Learning R

. . . as you learnt your mother tongue

Department of Biosciences, University of Helsinki

Learning R

...as you learnt your mother tongue

Draft

Pedro J. Aphalo

Helsinki, 15th April 2016

Unpublished

This file includes only some of the chapters to be included when complete.

Contents

Preface

This series of Notes covers different aspects of the use of R. They are meant to be used as a complement to a course or book, as explanations are rather short and terse. We do not discuss here statistics, just R as a tool and language for data manipulation and display. The idea is for you to learn the R language like children learn a language: they work-out what the rules are simply by listening to people speak and trying to utter what they want to tell their parents. I do give some explanations and comments, but the idea of these notes is mainly for you to use the numerous examples to find-out by yourself the overall patterns and coding philosophy behind the R language. Instead of parents being the sound board for your first utterances in R, the computer will play this role. You should look and try to repeat the examples, and then try your own hand and see how the computer responds, does it understand you or not?

I recommend you to use as an editor or IDE (integrated development environment) RStudio. RStudio is user friendly, actively maintained, and available both in desktop and server versions. The desktop version runs on Windows, Linux, and OS X and other Unixes. In addition it is available for free! R itself also runs under all these operating systems and a few more. Being R a command line application in its simplest incarnation, it can be made to work on relatively low computing resources like a personal computer of a couple of decades ago. Nowadays R can be made to run even on the Raspberry Pi, a Linux micro-controller board with the processing power of a modest smartphone. At the other end of the spectrum on really powerful servers it can be used for the analysis of big data sets with millions of observations.

Do not expect to ever know everything about R! R in a broad sense is vast because its capabilities can be expanded with independently developed packages. Currently there are thousands of packages publicly available for free. You just need to learn what you need. Being very popular there is nowadays lots of information available, plus a helpful and open minded on-line community willing

to help with those difficult problems for which Goggle will not be of help.

I have been using R since around 1998 or 1999, but I am still learning new things all the time. With time it has replaced in my work as a researcher and teacher several other pieces of software: SPSS, Systat, Origin, Excel, and it has become a central piece of the tool set I use for producing lecture slides, notes, books and even web pages. This is to say that it is the most useful piece of software and programming language I have ever learnt to use. Of course, in time it will be replaced by something better, but at the moment it is the "hot" thing to learn for anybody with a need to analyse and present data.

I hope you find these notes useful, but they are not meant to be read passively. The idea is that you will run all the code examples and try as many other variations as needed until you are sure to understand the rules of the R language and how each function or command works. In R for each function, data set, etc. there is a help page available. In addition, if you use RStudio, auto-completion is available as well as balloon help on the arguments accepted by functions. For scripts, there is syntax checking of the source code before its execution: *possible* mistakes and even formatting style problems are highlighted in the editor window. Error messages tend to be terse in R, and may require some lateral thinking and/or 'experimentation' to understand the real cause behind problems. When you are not sure to understand how some command works, it is useful in many cases to try simple examples for which you know the correct answer and see if you can reproduce them in R.

Do not expect to find a single answer or approach consistently recommended. Many computations can be done in R, as in any language, in several different ways, still obtaining the same result. The different approaches may differ mainly in two aspects: 1) how readable to humans are the instructions given to the computer as part of a script or program, and 2) how fast the code will run. Unless performance is an important bottleneck in your work, just concentrate on writing code that is easy to understand to you and to others, and consequently easy to check and reuse. Of course do always check any code you write for mistakes, preferably using actual numerical test cases for any complex calculation. When writing code, as for any other text intended for humans to read, consistent writing style and formatting go a long way in making your intentions clear.

These notes are work-in-progress. I will appreciate suggestions for further examples, notification of errors and unclear sections and also any larger contributions. Many of the examples here have been collected from diverse sources over many years and because of this not all sources are acknowledged. If you recognize any example as yours or someone else's please let me know so that I can add a proper acknowledgement. I warmly thank the students that over the years have asked the questions and posed the problems that have helped me

write this text and correct the mistakes and voids of previous versions. I have also received help on on-line forums and in person from numerous people, learnt from archived e-mail list messages, blog posts, books, articles, tutorials, webinars, and by struggling to solve some new problems on my own. In many ways this text owes much more to people who are not authors than to myself. However, as I am the one who has written this version and decided what to include and exclude, as author, I take full responsibility for any errors and inaccuracies.

1

Introduction

Although I whined and tried to hide under the rug, my inexorable publisher demanded an introduction…

— Ursula K. Le Guin, *Buffalo Gals and other Animal Presences*, 1985

1.1 R's built-in help

To access help pages through the command prompt we use function `help()` or a question mark. Every object exported by an R package (functions, methods, classes, data) is documented. Sometimes a single help page documents several R objects. Usually at the end of the help pages some us examples are given.

```
help("sum")
?sum
```

1.2 Obtaining help from on-line forums

Netiquette
  StackOverflow

1.3 Online webbinars and courses

1.4 R, editors and IDEs

1.5 Packages and repositories

1.6 Other tools

1

2

R as a powerful calculator

The desire to economize time and mental effort in arithmetical computations, and to eliminate human liability to error, is probably as old as the science of arithmetic itself.

— Howard Aiken, *Proposed automatic calculating machine*, presented to IBM in 1937

2.1 Aims of this chapter

In my experience, for those not familiar with computing programming or scripting languages, and who have mostly used computer programs through visual interfaces making heavy use of menus and icons, the best first step in learning R is to learn the basics of the language through its use at the R command prompt. This will teach not only the syntax and grammar rules, but also give a glimpse at the advantages and flexibility of this approach to data analysis.

Menu-driven programs are not necessarily bad, they are just unsuitable when there is a need to set very many options and chose from many different actions. They are also difficult to maintain when extensibility is desired, and when independently developed modules of very different characteristics need to be integrated. Textual languages also have the advantage, to be dealt with in the next chapter, that command sequences can be stored as a human- and computer readable text file that keeps a record of all the steps used and that in most cases makes it trivial to reproduce the same steps at a later time. The scripts are also a very simple and handy way of communicating to others how to do a given data analysis.

2.2 Working at the R console

I assume here that you have installed or have had installed by someone else R and RStudio and that you are already familiar enough with RStudio to find your

way around its user interface. The examples in this chapter use only the console window, and results are printed to the console. The values stored in the different variables are visible in the Environment tab in RStudio.

In the console you can type commands at the > prompt. When you end a line by pressing the return key, if the line can be interpreted as an R command, the result will be printed in the console, followed by a new > prompt. If the command is incomplete a + continuation prompt will be shown, and you will be able to type-in the rest of the command. For example if the whole calculation that you would like to do is $1 + 2 + 4$, if you enter in the console `1 + 2 +` in one line, you will get a continuation prompt where you will be able to type 3. However, if you type `1 + 2`, the result will be calculated, and printed.

When working at the command prompt, results are printed by default, but in other cases you may need to use the function `print` explicitly. The examples here rely on the automatic printing.

The idea with these examples is that you learn by working out how different commands work based on the results of the example calculations listed. The examples are designed so that they allow the rules, and also a few quirks, to be found by 'detective work'. This should hopefully lead to better understanding than just studying rules.

