforward to make a simple plot using functions from the **graphics** package (loaded by default):

```
> x <- (0:100)/10
> plot(x, x^3 - 13 * x^2 + 39 * x)
```

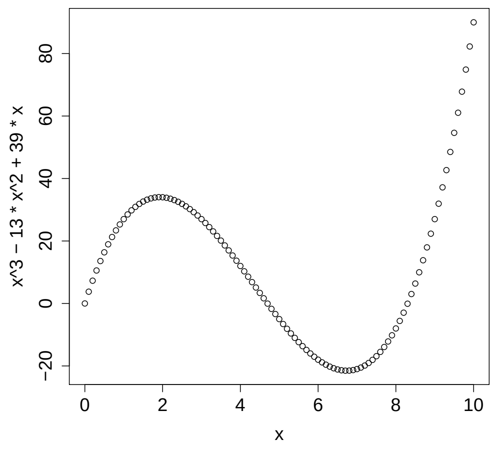


.53

Basic plot (II)

You can increase the size of the symbols on the axes and the axis labels (`cex` stands for character expansion factor):

```
> plot(x, x^3 - 13 * x^2 + 39 * x,cex.axis=1.5,cex.lab=1.5)
```

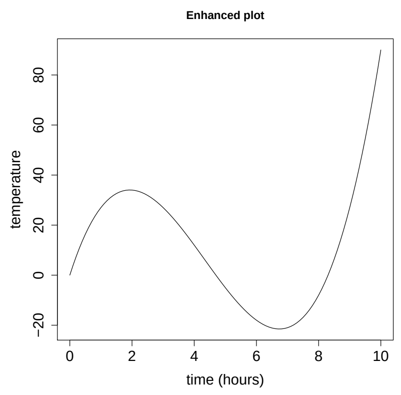


.54

Enhancing a plot (I)

- Change the type of plot via the argument `type`: `"p"` for **p**oints (is default), `"l"` for **l**ines, etc. See `?plot` for other options
- Change the titles for the axes via `xlab` and `ylab`
- Add an overall title for the plot via `main`

```
> plot(x,x^3-13*x^2+39*x,type="l",xlab="time (hours)",
  ylab="temperature",main="Enhanced plot",cex.axis=1.5,cex.lab=1.5)
```

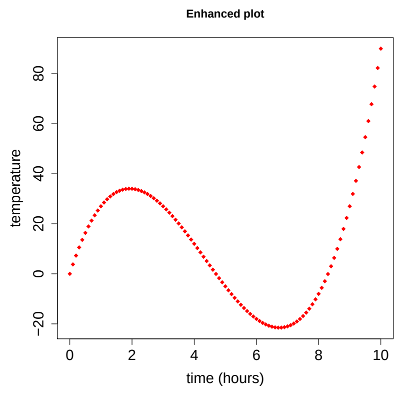


.55

Enhancing a plot (II)

- Change the plot symbol used from the default o via the argument pch
- Change the colour via the argument col. By name: see colors() for the 657 options. By number: see palette()

```
> plot(x,x^3-13*x^2+39*x,pch=18,xlab="time (hours)",
  ylab="temperature",col="red",main="Enhanced plot",
  cex.axis=1.5,cex.lab=1.5)
```

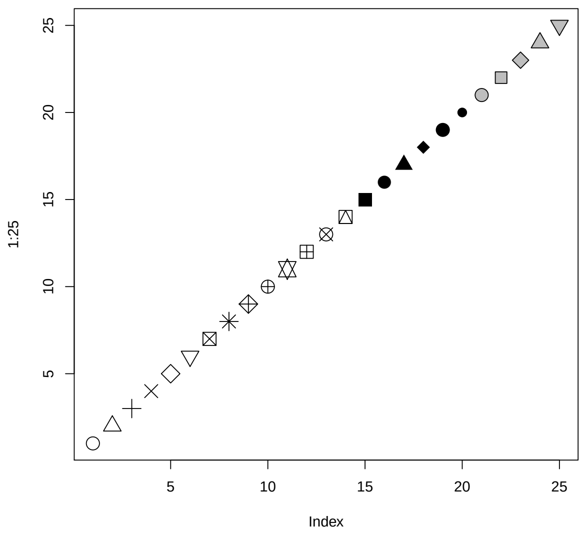


.56

Plot symbols

There are 25 different plot symbols, see ?points

```
> plot(1:25, pch=1:25,cex=2,bg="grey")
# bg: background colors for open plot symbols
```