2.3 Examples with numbers

When working with arithmetic expressions the normal mathematical precedence rules are respected, but parentheses can be used to alter this order. Parentheses can be nested and at all nesting levels the normal rounded parentheses are used. The number of opening (left side) and closing (right side) parentheses must be balanced, and they must be located so that each enclosed term is a valid mathematical expression. For example while `(1 + 2) * 3` is valid, `(1 +) 2 * 3` is a syntax error as `1 +` is incomplete and cannot be calculated.

```
1 + 1

## [1] 2

2 * 2

## [1] 4

2 + 10 / 5

## [1] 4

(2 + 10) / 5
```

```
## [1] 2.4
10^2 + 1
## [1] 101
sqrt(9)
## [1] 3
pi # whole precision not shown when printing
## [1] 3.141593
print(pi, digits=22)
## [1] 3.1415926535897931
sin(pi) # oops! Read on for explanation.
## [1] 1.224606e-16
log(100)
## [1] 4.60517
log10(100)
## [1] 2
log2(8)
## [1] 3
exp(1)
## [1] 2.718282
```

One can use variables to store values. Variable names and all other names in R are case sensitive. Variables a and A are two different variables. Variable names can be quite long, but usually it is not a good idea to use very long names. Here I am using very short names, that is usually a very bad idea. However, in cases like these examples where the stored values have no real connection to the real world and are used just once or twice, these names emphasize the abstract nature.

```
a <- 1
a + 1
## [1] 2
```

```
a

## [1] 1

b <- 10
b <- a + b
b

## [1] 11

3e-2 * 2.0

## [1] 0.06
```

There are some syntactically legal statements that are not very frequently used, but you should be aware that they are valid, as they will not trigger error messages, and may surprise you. The important thing is that you write commands consistently. `1  ->  a` is valid but rarely used. The use of the equals sign (=) for assignment although valid is generally discouraged as it is seldom used as this use has not earlier been part of the R language. Chaining assignments as in the first line below is sometimes used, and signals to the human reader that `a`, `b` and `c` are being assigned the same value.

```
a <- b <- c <- 0.0
a

## [1] 0

b

## [1] 0

c

## [1] 0

1 -> a
a

## [1] 1

a = 3
a

## [1] 3
```

Numeric variables can contain more than one value. Even single numbers are `vectors` of length one. We will later see why this is important. As you have seen

above the results of calculations were printed preceded with [1]. This is the index or position in the vector of the first number (or other value) displayed at the head of the current line.

One can use c 'concatenate' to create a vector of numbers from individual numbers.

```
a <- c(3,1,2)
a

## [1] 3 1 2

b <- c(4,5,0)
b

## [1] 4 5 0

c <- c(a, b)
c

## [1] 3 1 2 4 5 0

d <- c(b, a)
d

## [1] 4 5 0 3 1 2
```

One can also create sequences, or repeat values. In this case I leave to the reader to work out the rules by running these and his/her own examples.

```
a <- -1:5
a

## [1] -1  0  1  2  3  4  5

b <- 5:-1
b

## [1]  5  4  3  2  1  0 -1

c <- seq(from = -1, to = 1, by = 0.1)
c

##  [1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2
## [10] -0.1  0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7
## [19]  0.8  0.9  1.0

d <- rep(-5, 4)
d

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

Now something that makes R different from most other programming languages: vectorized arithmetic.

```
a + 1 # we add one to vector a defined above

## [1] 0 1 2 3 4 5 6

(a + 1) * 2

## [1]  0  2  4  6  8 10 12

a + b

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

a - a

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

It can be seen in first line above, another peculiarity of R, that is frequently called "recycling": as vector a is of length 6, but the constant 1 is a vector of length 1, this 1 is extended by recycling into a vector of the same length as the longest vector in the statement.

Make sure you understand what calculations are taking place in the chunk above, and also the one below.

```
a <- rep(1, 6)
a

## [1] 1 1 1 1 1 1

a + 1:2

## [1] 2 3 2 3 2 3

a + 1:3

## [1] 2 3 4 2 3 4

a + 1:4

## Warning in a + 1:4:  longer object length is not a multiple of shorter
object length

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

A useful thing to know: a vector can have length zero. Vectors of length zero may seem at first sight quite useless, but in fact they are very useful. They

allow the handling of "no input" or "nothing to do" cases as normal cases, which in the absence of vectors of length zero would require to be treated as special cases. We also introduce here two useful functions, `length()` which returns the length of a vector, and `is.numeric()` that can be used to test if an R object is `numeric`.

```r
z <- numeric(0)
z

## numeric(0)

length(z)

## [1] 0

is.numeric(z)

## [1] TRUE
```

It is possible to *remove* variables from the workspace with `rm`. Function `ls()` returns a list all objects in the current environment, or by supplying a `pattern` argument, only the objects with names matching the `pattern`. The pattern is given as a regular expression, with `[]` enclosing alternative matching characters, `∧` and `$` indicating the extremes of the name (start and end, respectively). For example `"∧z$"` matches only the single character 'z' while `"∧z"` matches any name starting with 'z'. In contrast `"∧[zy]$"` matches both 'z' and 'y' but neither 'zy' nor 'yz', and `"∧[a-z]"` matches any name starting with a lower case ASCII letter. If you are using RStudio, all objects are listed in the Environment pane, and the search box of the panel can be used to find a given object.

```r
ls(pattern="∧z$")

## [1] "z"

rm(z)
try(z)
ls(pattern="∧z$")

## character(0)
```

There are some special values available for numbers. NA meaning 'not available' is used for missing values. Calculations can yield also the following values NaN 'not a number', Inf and -Inf for ∞ and −∞. As you will see below, calculations yielding these values do **not** trigger errors or warnings, as they are arithmetically valid. Inf and -Inf are also valid numerical values for input and constants.

```
a <- NA
a

## [1] NA

-1 / 0

## [1] -Inf

1 / 0

## [1] Inf

Inf / Inf

## [1] NaN

Inf + 4

## [1] Inf

b <- -Inf
b * -1

## [1] Inf
```

Not available (NA) values are very important in the analysis of experimental data, as frequently some observations are missing from an otherwise complete data set due to "accidents" during the course of an experiment. It is important to understand how to interpret NA's. They are simple place holders for something that is unavailable, in other words *unknown*. Any operation, even tests of equality, involving one or more NA's return an NA. In other words when one input to a calculation is unknown, the result of the calculation is unknown.

```
A <- NA
A

## [1] NA

A + 1

## [1] NA

A + Inf

## [1] NA
```

One thing to be aware of, and which we will discuss again later, is that numbers

in computers are almost always stored with finite precision. This means that they not always behave as Real numbers as defined in mathematics. In R the usual numbers are stored as `double-precision floats`, which means that there are limits to the largest and smallest numbers that can be represented (approx. $-1 \cdot 10^{308}$ and $1 \cdot 10^{308}$), and the number of significant digits that can be stored (usually described as ϵ (epsilon, abbreviated `eps`, defined as the largest number for which $1 + \epsilon = 1$)). This can be sometimes important, and can generate unexpected results in some cases, especially when testing for equality. In the example below, the result of the subtraction is still exactly 1.

```
1 - 1e-20

## [1] 1
```

It is usually safer not to test for equality to zero when working with numeric values. One alternative is comparing against a suitably small number, which will depend on the situation, although `eps` is usually a safe bet, unless the expected range of values is known to be small. This type of precautions are specially important in what is usually called "production" code: a script or program that will be used many times and with little further intervention by the researcher or programmer. Such code must work correctly, or not work at all, and it should not under any imaginable circumstance possibly give a wrong answer.

```
eps <- .Machine$double.eps
abs(-1)

## [1] 1

abs(1)

## [1] 1

x <- 1e-40
abs(x) < eps * 2

## [1] TRUE

abs(x) < 1e-100

## [1] FALSE
```

The same precautions apply to tests for equality, so whenever possible according to the logic of the calculations, it is best to test for inequalities, for example using `x <= 1.0` instead of `x == 1.0`. If this is not possible, then the tests should be treated as above, for example replacing `x == 1.0` with

abs(x - 1.0) < eps. Function abs() returns the absolute value, in simple words, makes all values positive or zero, by changing the sign of negative values.

When comparing integer values these problems do not exist, as integer arithmetic is not affected by loss of precision in calculations restricted to integers (the L comes from 'long' a name sometimes used for a machine representation of integers. Because of the way integers are stored in the memory of computers, within the acceptable range, they are stored exactly. One can think of computer integers as a subset of whole numbers restricted to a certain range of values.

```
1L + 3L

## [1] 4

1L * 3L

## [1] 3

1L %/% 3L

## [1] 0

1L %% 3L

## [1] 1

1L / 3L

## [1] 0.3333333
```

The last statement in example immediately above, using the 'usual' division operator yields a floating-point double result, while the integer division operator %/% yields an integer result, and %% returns the remainder from the integer division.

Both doubles and integers are considered numeric. In most situations conversion is automatic and we do not need to worry about the differences between these two types of numeric values. This last chunk shows values returned that are either TRUE or FALSE. These are logical values that will be discussed in the next section.

```
is.numeric(1L)

## [1] TRUE

is.double(1L)

## [1] FALSE
```

12

```
is.double(1L / 3L)
```

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

```
is.numeric(1L / 3L)
```

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

2.4 Examples with logical values

What in maths are usually called Boolean values, are called `logical` values in R. They can have only two values TRUE and FALSE, in addition to NA (not available). They are vectors as all other simple types in R. There are also logical operators that allow Boolean algebra (and support for set operations that we will only describe very briefly). In the chunk below we work with logical vectors of length one.

```
a <- TRUE
b <- FALSE
a
```

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

```
!a # negation
```

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

```
a && b # logical AND
```

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

```
a || b # logical OR
```

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

Again vectorization is possible. I present this here, and will come back to this later, because this is one of the most troublesome aspects of the R language for beginners. There are two types of 'equivalent' logical operators that behave differently, but use similar syntax! The vectorized operators have single-character names & and |, while the non vectorized ones have double-character names && and ||. There is only one version of the negation operator ! that is vectorized. In some, but not all cases, a warning will indicate that there is a possible problem.

```
a <- c(TRUE,FALSE)
b <- c(TRUE,TRUE)
a

## [1]  TRUE FALSE

b

## [1] TRUE TRUE

a & b # vectorized AND

## [1]  TRUE FALSE

a | b # vectorized OR

## [1] TRUE TRUE

a && b # not vectorized

## [1] TRUE

a || b # not vectorized

## [1] TRUE
```

Functions `any` and `all` take a logical vector as argument, and return a single logical value 'summarizing' the logical values in the vector. `all` returns TRUE only if every value in the argument is TRUE, and `any` returns TRUE unless every value in the argument is FALSE.

```
any(a)

## [1] TRUE

all(a)

## [1] FALSE

any(a & b)

## [1] TRUE

all(a & b)

## [1] FALSE
```

Another important thing to know about logical operators is that they 'short-cut' evaluation. If the result is known from the first part of the statement, the

rest of the statement is not evaluated. Try to understand what happens when you enter the following commands. Short-cut evaluation is useful, as the first condition can be used as a guard preventing a later condition to be evaluated when its computation would result in an error (and possibly abort of the whole computation).

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

When using the vectorized operators on vectors of length greater than one, 'short-cut' evaluation still applies for the result obtained.

```
a & b & NA
## [1]    NA FALSE
a & b & c(NA, NA)
## [1]    NA FALSE
a | b | c(NA, NA)
## [1] TRUE TRUE
```

2.5 Comparison operators

Comparison operators yield as a result logical values.

```
1.2 > 1.0

## [1] TRUE

1.2 >= 1.0

## [1] TRUE

1.2 == 1.0 # be aware that here we use two = symbols

## [1] FALSE

1.2 != 1.0

## [1] TRUE

1.2 <= 1.0

## [1] FALSE

1.2 < 1.0

## [1] FALSE

a <- 20
a < 100 && a > 10

## [1] TRUE
```

   Again these operators can be used on vectors of any length, returning as result a logical vector.

```
a <- 1:10
a > 5

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE

a < 5

##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

a == 5
```

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

all(a > 5)

## [1] FALSE

any(a > 5)

## [1] TRUE

b <- a > 5
b

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE

any(b)

## [1] TRUE

all(b)

## [1] FALSE
```

Be once more aware of 'short-cut evaluation'. If the result would not be affected by the missing value then the result is returned. If the presence of the NA makes the end result unknown, then NA is returned.

```
c <- c(a, NA)
c > 5

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE    NA

all(c > 5)

## [1] FALSE

any(c > 5)

## [1] TRUE

all(c < 20)

## [1] NA

any(c > 20)

## [1] NA
```

```
is.na(a)

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

is.na(c)

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE  TRUE

any(is.na(c))

## [1] TRUE

all(is.na(c))

## [1] FALSE
```

This behaviour can be changed by using the optional argument `na.rm` which removes NA values **before** the function is applied. (Many functions in R have this optional parameter.)

```
all(c < 20)

## [1] NA

any(c > 20)

## [1] NA

all(c < 20, na.rm=TRUE)

## [1] TRUE

any(c > 20, na.rm=TRUE)

## [1] FALSE
```

You may skip until the end of the section on first read, also see page 11. Here are some examples for which the finite resolution of computer machine floats as compared to Real numbers as defined in mathematics makes a difference.

```
1e20 == 1 + 1e20

## [1] TRUE

1 == 1 + 1e-20

## [1] TRUE
```

```
0 == 1e-20

## [1] FALSE
```

As R can run on different types of computer hardware, the actual machine limits may vary. It is possible to obtain this values from variable `.Machine`.

```
.Machine$double.eps

## [1] 2.220446e-16

.Machine$integer.max

## [1] 2147483647
```

In many situations, when writing programs one should avoid testing for equality of floating point numbers ('floats'). Here we show how to handle gracefully rounding errors. As the example shows, some rounding errors may accumulate, and in practice `.Machine$double.eps` may be too large a value to safely use in tests for zero.

```
a == 0.0 # may not always work

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

abs(a) < 1e-15 # is safer

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [8] FALSE FALSE FALSE

sin(pi) == 0.0 # angle in radians, not degrees!

## [1] FALSE

sin(2 * pi) == 0.0

## [1] FALSE

abs(sin(pi)) < 1e-15

## [1] TRUE

abs(sin(2 * pi)) < 1e-15

## [1] TRUE

sin(pi)
```

```
## [1] 1.224606e-16

sin(2 * pi)

## [1] -2.449213e-16

.Machine$double.eps # see help for .Machine for explanation

## [1] 2.220446e-16

.Machine$double.neg.eps

## [1] 1.110223e-16
```

2.6 Character values

Character variables can be used to store any character. Character constants are written by enclosing characters in quotes. There are three types of quotes in the ASCII character set, double quotes ", single quotes ', and back ticks '. The first two types of quotes can be used for delimiting characters. There are in R two predefined vectors with characters for letters stored in alphabetical order.

```
a <- "A"
b <- letters[2]
c <- letters[1]
a

## [1] "A"

b

## [1] "b"

c

## [1] "a"

d <- c(a, b, c)
d

## [1] "A" "b" "a"

e <- c(a, b, "c")
e

## [1] "A" "b" "c"

h <- "1"
try(h + 2)
```

Vectors of characters are not the same as character strings. In character vectors each position in the vector is occupied by a single character, while in character strings, each string of characters, like a word enclosed in double or single quotes occupies a single position or slot in the vector.

```
f <- c("1", "2", "3")
g <- "123"
f == g

## [1] FALSE FALSE FALSE

f

## [1] "1" "2" "3"

g

## [1] "123"
```

One can use the 'other' type of quotes as delimiter when one wants to include quotes within a string. Pretty-printing is changing what I typed into how the string that is stored in R: I typed b <- 'He said "hello" when he came in' in the second statement below, try it.

```
a <- "He said 'hello' when he came in"
a

## [1] "He said 'hello' when he came in"

b <- 'He said "hello" when he came in'
b

## [1] "He said \"hello\" when he came in"
```

The outer quotes are not part of the string, they are 'delimiters' used to mark the boundaries. As you can see when b is printed special characters can be represented using 'escape sequences'. There are several of them, and here we will show just two, newline and tab. We also show here the different behaviour of print() and cat(), with cat() *interpreting* the escape sequences and print() not.

```
c <- "abc\ndef\txyz"
print(c)

## [1] "abc\ndef\txyz"

cat(c)
```

```
## abc
## def xyz
```

Above, you will not see any effect of these escapes when using `print`: `\n` represents 'new line' and `\t` means 'tab' (tabulator). The *scape codes* work only in some contexts, as when using `cat` to generate the output. They also are very useful when one wants to split an axis-label, title or label in a plot into two or more lines as they can be embedded in any string.