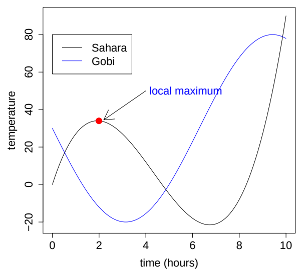


.57

Enhancing a plot (III)

You can add points, arrows, text, lines, and a legend to an existing plot:

```
> x<-(0:100)/10
> plot(x,x^3-13*x^2+39*x,type="l",xlab=
  "time (hours)",ylab="temperature",cex.axis=1.5,cex.lab=1.5)
> points(2,34,col="red",pch=16,cex=2)
> arrows(4,50,2.2,34.5)
> text(4.15,50,"local maximum",adj=0,col="blue",cex=1.5)
> lines(x,30-50*sin(x/2),col="blue")
> legend(x=0,y=80,legend=c("Sahara","Gobi"),col=c("black","blue"),
  cex=1.5)
```



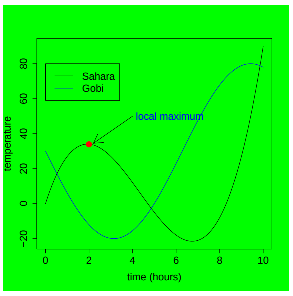
.58

Graphical parameters (I)

You can change the default value of many graphical parameters via `par` (see `?par`). For example to reset the background of a plot to green:

```
> par(bg="green")
```

and then rerun the plot commands



You can set a parameter back to its default value (white) by `par(bg="white")`

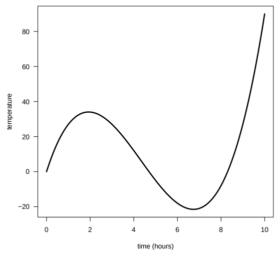
.59

Graphical parameters (II)

Other often used options:

- `lwd` sets the line width
- `mfrow` and `mfcol` enable multiple plots in one figure
- `las` to rotate axis symbols
- `mar` to change the default margins of the figure

```
> x<-(0:100)/10
> plot(x,x^3-13*x^2+39*x,type="l",
  xlab= "time (hours)",ylab="temperature",
  lwd=3,las=1,cex.axis=1.5,cex.lab=1.5)
```

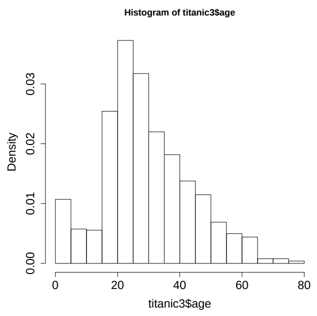


.60

Histograms

Use `hist` for plotting histograms. As always, see `?hist` for the many arguments of this function

```
> hist(titanic3$age,breaks=15,freq=FALSE,
  cex.axis=1.5,cex.lab=1.5)
```

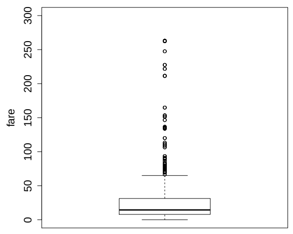


.61

Boxplot

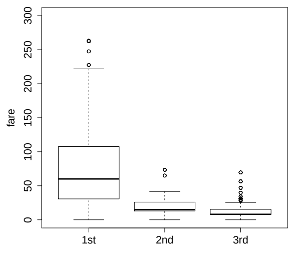
The function `boxplot` can be used on a vector

```
> boxplot(titanic3$fare,
  ylim=c(0,300),ylab="fare",
  cex.axis=1.5,cex.lab=1.5)
```



`boxplot` also has a formula interface

```
> boxplot(fare ~ pclass,
  data=titanic3,ylim=c(0,300),ylab="fare",
  cex.axis=1.5,cex.lab=1.5)
```