2.7 Finding the 'mode' of objects

Variables have *mode* that depends on what can be stored in them. But differently to other languages, assignment of to variable of a different mode is allowed and in most cases its mode changes together with its contents. However, there is a restriction that all elements in a vector, array or matrix, must be of the same mode, while this is not required for lists. Functions with names starting with `is.` are tests returning a logical value, TRUE, FALSE or NA.

```
my_var <- 1:5
mode(my_var)

## [1] "numeric"

is.numeric(my_var)

## [1] TRUE

is.logical(my_var)

## [1] FALSE

is.character(my_var)

## [1] FALSE

my_var <- "abc"
mode(my_var)

## [1] "character"
```

2.8 Type conversions

The least intuitive ones are those related to logical values. All others are as one would expect. By convention, functions used to convert objects from one mode to a different one have names starting with `as.`.

```r
as.character(1)
```

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

```r
as.character(3.0e10)
```

```
## [1] "3e+10"
```

```r
as.numeric("1")
```

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

```r
as.numeric("5E+5")
```

```
## [1] 5e+05
```

```r
as.numeric("A")
```

```
## Warning:  NAs introduced by coercion
```

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

```r
as.numeric(TRUE)
```

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

```r
as.numeric(FALSE)
```

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

```r
TRUE + TRUE
```

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

```r
TRUE + FALSE
```

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

```r
TRUE * 2
```

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

```r
FALSE * 2
```

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

```r
as.logical("T")
```

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

```r
as.logical("t")
```

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

```r
as.logical("TRUE")
```

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

```r
as.logical("true")
```

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

```r
as.logical(100)
```

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

```r
as.logical(0)
```

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

```r
as.logical(-1)
```

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

```r
f <- c("1", "2", "3")
g <- "123"
as.numeric(f)
```

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

```r
as.numeric(g)
```

```
## [1] 123
```

Some tricks useful when dealing with results. Be aware that the printing is being done by default, these functions return numerical values that are different from their input. Look at the help pages for further details. Very briefly `round` is used to round numbers to a certain number of decimal places after or before the decimal point, while `signif()` keeps the requested number of significant digits.

```r
round(0.0124567, 3)
```

```
## [1] 0.012
```

```
round(0.0124567, 1)
```

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

```
round(0.0124567, 5)
```

```
## [1] 0.01246
```

```
signif(0.0124567, 3)
```

```
## [1] 0.0125
```

```
round(1789.1234, 3)
```

```
## [1] 1789.123
```

```
signif(1789.1234, 3)
```

```
## [1] 1790
```

```
a <- 0.12345
b <- round(a, 2)
a == b
```

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

```
a - b
```

```
## [1] 0.00345
```

```
b
```

```
## [1] 0.12
```

When applied to vectors, `signif` behaves slightly differently.

```
signif(c(123, 0.123), 3)
```

```
## [1] 123.000   0.123
```

Other functions relevant to the formatting of numbers and other output are `format()`, and `sprintf()`.

2.9 Vectors

You already know how to create a vector. Now we are going to see how to extract individual elements (e.g. numbers or characters) out of a vector. Elements are

accessed using an index. The index indicates the position in the vector, starting from one, following the usual mathematical tradition. What in maths would be x_i for a vector x, in R is represented as x[i]. (In R indexes (or subscripts) always start from one, while in some other programming languages indexes start from zero.)

```
a <- letters[1:10]
a

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[2]

## [1] "b"

a[c(3,2)]

## [1] "c" "b"

a[10:1]

##  [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a"
```

The examples below demonstrate what is the result of using a longer vector of indexes than the indexed vector. The length of the indexing vector has no restriction, but the acceptable range of values for the indexes is given by the length of the indexed vector.

```
a[c(3,3,3,3)]

## [1] "c" "c" "c" "c"

a[c(10:1, 1:10)]

##  [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "a" "a"
## [12] "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Negative indexes have a special meaning, they indicate the positions at which values should be excluded.

```
a[-2]

## [1] "a" "c" "d" "e" "f" "g" "h" "i" "j"

a[-c(3,2)]

## [1] "a" "d" "e" "f" "g" "h" "i" "j"
```

Results from indexing with out-of-range values may be surprising.

```
a[11]
## [1] NA
a[1:11]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" NA
```

Results from indexing with special values may be surprising.

```
a[ ]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
a[numeric(0)]
## character(0)
a[NA]
##  [1] NA NA NA NA NA NA NA NA NA NA
a[c(1, NA)]
## [1] "a" NA
a[NULL]
## character(0)
a[c(1, NULL)]
## [1] "a"
```

Another way of indexing, which is very handy, but not available in most other programming languages, is indexing with a vector of logical values. In practice, the vector of logical values used for 'indexing' is in most cases of the same length as the vector from which elements are going to be selected. However, this is not a requirement, and if the logical vector is shorter it is 'recycled' as discussed above in relation to operators.

```
a[TRUE]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
a[FALSE]
```

27

```
## character(0)

a[c(TRUE, FALSE)]

## [1] "a" "c" "e" "g" "i"

a[c(FALSE, TRUE)]

## [1] "b" "d" "f" "h" "j"

a > "c"

##  [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE

a[a > "c"]

## [1] "d" "e" "f" "g" "h" "i" "j"

selector <- a > "c"
a[selector]

## [1] "d" "e" "f" "g" "h" "i" "j"

which(a > "c")

## [1]  4  5  6  7  8  9 10

indexes <- which(a > "c")
a[indexes]

## [1] "d" "e" "f" "g" "h" "i" "j"

b <- 1:10
b[selector]

## [1]  4  5  6  7  8  9 10

b[indexes]

## [1]  4  5  6  7  8  9 10
```

Make sure to understand the examples above. These type of constructs are very widely used in R scripts because they allow for concise code that is easy to understand once you are familiar with the indexing rules.

Indexing can be used on both sides of an assignment. This may look rather esoteric at first sight, but it is just a simple extension of the logic of indexing described above.

28

```
a <- 1:10
a

## [1]  1  2  3  4  5  6  7  8  9 10

a[1] <- 99
a

## [1] 99  2  3  4  5  6  7  8  9 10

a[c(2,4)] <- -99
a

## [1]  99 -99   3 -99   5   6   7   8   9  10

a[TRUE] <- 1
a

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

a <- 1
```

We can also have subscripting on both sides.

```
a <- letters[1:10]
a

## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[1] <- a[10]
a

## [1] "j" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a <- a[10:1]
a

## [1] "j" "i" "h" "g" "f" "e" "d" "c" "b" "j"

a[10:1] <- a
a

## [1] "j" "b" "c" "d" "e" "f" "g" "h" "i" "j"

a[5:1] <- a[c(TRUE,FALSE)]
a

## [1] "i" "g" "e" "c" "j" "f" "g" "h" "i" "j"
```

Do play with subscripts to your heart's content, really grasping how they work and can how they can be used will be very useful in anything you do in the future with R.

2.10 Factors

Factors are used for indicating categories, most frequently the factors describing the treatments in an experiment, or categories in a survey. They can be created either from numerical or character vectors. The different possible values are called *levels*. Normal factors created with `factor` are unordered or categorical. R also defines ordered factors that can be created with function `ordered`.

```r
my.vector <- c("treated", "treated", "control", "control", "control", "treated")
my.factor <- factor(my.vector)
my.factor <- factor(my.vector, levels=c("treatment", "control"))
```

It is always preferable to use meaningful names for levels, although it is possible to use numbers. The order of levels becomes important when plotting data, as it affects the order of the levels along the axes, or in legends. Converting factors to numbers is not intuitive, because even if the levels look like numbers when displayed, they are just character strings.

```r
my.vector2 <- rep(3:5, 4)
my.factor2 <- factor(my.vector2)
as.numeric(my.factor2)

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

as.numeric(as.character(my.factor2))

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

Internally factor levels are stored as running numbers starting from one, and those are the numbers returned by `as.numeric()` when applied to a factor.

Factors are very important in R. In contrast to other statistical software in which the role of a variable is set when defining a model to be fitted or setting up a test, in R models are specified exactly in the same way for ANOVA and regression analysis, as linear models. What 'decides' what type of model is fitted is whether the explanatory variable is a factor (giving ANOVA) or a numerical variable (giving regression). This makes a lot of sense, as in most cases, considering an explanatory variable as categorical or not, depends on the design of the experiment or survey, in other words, is a property of the data rather than of the analysis.

2.11 Lists

Elements of a `list` are not ordered, and can be of different type. Lists can be also nested. Elements in list are named, and normally are accessed by name. Lists are defined using function `list`.

```r
a.list <- list(x = 1:6, y = "a", z = c(TRUE, FALSE))
a.list
## $x
## [1] 1 2 3 4 5 6
##
## $y
## [1] "a"
##
## $z
## [1]  TRUE FALSE

str(a.list)
## List of 3
##  $ x: int [1:6] 1 2 3 4 5 6
##  $ y: chr "a"
##  $ z: logi [1:2] TRUE FALSE

a.list$x
## [1] 1 2 3 4 5 6

a.list[["x"]]
## [1] 1 2 3 4 5 6

a.list[[1]]
## [1] 1 2 3 4 5 6

a.list["x"]
## $x
## [1] 1 2 3 4 5 6

a.list[1]
## $x
## [1] 1 2 3 4 5 6

a.list[c(1,3)]
## $x
## [1] 1 2 3 4 5 6
##
## $z
## [1]  TRUE FALSE

try(a.list[[c(1,3)]])
## [1] 3
```

Using double square brackets for indexing gives the element stored in the list, in its original mode, in the example above, `a.list[["x"]]` returns a numeric vector, while `a.list[1]` returns a list containing the numeric vector x. `a.list$x` returns the same value as `a.list[["x"]]`, a numeric vector. While `a.list[c(1,3)]` returns a list of length two, `a.list[[c(1,3)]]`.

2.12 Data frames

Data frames are a special type of list, in which each element is a vector or a factor of the same length. The are created with function `data.frame` with a syntax similar to that used for lists. When a shorter vector is supplied as argument, it is recycled, until the full length of the variable is filled. This is very different to what we obtained in the previous section when we created a list.

```
a.df <- data.frame(x = 1:6, y = "a", z = c(TRUE, FALSE))
a.df

##   x y     z
## 1 1 a  TRUE
## 2 2 a FALSE
## 3 3 a  TRUE
## 4 4 a FALSE
## 5 5 a  TRUE
## 6 6 a FALSE

str(a.df)

## 'data.frame': 6 obs. of  3 variables:
##  $ x: int  1 2 3 4 5 6
##  $ y: Factor w/ 1 level "a": 1 1 1 1 1 1
##  $ z: logi  TRUE FALSE TRUE FALSE TRUE FALSE

a.df$x

## [1] 1 2 3 4 5 6

a.df[["x"]]

## [1] 1 2 3 4 5 6

a.df[[1]]

## [1] 1 2 3 4 5 6

class(a.df)

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

R is an object oriented language, and objects belong to classes. With function `class` we can query the class of an object. As we saw in the two previous chunks lists and data frames objects belong to two different classes.

We can add also to lists and data frames.

```
a.df$x2 <- 6:1
a.df$x3 <- "b"
a.df

##   x y     z x2 x3
## 1 1 a  TRUE  6  b
## 2 2 a FALSE  5  b
## 3 3 a  TRUE  4  b
## 4 4 a FALSE  3  b
## 5 5 a  TRUE  2  b
## 6 6 a FALSE  1  b
```

We have added two columns to the data frame, and in the case of column `x3` recycling took place. Data frames are extremely important to anyone analysing or plotting data in R. One can think of data frames as tightly structured worksheets, or as lists. As you may have guessed from the examples earlier in this section, there are several different ways of accessing columns, rows, and individual observations stored in a data frame. The columns can to some extent be treated as elements in a list, and can be accessed both by name or index (position). When accessed by name, using $ or double square brackets a single column is returned as a vector or factor. In contrast to lists, data frames are 'rectangular' and for this reason the values stored can be also accessed in a way similar to how elements in a matrix are accessed, using two indexes. As we saw for vectors indexes can be vectors of integer numbers or vectors of logical values. For columns they can in addition be vectors of character strings matching the names of the columns. When using indexes it is extremely important to remember that the indexes are always given **row first**.

```
a.df[ , 1]    # first column

## [1] 1 2 3 4 5 6

a.df[ , "x"] # first column

## [1] 1 2 3 4 5 6

a.df[1, ]     # first row

##   x y    z x2 x3
## 1 1 a TRUE  6  b
```

33

```
a.df[1:2, c(FALSE, FALSE, TRUE, FALSE, FALSE)]

## [1]  TRUE FALSE

            # first two rows of the third column
a.df[a.df$z , ] # the rows for which z is true

##   x y    z x2 x3
## 1 1 a TRUE  6  b
## 3 3 a TRUE  4  b
## 5 5 a TRUE  2  b

a.df[a.df$x > 3, -3] # the rows for which x > 3 for

##   x y x2 x3
## 4 4 a  3  b
## 5 5 a  2  b
## 6 6 a  1  b

                # all columns except the third one
```

When the names of data frames are long, complex conditions become awkward to write. In such cases `subset` is handy because evaluation is done in the 'environment' of the data frame, i.e. the names of the columns are recognized if entered directly.

```
subset(a.df, x > 3)

##   x y     z x2 x3
## 4 4 a FALSE  3  b
## 5 5 a  TRUE  2  b
## 6 6 a FALSE  1  b
```

When calling functions that return a vector, data frame, or other structure, the square brackets can be appended to the rightmost parenthesis of the function call, in the same way as to the name of a variable holding the same data.

```
subset(a.df, x > 3)[ , -3]

##   x y x2 x3
## 4 4 a  3  b
## 5 5 a  2  b
## 6 6 a  1  b

subset(a.df, x > 3)$x

## [1] 4 5 6
```

None of the examples in the last three code chunks alter the original data frame `a.df`. We can store the returned value using a new name, if we want to preserve `a.df` unchanged, or we can assign the result to `a.df` deleting in the process the original `a.df`. The next to examples do assignment to `a.df`, but either to only one columns, or by indexing the individual values in both the 'right side' and 'left side' of the assignment. Another way to delete a column from a data frame is to assign NULL to it.

```
a.df[["x2"]] <- NULL
a.df$x3 <- NULL
a.df

##   x y      z
## 1 1 a   TRUE
## 2 2 a  FALSE
## 3 3 a   TRUE
## 4 4 a  FALSE
## 5 5 a   TRUE
## 6 6 a  FALSE
```

In the previous code chuck we deleted the last two columns of the data frame `a.df`. Finally an esoteric trick for you think about.

```
a.df[1:6, c(1,3)] <- a.df[6:1, c(3,1)]
a.df

##   x y z
## 1 0 a 6
## 2 1 a 5
## 3 0 a 4
## 4 1 a 3
## 5 0 a 2
## 6 1 a 1
```

Although in this last example we used numeric indexes to make in more interesting, in practice, especially in scripts or other code that will be reused, do use column names instead of positional indexes. This makes your code much more reliable, as changes elsewhere in the script are much less likely to lead to undetected errors.

2.13 Simple built-in statistical functions

Being R's main focus in statistics, it provides functions for both simple and complex calculations, going from means and variances to fitting very complex models. we will start with the simple ones.