.62

5.2 Other types of graphics

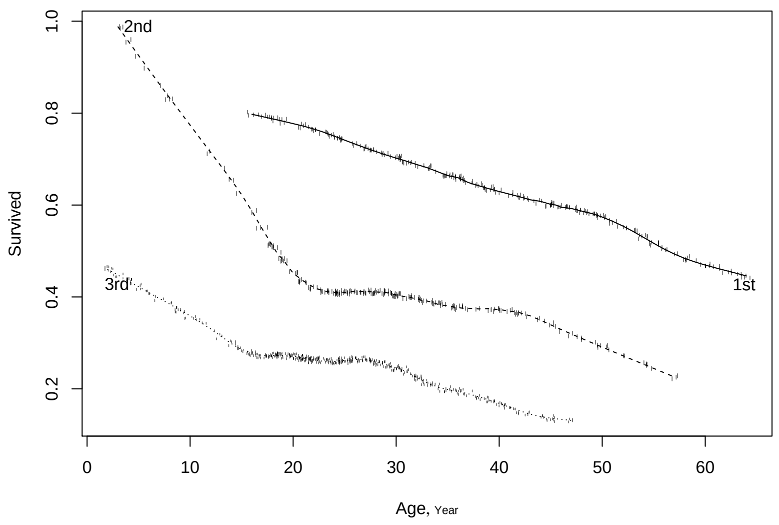
Advanced R graphics

- Ch 12 of "An Introduction to R" gives an introduction to base graphics
- **lattice**: very powerful for multipanel conditioning
  needs to be loaded first; `xyplot` is the main function
- **ggplot2**: based on "the grammar of graphics"
- **rCharts**, **ggvis**, **Shiny**: interactive visualizations
- and in many more packages (**gplots**, **plotrix**, . . .)

.63

We give an example of what can be done with functions and packages with respect to plotting. In the **Hmisc** package, which has many interesting functions, we can find a function that calculates and plots trends by subgroup. We save the file in "pdf" format (you need to create a directory called "Graphs" first).

```
> library(Hmisc)
> pdf("Graphs/TitanicByPclass.pdf",width=8,height=6)
> with(titanic3, plsmo(age, survived, group=pclass, datadensity=TRUE) )
> dev.off()
```



Export: two types of formats

- Vector format (pdf, eps, wmf, emf)
  - digital image consisting of independent geometric objects (segments, polygons, curves, etc.)
  - can be enlarged without losing resolution
- Raster (png, jpeg, tiff).
  - rectangular grid of pixels, possibly with color
  - Resolution impaired if image is enlarged
- Graphics can be saved via the menu in the graphics/plots window, or a specific graphics file type can be created directly (`pdf(...)`, `win.metafile(...)`, `png(...)` and ending with `dev.off()`)

.64

6 Internal and external communication

With “internal communication” we refer to how R is organised. Whe have seen that we can create objects (data as well as functions) in R. But how are these objects stored internally?

With “external communication” we refer to the exchange of information between R and the operating system. We already explained how data sets can be imported into R. We also explained how to save the plots we made. But how do we export the results of our analyses, or the data sets that we have created or changed in our R session? How do we save them in an R format or some other format?

6.1 The structure of R

The internal organisation of R is not very different from the organisation of an operating system like Windows. Objects somewhat resemble files. They can be of different type (data.frame, function, list, ...), just like files can be (ASCII text files, MS Word files, files that are used by the operating system,...). All objects (example data sets and functions) that come included with R are stored in packages. The model fitting functions, such as `lm` for fitting a linear regression model, are in the *stats* packages. The basic functions for making graphs are in the *graphics* package. The more basic functions (`sqrt`, `table`) are in the *base* package.

Objects live in an environment during your R session. Environments are like folders in an operating system. When a package is loaded, an environment is created that contains the objects (data, functions, ...) from that package. When R is started, several packages are loaded by default. The objects that we create during our R session are in the “Workspace” environment.

Structure of R

- *Objects*: data, functions (statistical procedures), model output
- *Environment*: a collection of objects that is accessible in R session
  - Objects we create: in “Workspace” (RStudio: in Global Environment window)
  - Packages with existing functions: **base**, **stats**, **graphics**
  - When a package is loaded, a new environment is created
  - Some more environments, e.g. some tools in RStudio
- `search()` shows the environments in the search path
  `ls()` or `objects()` shows the objects in an environment
- Hierarchical structure of environments; needed for dealing with duplicate names

.65

R resembles operating system

R	OS
objects	files
Workspace	current folder
environments	folders in “path” variable
RStudio “Environment” window	Explorer window

.66

The `search` command lists the loaded packages and all standard R objects (functions, data)[4]:

```
> search()
 [1] ".GlobalEnv"        "package:RODBC"     "package:tools"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
```

We can have a look at the contents of each of these environments by changing the default value of the *name* argument in the `ls` function.