```
x <- 1:20
mean(x)

## [1] 10.5

var(x)

## [1] 35

median(x)

## [1] 10.5

mad(x)

## [1] 7.413

sd(x)

## [1] 5.91608

range(x)

## [1]   1 20

max(x)

## [1] 20

min(x)

## [1] 1

length(x)

## [1] 20
```

2.14 Functions and execution flow control

Although functions can be defined and used at the command prompt, we will discuss them when looking at scripts in the next chapter. We will do the same in the case of flow-control statements (e.g. repetition and conditional execution).

Significance tests and model fitting will be the subject of later chapters, not yet written.

3

R Scripts and Programming

An R script is simply a text file containing (almost) the same commands that you would enter on the command line of R.

— Kickstarting R

3.1 What is a script?

We call *script* to a text file that contains the same commands that you would type at the console prompt. A true script is not for example an MS-Word file where you have pasted or typed some R commands. A script file has the following characteristics.

- The script is a text file (ASCII or some other encoding e.g. UTF-8 that R uses in your set-up).

- The file contains valid R statements (including comments) and nothing else.

- Comments start at a # and end at the end of the line. (True end-of line as coded in file, the editor may wrap it or not at the edge of the screen).

- The R statements are in the file in the order that they must be executed.

- R scripts have file names ending in `.r` or `.R`.

It is good practice to write scripts so that they are self-contained. Such a scripts will run in a new R session by including library commands to load all the required packages.

3.2 How do we use a scrip?

A script can be sourced.

If we have a text file called `my.first.script.r` containing the following text:

```
# this is my first R script
print(3+4)
```

And then source this file:

```
source("my.first.script.r")
## [1] 7
```

The results of executing the statements contained in the file will appear in the console. The commands themselves are not shown (the sourced file is not echoed) and the results will not be printed unless you include an explicit `print` command in the script. This applies in many cases also to plots—e.g. A figure created with `ggplot` needs to be printed if we want to see it when the script is run. Adding a redundant `print` is harmless.

From within RStudio, if you have an R script open in the editor, there will a "source" drop box (≠ DropBox) visible from where you can choose "source" as described above, or "source with echo" for the currently open file.

When a script is sourced, the output can be saved to a text file instead of being shown in the console. It is also easy to call R with the script file as argument directly at the command prompt of the operating system.

```
RScript my.first.script.r
```

You can open a 'shell' from the Tools menu in RStudio, to run this command. The output will be printed to the shell console. If you would like to save the output to a file, use redirection.

```
RScript my.first.script.r > my.output.txt
```

Sourcing is very useful when the script is ready, however, while developing a script, or sometimes when testing things, one usually wants to run (= execute) one or a few statements at a time. This can be done using the "run" button after either locating the cursor in the line to be executed, or selecting the text that one would like to run (the selected text can be part of a line, a whole line, or a group of lines, as long as it is syntactically valid).

3.3 How to write a script?

The approach used, or mix of approaches will depend on your preferences, and on how confident you are that the statements will work as expected.

If one is very familiar with similar problems One would just create a new text file and write the whole thing in the editor, and then test it. This is rather unusual.

If one if moderately familiar with the problem One would write the script as above, but testing it, part by part as one is writing it. This is usually what I do.

If ones mostly playing around Then if one is using RStudio, one type statements at the console prompt. As you should know by now, everything you run at the console is saved to the "History". In RStudio the History is displayed in its own pane, and in this pane one can select any previous statement and by pressing a single having copy and pasted to either the console prompt, or the cursor position in the file visible in the editor. In this way one can build a script by copying and pasting from the history to your script file the bits that have worked as you wanted.

3.3.1 Exercise

By now you should be familiar enough with R to be able to write your own script.

1. Create a new R script (in RStudio, from 'File' menu, "+" button, or by typing "Ctrl + Shift + N").

2. Save the file as "my.second.script.r".

3. Use the editor pane in RStudio to type some R commands and comments.

4. **Run** individual commands.

5. **Source** the whole file.

3.4 The need to be understandable to people

When you write a script, it is either because you want to document what you have done or you want re-use it at a later time. In either case, the script itself although still meaningful for the computer could become very obscure to you, and even more to someone seeing it for the first time.

How does one achieve an understandable script or program?

- Avoid the unusual. People using a certain programming language tend to use some implicit or explicit rules of style[1]. As a minimum try to be consistent with yourself.

- Use meaningful names for variables, and any other object. What is meaningful depends on the context. Depending on common use a single letter may be more meaningful than a long word. However self explaining names are better: e.g. using `n.rows` and `n.cols` is much clearer than using `n1` and `n2` when dealing with a matrix of data. Probably `number.of.rows` and `number.of.columns` would just increase the length of the lines in the script, and one would spend more time typing without getting much in return.

- How to make the words visible in names: traditionally in R one would use dots to separate the words and use only lower case. Some years ago, it became possible to use underscores. The use of underscores is quite common nowadays because in some contexts is "safer" as in some situations a dot may have a special meaning. What we call "camel case" is only infrequently used in R programming but is common in other languages like Pascal. An example of camel case is `NumCols`. In some cases it can become a bit confusing as in `UVMean` or `UvMean`.

3.4.1 Exercise

Here is an example of bad style in a script, edit so that it becomes easier to read.

```r
a <- 2 # height
b <- 4 # length
C <-
    a *
b
C -> variable
      print(
"area: ", variable
)
```

3.5 Functions

When writing scripts, or any program, one should avoid repeating blocks of code (groups of statements). The reasons for this are: 1) if the code needs to be changed,

---

[1]Style includes *indentation* of statements, *capitalization* of variable and function names.

you have to make changes in more than one place in the file, or in more than one file. Sooner or later, some copies will remain unchanged by mistake. 2) it makes the script file longer, and this makes debugging, commenting, etc. more tedious, and error prone.

How do we avoid repeating bits of code? We write a function containing the statements that we would need to repeat, and then `call` (use) the function in their place.

Functions are defined by means of **function**, and saved like any other object in R by assignment to a variable. In the example below x and y are both formal parameters, or names used within the function for objects that will be supplied as "arguments" when the function is called. One can think of parameter names as place-holders.

```
my.prod <- function(x, y){x * y}
my.prod(4, 3)

## [1] 12
```

First some basic knowledge. In R, arguments are passed by copy. This is something very important to remember. Whatever you do within a function to modify an argument, its value outside the function will remain (almost) always unchanged.

```
my.change <- function(x){x <- NA}
a <- 1
my.change(a)
a

## [1] 1
```

Any result that needs to be made available outside the function must be returned by the function. If the function `return` is not explicitly used, the value returned by the last statement *executed* within the body of the function will be returned.

```
print.x.1 <- function(x){print(x)}
print.x.1("test")

## [1] "test"

print.x.2 <- function(x){print(x); return(x)}
print.x.2("test")

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

```
print.x.3 <- function(x){return(x); print(x)}
print.x.3("test")

## [1] "test"

print.x.4 <- function(x){return(); print(x)}
print.x.4("test")

## NULL

print.x.5 <- function(x){x}
print.x.4("test")

## NULL
```

Now we will define a useful function: a function for calculating the standard error of the mean from a numeric vector.

```
SEM <- function(x){sqrt(var(x)/length(x))}
a <- c(1, 2, 3, -5)
a.na <- c(a, NA)
SEM(x=a)

## [1] 1.796988

SEM(a)

## [1] 1.796988

SEM(a.na)

## [1] NA
```

For example in SEM(a) we are calling function SEM with a as argument.

The function we defined above may sometimes give a wrong answer because NAs will be counted by length, so we need to remove NAs before calling length.

```
simple_SEM <- function(x) {
 sqrt(var(x, na.rm=TRUE)/length(na.omit(x)))
}
a <- c(1, 2, 3, -5)
a.na <- c(a, NA)
simple_SEM(x=a)

## [1] 1.796988

simple_SEM(a)

## [1] 1.796988
```

```
simple_SEM(a.na)
```

```
## [1] 1.796988
```

R does not have a function for standard error, so the function above would be generally useful. If we would like to make this function both safe, and consistent with other R functions, one could define it as follows, allowing the user to provide a second argument which is passed as an argument to `var`:

```
SEM <- function(x, na.rm=FALSE){
  sqrt(var(x, na.rm=na.rm)/length(na.omit(x)))
}
SEM(a)
```

```
## [1] 1.796988
```

```
SEM(a.na)
```

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

```
SEM(a.na, TRUE)
```

```
## [1] 1.796988
```

```
SEM(x=a.na, na.rm=TRUE)
```

```
## [1] 1.796988
```

```
SEM(TRUE, a.na)
```

```
## Warning in if (na.rm) "na.or.complete" else "everything":  the condition
has length > 1 and only the first element will be used
```

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

```
SEM(na.rm=TRUE, x=a.na)
```

```
## [1] 1.796988
```

In this example you can see that functions can have more than one parameter, and that parameters can have default values to be used if no argument is supplied. In addition if the name of the parameter is indicated, then arguments can be supplied in any order, but if parameter names are not supplied, then arguments are assigned to parameters based on their position. Once one parameter name is given, all later arguments need also to be explicitly matched to parameters. Obviously if given by position, then arguments should be supplied explicitly for all parameters at 'intermediate' positions.

3.5.1 Exercise

1) Test the behaviour of `print.x.1` and `print.x.5` at the command prompt, and in a script, by writing a script. The behaviour of one of these functions will be different when the script is source than at the command prompt. Explain why.

2) Define your own function to calculate the mean in a similar way as SEM() was defined above. Hint: function `sum()` could be of help.

3) Create some additional vectors containing NAs or not. Use them to test functions `simple_SEM()` and SEM() defined above, and then explain why SEM() returns always the correct value, even though "`na.omit(x)`" is non-conditionally (always) applied to `x` before calculating its length.

3.6 Control of execution flow

We call control of execution statements those that allow the execution of sections of code when a certain dynamically computed condition is TRUE. Some of the control of execution flow statements, function like 'ON-OFF switches' for program statements. Others, allow statements to executed repeatedly while or until a condition is met, or until all members of a list or a vector are processed.

3.6.1 Conditional execution

Non-vectorized

R has two types of "if" statements, non-vectorized and vectorized. We will start with the non-vectorized one, which is similar to what is available in most other computer programming languages.

Before this we need to explain compound statements. Individual statements can be grouped into compound statements by enclosed them in curly braces.

```
print("A")

## [1] "A"

{
  print("B")
  print("C")
}

## [1] "B"
## [1] "C"
```

44

The example above is pretty useless, but becomes useful when used together with 'control' constructs. The `if` construct controls the execution of one statement, however, this statement can be a compound statement of almost any length or complexity. Play with the code below by changing the value assigned to `printing`, including NA, and logical(0).

```r
printing <- TRUE
if (printing) {
  print("A")
  print("B")
}

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

The condition '( )' can be anything yielding a logical vector, however, as this is not vectorized, only the first element will be used. Play with this example by changing the value assigned to `a`.

```r
a <- 10.0
if (a < 0.0) print("'a' is negative") else print("'a' is not negative")

## [1] "'a' is not negative"

print("This is always printed")

## [1] "This is always printed"
```

As you can see above the statement immediately following `else` is executed if the condition is false. Later statements are executed independently of the condition.

Do you still remember the rules about continuation lines?

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

```r
# 1
a <- 1
if (a < 0.0)
  print("'a' is negative") else
    print("'a' is not negative")

## [1] "'a' is not negative"
```

Why does the statement below (not here) trigger an error?

```
# 2 (not evaluated here)
if (a < 0.0) print("'a' is negative")
else print("'a' is not negative")
```

Play with the use conditional execution, with both simple and compound statements, and also think how to combine `if` and `else` to select among more than two options.

There is in R a `switch` statement, that we describe here, which can be used to select among "cases", or several alternative statements, based on an expression evaluating to a number or a character string. The switch statement returns a value, the value returned by the code corresponding to the matching switch value, or the default if there is no match, and a default has been included in the code. Both character values or numeric values can used.

```
my.object <- "two"
b <- switch(my.object,
            one = 1,
            two = 1 / 2,
            three = 1/ 4,
            0
)
b

## [1] 0.5
```

Do play with the use of the switch statement.

Vectorized

Vectorized conditional execution is coded by means of a **function** called `ifelse` (one word). This function takes three arguments: a logical vector, a result vector for TRUE, a result vector for FALSE. All three can be any construct giving the necessary argument as their return value. In the case of result vectors, recycling will apply if they are not of the correct length. The length of the result is determined by the length of the logical vector in the first argument!.

```
a <- 1:10
ifelse(a > 5, 1, -1)

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

ifelse(a > 5, a + 1, a - 1)

##  [1]  0  1  2  3  4  7  8  9 10 11
```

```r
ifelse(any(a>5), a + 1, a - 1) # tricky

## [1] 2

ifelse(logical(0), a + 1, a - 1) # even more tricky

## logical(0)

ifelse(NA, a + 1, a - 1) # as expected

## [1] NA
```

Try to understand what is going on in the previous example. Create your own examples to test how `ifelse` works.

Exercise: write using `ifelse` a single statement to combine numbers from a and b into a result vector d, based on whether the corresponding value in c is the character "a" or "b".

```r
a <- rep(-1, 10)
b <- rep(+1, 10)
c <- c(rep("a", 5), rep("b", 5))
# your code
```

If you do not understand how the three vectors are built, or you cannot guess the values they contain by reading the code, print them, and play with the arguments, until you have clear what each parameter does.

3.6.2 Why using vectorized functions and operators is important

If you have written programs in other languages, it would feel to you natural to use loops (for, repeat while, repeat until) for many of the things for which we have been using vectorization. When using the R language it is best to use vectorization whenever possible, because it keeps the listing of scripts and programs shorter and easier to understand (at least for those with experience in R). However, there is another very important reason: execution speed. The reason behind this is that R is an interpreted language. In current versions of R it is possible to byte-compile functions, but this is rarely used for scripts, and even byte-compiled loops are much slower and vectorized functions.

However, there are cases were we need to repeatedly execute statements in a way that cannot be vectorized, or when we do not need to maximize execution speed. The R language does have loop constructs, and we will describe them next.

3.6.3 Repetition

The most frequently used type of loop is a `for` loop. These loops work in R are based on lists or vectors of values to act upon.

```r
b <- 0
for (a in 1:5) b <- b + a
b

## [1] 15

b <- sum(1:5) # built-in function
b

## [1] 15
```

Here the statement `b <- b + a` is executed five times, with a sequentially taking each of the values in `1:5`. Instead of a simple statement used here, also a compound statement could have been used.

Here are a few examples that show some of the properties of `for` loops and functions, combined with the use of a function.

```r
test.for <- function(x) {
  for (i in x) {print(i)}
}
test.for(numeric(0))
test.for(1:3)

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

test.for(NA)

## [1] NA

test.for(c("A", "B"))

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

test.for(c("A", NA))

## [1] "A"
## [1] NA

test.for(list("A", 1))

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

```
test.for(c("z", letters[1:4]))

## [1] "z"
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

In contrast to other languages, in R function arguments are not checked for 'type' when the function is called. The only requirement is that the function code can handle the argument provided. In this example you can see that the same function works with numeric and character vectors, and with lists. We haven't seen lists before. As earlier discussed all elements in a vector should have the same type. This is not the case for lists. It is also interesting to note that a list or vector of length zero is a valid argument, that triggers no error, but that as one would expect, causes the statements in the loop body to be skipped.

Some examples of use of `for` loops — and of how to avoid there use.

```
a <- c(1, 4, 3, 6, 8)
for(x in a) x*2 # result is lost
for(x in a) print(x*2) # print is needed!