Workspace management; connection with OS

- Save complete Workspace on disk
  - R: **File** → **Save Workspace** (or the `save.image` function)
  - RStudio: Floppy disk icon in the Global Environment window
  - Asked when you close the R session (e.g. via command: `q()`)
- Save specific objects: via `save` function
- Binary format file with extension: ".RData"
- `load` can import R workspace or collection of R objects
- Delete objects from workspace within R via `rm` function

  ```
  > rm(titanic3)
  ```

  Remove all objects from workspace:

  ```
  > rm(list=ls())
  ```

.67

Project management

- Every project (analysis) in separate folder (*working directory*)
- Users can have several working directories with separate .RData files and script files
- R best started via double clicking on script file with ".R" extension or workspace file with ".RData" extension. Working directory is that same folder
- Otherwise, use commands `getwd` and `setwd` or the GUI to get and set the working directory
  Note: R uses / or \\ instead of \ in path specification
- RStudio has an elegant Project concept
  `https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects`

.68

6.2 Export to other formats

At the end of the analysis, you may want to use your results in a report, paper or presentation. The creation and export of graphs has been discussed in Section 5. For basic tables, we can use the `write.table` function to save it as an ASCII file and the `write.xlsx` function in packages **openxlsx** and **xlsx** to save it as Excel file. Other packages contain functions that allow for more flexibility when saving tables in html or LaTeX format.

[4]If you run this function yourself, the result may be different, because you may have other packages loaded.

Export tables to other formats

- Copy and paste
- Use `write.table`
- Function `write.xlsx` in package **xlsx** for Excel
- HTML output: packages **xtable**, **R2HTML** and **PrettyR**
- Many options for LaTeX users, e.g. **Hmisc, xtable**

.69

For data export, we can use the **foreign** package. It contains the function `write.dta` to save a data set in Stata format and `write.foreign` to save in SPSS format. The `haven` package also provides several options.

For the exporting of complete model fits (stored as R objects), the standard options in R are rather basic. We can copy and paste the ASCII output. Several packages have been developed that allow for html or LaTeX output (e.g. **R2HTML** and **PrettyR**).

Better, although more time-consuming in the beginning, is to make the whole process of a statistical analysis *reproducible*, starting from the data management and ending in the final paper that will be published. More and more journals require some form of reproducibility of the analyses that led to the submitted paper.

The most elegant approach is to have all text and R code in a single file with a specific format. This file is "compiled", which means that all R code is run and a file is generated that combines the text as well as the results of the analyses (often in the form of tables and figures). For Windows users, the **R2wd** and **R2PPT** packages, the **ReporteRs** package (`http://davidgohel.github.io/ReporteRs/index.html`) or the **SWord** program (`http://rcom.univie.ac.at/download.html`) can be used to create Word or Powerpoint documents.

However, we recommended to generate MS Word files using the `R Markdown` format and the **knitr** package. The file `CourseMain.Rmd` that contains all the example code that we showed during the course has been written in this format. In RStudio, it is very easy to convert this file to MS Word format (see `https://support.rstudio.com/hc/en-us/articles/205368677-R-Markdown-Dynamic-Documents-for-R`). Instead of MS Word format, you can also choose to generate a file in html or pdf format.

The `knitr` package can also be used to convert LaTeX files with R code into html or pdf files using a format that is more flexible than `R Markdown`[5]. Another option is to use the **odfWeave** to generate Open Office documents. Note that HTML or Open Office files can be imported into MS Word.

Reproducible research

- See Task View at
  `http://cran.r-project.org/web/views/ReproducibleResearch.html`
- Most elegant approach: both R *code* and explanatory *text* in same file
- Compilation: run R code, and keep the surrounding text
- Recommended: use Markdown format in Rstudio
  Compilation via **knitr** package
- `http://andrewgelman.com/2014/09/19/never-happened-r-markdown/`

.70

[5]More information on the use of LaTeX for reproducible research can be found at `https://biostat.app.vumc.org/wiki/Main/StatReport`

7 Data manipulation and inspection

Usually, collected data need some management before it is used for analysis. We may want to sort the data set by some variable. Or we may want to combine information that is spread over tow or more data sets. If we have longitudinal data, i.e. several measurements within the same individual over time, sometimes the data are in “wide” format. This means that each row contains all data from one individual. For data analysis, it is usually required to transform such data to “long” format, i.e. one row per measurement.

R provides functions for data management in the **base** package, which is loaded by default at startup. However, there are other packages that offer more flexibility and may be easier to use. We describe some options in the packages **dplyr** and **tidyr**. See `https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html` for some useful suggestions on the use of both packages. A Cheat Sheet that summarizes the functionality of both packages can be downloaded from the RStudio website.

Some functions for data management

- Sorting. Base R: `sort` and `order`. **dplyr**: `arrange`.
  Rstudio: sorting in spreadsheet window (not saved in object)
- Merging. R: `merge`. **dplyr**: `left_join` (3 other options, see Data Transformation Cheat Sheet).
- Long to wide. R: `reshape`. **tidyr**: `pivot_wider`
  Wide to long. Base R: `reshape`. **tidyr**: `pivot_longer`

.71

In R, we can use transformed values right in the model specification. There is no need to create a column with the transformed values. Still, creating a transformed variable is recommended if you are going to use it a lot in your analysis.

Creating transformed variables

- Arithmetic functions: `log` etc.
- `cut` to split continuous variable into groups
- Note: transformations not needed for model fitting
- Adding variables
  - Base R: via $
    Functions `within` and `transform` may be helpful
  - **dplyr**: `mutate` or `transmute`

.72

As an example of creating derived (transformed) variables, suppose we want to categorize the age variable, and want to add a variable that shows whether there were any relatives (sibling, spouse, parent, child) aboard. We can use the functions `cut`, which is used to categorize continuous variables, and `ifelse`, which is a vectorized function for conditional assignments.

```
> titanic3 <- within(titanic3, {
    agecat <- cut(age, breaks=c(0,18,30,40,80))
    family <- factor(ifelse(parch==0 & sibsp==0, "no", "yes"))
  })
> summary(titanic3[,c("age","agecat","family")])
       age                agecat     family
 Min.   :  0.1667   (0,18] :193   no  :790
 1st Qu.: 21.0000   (18,30]:416   yes :519
 Median : 28.0000   (30,40]:210   NA's:  1
 Mean   : 29.8811   (40,80]:227
 3rd Qu.: 39.0000   NA's   :264
 Max.   : 80.0000
 NA's   :264.0000
```

After the data have been made into the right format, a next step is often to create some summaries. We have already seen the `summary`, `table` and `xtabs` functions to create summaries and tables. Often, we want to create summaries for different subgroups.

Making basic summaries

- Data summary: `summary`
- Contingency tables: `table`, `xtabs`
  `CrossTable` in **descr** package
- Summary by subgroups
  - Base R: `aggregate`, `tapply`
  - Several functions in packages **doBy**, **Hmisc**, **compareGroups**, **dplyr**

.73

8 Documentation and help

R is very flexible and has a lot of functionality. The problem is: how do we find the thing we need? It is impossible to know all the functions in R by heart. Fortunately, there are many ways that help in finding the right information and knowing some of these options quickly pays off.

First, there are the manuals. There is a reference manual that lists the help files of the functions that comes installed with R. Help for a specific function can also be found via the `help` function. When looking for a function description via `help`, one has to know that R uses a programming style called *object orientation* (OOP: object oriented programming). Many functions have specific methods depending on the type (“class”) of the object that is given as argument. For example, in a logistic regression the effects of the covariables are usually presented as odds ratios, whereas in a linear model there is no odds ratio. Therefore, the `summary` command has a specific method for every type of model. The function that summarizes the output from a linear model is called `summary.lm`. The help file for this function is found via `help(summary.lm)`, not via `help(summary)`.

Another example where this object orientation (or automatic recognizing of the type of structure) can be seen is in the plot function.

`plot(age ~ pclass, data=titanic3)` gives a box plot, whereas `plot(age ~ fare, data=titanic3)` generates a scatter plot. The reason is that `pclass` is of *class* factor.

Finding Information

- Function `help`. R is object oriented!
- Function `help.search`
- Function `RSiteSearch`
  Opens web browser with all keyword specific info on functions from CRAN
- Package **sos**
- Manuals in R
- CRAN (Task Views, Vignettes, list with packages)
- Search in R mailing lists on `http://r-help.markmail.org`
- Search on Google using `language:r`
- `http://rseek.org`
- `http://stackoverflow.com/questions/tagged/r`
- Have a look at the links provided at the end of the handout or at `https://bioinformatics.amc.nl/education/gs-computing-in-r/`

.74

9 Model fitting; formulas

Suppose we want to perform a Student's t-test in R, comparing the age distribution of boys and girls in the Titanic data set. As a novice R user, we have no idea what to do. Hence, we perform a search in R with `help.search` on the key word "Student" ("t" as a key word would be too general). We find that there exists a function `t.test`, and we have a look at its help file. From the help file, we learn that we can specify the two groups in the "x" and "y" arguments.

```
> help.search("Student")
> help(t.test)
> with(titanic3,t.test(age[sex=="male"],age[sex=="female"]))

        Welch Two Sample t-test

data:  age[sex == "male"] and age[sex == "female"]
t = 2.0497, df = 798.36, p-value = 0.04072
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.08036699 3.71595773
sample estimates:
mean of x mean of y
 30.58523  28.68707
```

We use the `with` function in order to save some writing. An alternative, less elegant formulation is

```
> t.test(subset(titanic3,sex=="male")$age,subset(titanic3,sex=="female")$age)
```

A further look at the help file for `t.test` reveals that there is also a "formula" way to use the function. This formula structure is based on notation introduced by Wilkinson and Rogers, and is also used in some other statistical programs. Once you have learned some basic rules, the formula structure turns out to be very flexible. In R, almost all functions that fit a regression model have a formula as the first argument. The same holds for several functions for data management and making graphs.

Regression; Formulas

The regression equation is represented as a *formula*

General form `dependent` $\sim$ `independent`
Dependent Depends on type of model, check help file of modeling function
Independent Variable names separated by operators, without explicit reference to parameters
`fare` $\sim$ `age` + `pclass` + `sex`   three main effects

- interactions are denoted by ":"
  interaction and main effects by "$*$"   `age` $*$ `sex` = `age` + `sex` + `age` : `sex`
- formulas may involve existing functions:
  `log(fare), I(age+dob), sqrt(age), cut(age,breaks=3)`

.75

The output of an analysis has many different components. It should contain at least the parameter estimates and their standard errors. But it will also contain information on the data set that was used (e.g. sample size). Since it has components of different type, the output of an analysis is typically stored in a list format.

Model output

Output model stored in a list. Results observed via functions

print Short summary of model outcome; typing name is sufficient
summary Longer summary of model description
coef Parameter values
confint Confidence intervals
anova Sequential anova table or compare two models
fitted Calculates fitted values for records in model
predict Calculates predicted values for certain values of covariates
update Used to refit the model with small changes

.76

```
> fit.age <- lm(age ~ pclass, data=titanic3)
> fit.age
Call:
lm(formula = age ~ pclass, data = titanic3)

Coefficients:
(Intercept)    pclass2nd    pclass3rd
     39.160       -9.653      -14.344
> names(fit.age)
 [1] "coefficients"  "residuals"     "effects"       "rank"          "fitted.values"
 [6] "assign"        "qr"            "df.residual"   "na.action"     "contrasts"
[11] "xlevels"       "call"          "terms"         "model"
```

```
> coef(fit.age)
(Intercept)   pclass2nd   pclass3rd
  39.159918   -9.653213  -14.343551
> predict(fit.age, newdata=data.frame(pclass=factor(levels(titanic3$pclass))))
       1        2        3
39.15992 29.50670 24.81637
> update(fit.age,. ~ .+sex)

Call:
lm(formula = age ~ pclass + sex, data = titanic3)

Coefficients:
(Intercept)    pclass2nd    pclass3rd      sexmale
     37.182       -9.927      -14.957        3.720
```

Formula structure

Same formula structure in other types of analysis

- graphics

  ```
  > plot(age ~ fare, data=titanic3)
  ```

- summaries (`xtabs`)
- packages (**doBy**, **Hmisc**, **compareGroups**)
- and many many more

.77

10 Programming and ply functions

10.1 Programming constructs

Statements: if-then-else

- R also has a *conditional* construct: depending on the outcome of a test, execute one or another statement

  ```
  if (logical statement){
    do this
  } else {
    do that
  }
  ```

  ```
  > x <- 10
  > z <- if (x < 2) 4 else 3
  > z
  [1] 3
  ```

.78

Statements: repetition (I)

- Let us look at a simple example using matrix A

```
           array 1 array 2 array 3 array 4 array 5
gene 1           1       3       5       7       9
gene 2           2       4       6       8      10
```

```
> results <- numeric(2)
> results
1] 0 0
> for (i in 1:2) {
      results[i] <- mean(A[i, ])
  }
> results
[1] 5 6
```

- We iteratively calculated the mean of each row

.79

Statements: repetition (II)

Imagine that you have to repeat the same analysis for many files that are all in the same folder on your computer. A short solution using an iterative construct would be

```
> files <- dir()
> for (filename in files){
    infile <- read.table(filename, ...)
    do something with infile
  }
```

.80

10.2 The apply family

Apply

- Functions from the apply family are convenient shorthands for repetitions

```
apply(X, MARGIN, FUN, ...)
Arguments
X       an array, including a matrix
MARGIN  for a matrix 1 indicates rows, 2 indicates columns
FUN     the function to be applied
```

- Taking a row-wise mean can be handled using apply

```
> apply(A, 1, mean)
gene 1 gene 2
     5      6
```

.81

Similarly, you can calculate the columnwise mean and standard deviation using apply:

```
> x <- rnorm(10, -5, 0.1)
> y <- rnorm(10, 5, 2)
> X <- cbind(x, y) # the columns of X keep holding the names "x" and "y"
> X
             x         y
[1,] -5.146915 0.9761563
[2,] -5.225160 7.6419268
[3,] -4.978514 4.6326178
[4,] -5.083719 2.1867986
[5,] -5.083873 5.5591872
[6,] -5.002992 3.5533913
[7,] -5.075303 4.5484952
[8,] -4.908258 3.9063953
[9,] -4.973577 6.9551233
[10,] -5.105114 5.5442915
> apply(X, 2, mean)
        x         y
-5.058342  4.550438
> apply(X, 2, sd)
         x          y
0.09344694 2.03095668
```

Other members of the apply family

- `lapply`: apply a function over a list or vector
- `sapply`: similar to `lapply` but more user-friendly if output can be coerced into a vector
- `tapply`: can be used to split a vector in subgroups and apply a function to each of the subgroups
- `replicate`: simpler version of `sapply` for the repeated evaluation of an expression. Often used for random number generation
- `aggregate`: extension of `tapply` for data frames that splits the data into subgroups and computes summary statistics for each of the subgroups.

.82

tapply: example

- Let us again have a look at the *Titanic* data

```
> head(titanic3[,c("fare","pclass")])
         fare      pclass
1    211.3375         1st
2    151.5500         1st
3    151.5500         1st
4    151.5500         1st
5    151.5500         1st
6     26.5500         1st
```

- Now we can use `tapply` to calculate the mean fare per passenger class

```
> with(titanic3, tapply(fare, pclass, mean, na.rm=TRUE))
     1st      2nd      3rd
87.50899 21.17920 13.30289
```

- **dplyr**: `group_by`, `summarize`

.83

11 Appendix: Installation of R and its packages

R and RStudio have already been installed on the computers in the lecture room. Below we explain how to install R and RStudio on your own PC or laptop. We assume Windows is used, other major operating systems are also supported.

Installation Go to the R website `http://www.r-project.org`.
In the column on the left hand side, click on the `CRAN` link under the subheading **Download**. The right hand side changes to a page with "CRAN Mirrors". This is a list of servers from which the R software and extensions can be downloaded.
Click on a link (e.g. one from Austria). Now both the left hand side and the right hand side of the page change. (This page can be saved as a bookmark/favorite, so that the choice of CRAN mirror need not be made again.)
On the right hand side, click on `Download R for Windows`. Then click on `base` and click on `Download R 4.0.2 for Windows` to download/install `R-4.0.4-win.exe`. It is recommended to always install the latest version as well as to update the program if a new version is released (currently once every year, in general April). You can use the default installation options.

First Steps Start the program, e.g. via the **Start** menu. One window inside the main R window is opened, which is called the **R Console** window. Read the text that is shown in this window at startup.
In its basic form, the Console window serves as output window, but can also be used as a type of script window. For example, we can write 2+2 after the ">" sign , hit the Enter key, and the value 4 is given.
Next, write the text `demo(graphics)` after the ">" sign and hit the *Return/Enter* key twice. A graphics window is opened. Hitting *Return/Enter* changes the graph that is displayed. After a few more graphs, the demo is finished. Next write the text `help.start()` in the console window. A page is opened in your web browser with links to documentation, the installed "packages", a search engine, a "Frequently Asked Questions" and a "Windows FAQ". If you want to become a regular R user, you are strongly advised to read "An Introduction to R" and both FAQs.

A Further Look at the Website On the left hand side, several links are of interest. The most interesting ones for a starter are the links under `Documentation`. The link `Contributed` gives access to weeks of freely downloadable reading material. Also have a look at the `FAQs` once you start using R seriously. Other interesting links are `Packages` under **Software** and `Task Views` and `Search` under **CRAN**.

RStudio Download **RStudio** from `http://www.rstudio.org` and install the program.

- Open RStudio. Via the *Tools* menu, we can change the appearance of the windows (e.g. Line Wrap can be activated). Change the working directory if needed. Via the `Help` menu, we can obtain a list of shortcuts (e.g. Alt+- is a shortcut for the arrow sign "<-", with automatic creation of space around the arrow for increased readability).

12 Appendix: Some useful links

You have learned the basics of R. If you want to continue using R, we provide some links where you can find specific information that may suit your needs. See also the course website for other pointers to useful information.

- There are several sites where you can find information by topic. The site `http://a-little-book-of-r-for-biomedical-statistics.readthedocs.org/` is aimed at biomedical statistics. It also has a nice booklet with information.

  Other useful sites are
  `http://www.burns-stat.com/documents/tutorials/impatient-r`,
  `http://www.statmethods.net`,
  `http://ww2.coastal.edu/kingw/statistics/R-tutorials/index.html`,
  `http://www.mayin.org/ajayshah/KB/R/index.html`.
- Some information is aimed at those who have quite a lot of experience with some other program, for example the books, "R for SAS and SPSS users" and "R for Stata Users". More information on these books can be found at
  `http://r4stats.com/`.
  This web site also has a freely downloadable early version of these books (under the "Books" menu).
  Some free information for Stata users can be found on
  `http://dlab.berkeley.edu/blog/quick-and-easy-way-turn-your-stata-knowledge-r-knowledge`.
  And there is a manual which compares R and MATLAB at
  `http://www.math.umaine.edu/~hiebeler/comp/matlabR.html`.
- There are lots of videos that explain R. A good one can be found via
  `http://blog.revolutionanalytics.com/2012/12/coursera-videos.html`.
- Many books have been written that explain how to use R for statistical analysis. See
  `http://www.r-project.org/doc/bib/R-books.html`.

  An overview of free manuals can be found at
  `http://cran.r-project.org/other-docs.html` and
  `http://cran.r-project.org/manuals.html`.
  Free manuals with a detailed description of R are
  `http://stats.idre.ucla.edu/ucla/books/#R`.
  This latter university also hosts a website with detailed examples of all types of statistical analysis that is highly recommended
  `https://stats.idre.ucla.edu/other/dae/`.
- If you think you need a graphical user interface like SPSS has, there are several options. A popular GUI for beginners in R is R Commander
  `http://socserv.mcmaster.ca/jfox/Misc/Rcmdr`).

  Other GUIs are found at
  `http://www.deducer.org`,
  `https://rkward.kde.org/`.
- You have learned to write your code and perform your analyses in RStudio. A more extended environment is Eclipse. See `http://www.walware.de/goto/statet`.
  Other editors are

Tinn-R (`http://sourceforge.net/projects/tinn-r`),
WinEdt (`http://www.winedt.com`) together with the RWinEdt plug-in,
see `http://cran.r-project.org/web/packages/RWinEdt/`,
Notepad++ (`http://notepad-plus-plus.org/`) together with the R-plugin (`http://sourceforge.net/projects/npptor`)
or Relax
`http://www.wiwi.uni-bielefeld.de/~wolf/software/relax/relax.html`.

Especially for those who work with "Big Data", another environment is from `https://www.microsoft.com/en-us/sql-server/machinelearningserver`.
Academic users can download the program for free.

- A web site with many beautiful graphs and the corresponding R code is `http://rgraphgallery.blogspot.nl/`.
- There are programs that allow to run R from Excel (`http://rcom.univie.ac.at`)
and from SPSS (search on `R plug-in SPSS` at `https://www.ibm.com/developerworks/`).