## [1] 2
## [1] 8
## [1] 6
## [1] 12
## [1] 16

b <- for(x in a) x*2 # doesn't work as expected, but triggers no error
b

## NULL

for(x in a) b <- x*2 # a bit of a surprise, as b is not a vector!
b

## [1] 16

for(i in seq(along=a)) {
  b[i] <- a[i]^2
  print(b)
}

## [1] 1
## [1]  1 16
## [1]  1 16  9
## [1]  1 16  9 36
## [1]  1 16  9 36 64
```

```
b # is a vector!

## [1]  1 16  9 36 64

# a bit faster if we first allocate a vector of the required length
b <- numeric(length(a))
for(i in seq(along=a)) {
  b[i] <- a[i]^2
  print(b)
}

## [1] 1 0 0 0 0
## [1]  1 16  0  0  0
## [1]  1 16  9  0  0
## [1]  1 16  9 36  0
## [1]  1 16  9 36 64

b # is a vector!

## [1]  1 16  9 36 64

# vectorization is simplest and fastest
b <- a^2
b

## [1]  1 16  9 36 64
```

Look at the results from the above examples, and try to understand where does the returned value come from in each case.

We sometimes may not be able to use vectorization, or may be easiest to not use it. However, whenever working with large data sets, or many similar datasets, we will need to take performance into account. As vectorization usually also makes code simpler, it is good style to use it whenever possible.

```
b <- numeric(length(a)-1)
for(i in seq(along=b)) {
  b[i] <- a[i+1] - a[i]
  print(b)
}

## [1] 3 0 0 0
## [1]  3 -1  0  0
## [1]  3 -1  3  0
## [1]  3 -1  3  2

# although in this case there were alternatives, there
# are other cases when we need to use indexes explicitly
b <- a[2:length(a)] - a[1:length(a)-1]
b
```

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

```
# or even better
b <- diff(a)
b
```

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

`seq(along=b)` builds a new numeric vector with a sequence of the same length as the length as the vector given as argument for parameter 'along'.

`while` loops are quite frequently also useful. Instead of a list or vector, they take a logical argument, which is usually an expression, but which can also be a variable. For example the previous calculation could be also done as follows.

```
a <- c(1, 4, 3, 6, 8)
i <- 1
while (i < length(a)) {
  b[i] <- a[i]^2
  print(b)
  i <- i + 1
}
```

```
## [1]   1 -1   3   2
## [1]   1 16   3   2
## [1]   1 16   9   2
## [1]   1 16   9 36
```

```
b
```

```
## [1]   1 16   9 36
```

Here is another example. In this case we use the result of the previous iteration in the current one. In this example you can also see, that it is allowed to put more than one statement in a single line, in which case the statements should be separated by a semicolon (;).

```
a <- 2
while (a < 50) {print(a); a <- a^2}
```

```
## [1] 2
## [1] 4
## [1] 16
```

```
print(a)
```

```
## [1] 256
```

51

Make sure that you understand why the final value of `a` is larger than 50.

`repeat` is seldom used, but adds flexibility as `break` can be in the middle of the compound statement.

```
a <- 2
repeat{
  print(a)
  a <- a^2
  if (a > 50) {print(a); break()}
}

## [1] 2
## [1] 4
## [1] 16
## [1] 256

# or more elegantly
a <- 2
repeat{
  print(a)
  if (a > 50) break()
  a <- a^2
}

## [1] 2
## [1] 4
## [1] 16
## [1] 256
```

Please, make sure you understand what is happening in the previous examples.

3.6.4 Nesting

All the execution-flow control statements seen above can be nested. We will show an example with two `for` loops. We first need a matrix of data to work with:

```
A <- matrix(1:50, 10)
A

##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
```

```
## [9,]    9   19   29   39   49
## [10,]  10   20   30   40   50

A <- matrix(1:50, 10, 5)
A

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1   11   21   31   41
## [2,]    2   12   22   32   42
## [3,]    3   13   23   33   43
## [4,]    4   14   24   34   44
## [5,]    5   15   25   35   45
## [6,]    6   16   26   36   46
## [7,]    7   17   27   37   47
## [8,]    8   18   28   38   48
## [9,]    9   19   29   39   49
## [10,]  10   20   30   40   50

# argument names used for clarity
A <- matrix(1:50, nrow = 10)
A

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1   11   21   31   41
## [2,]    2   12   22   32   42
## [3,]    3   13   23   33   43
## [4,]    4   14   24   34   44
## [5,]    5   15   25   35   45
## [6,]    6   16   26   36   46
## [7,]    7   17   27   37   47
## [8,]    8   18   28   38   48
## [9,]    9   19   29   39   49
## [10,]  10   20   30   40   50

A <- matrix(1:50, ncol = 5)
A

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1   11   21   31   41
## [2,]    2   12   22   32   42
## [3,]    3   13   23   33   43
## [4,]    4   14   24   34   44
## [5,]    5   15   25   35   45
## [6,]    6   16   26   36   46
## [7,]    7   17   27   37   47
## [8,]    8   18   28   38   48
## [9,]    9   19   29   39   49
## [10,]  10   20   30   40   50

A <- matrix(1:50, nrow = 10, ncol = 5)
A
```

```
##        [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50
```

All the statements above are equivalent, but some are easier to read than others.

```
row.sum <- numeric() # slower as size needs to be expanded
for (i in 1:nrow(A)) {
  row.sum[i] <- 0
  for (j in 1:ncol(A))
    row.sum[i] <- row.sum[i] + A[i, j]
}
print(row.sum)

##   [1] 105 110 115 120 125 130 135 140 145 150
```

```
row.sum <- numeric(nrow(A)) # faster
for (i in 1:nrow(A)) {
  row.sum[i] <- 0
  for (j in 1:ncol(A))
    row.sum[i] <- row.sum[i] + A[i, j]
}
print(row.sum)

##   [1] 105 110 115 120 125 130 135 140 145 150
```

Look at the output of these two examples to understand what is happening differently with `row.sum`.

The code above is very general, it will work with any size of two dimensional matrix, which is good programming practice. However, sometimes we need more specific calculations. `A[1, 2]` selects one cell in the matrix, the one on the first row of the second column. `A[1, ]` selects row one, and `A[ , 2]` selects column two. In the example above the value of `i` changes for each iteration of the outer loop. The value of `j` changes for each iteration of the inner loop, and the inner loop is run in full for each iteration of the outer loop. The inner loop index `j` changes fastest.

Exercises: 1) modify the example above to add up only the first three columns of A, 2) modify the example above to add the last three columns of A.

Will the code you wrote continue working as expected if the number of rows in A changed? and what if the number of columns in A changed, and the required results still needed to be calculated for relative positions? What would happen if A had fewer than three columns? Try to think first what to expect based on the code you wrote. Then create matrices of different sizes and test your code. After that think how to improve the code, at least so that wrong results are not produced.

Vectorization can be achieved in this case easily for the inner loop.

```r
row.sum <- numeric(nrow(A)) # faster
for (i in 1:nrow(A)) {
  row.sum[i] <- sum(A[i, ])
}
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

`A[i, ]` selects row `i` and all columns. In R, the row index always comes first, which is not the case in all programming languages.

Full vectorization can be achieved with `apply` functions.

```r
row.sum <- apply(A, MARGIN = 1, sum) # MARGIN=1 inidcates rows
print(row.sum)

##  [1] 105 110 115 120 125 130 135 140 145 150
```

How would you change this last example, so that only the last three columns are added up? (Think about use of subscripts to select a part of the matrix.)

There are many variants of `apply` functions, both in base R and in contributed packages.

3.7 Packages

In R speak 'library' is the location where 'packages' are installed. Packages are sets of functions, and data, specific for some particular purpose, that can be loaded into an R session to make them available so that they can be used in the same way as built-in R functions and data. The function `library` is used to load packages, already installed in the local R library, into the current session, while the function `install.packages` is used to install packages, either from a file, or directly from the internet into the library. When using RStudio it is easiest to use RStudio commands (which call `install.packages` and `update.packages`) to install and update packages.

```
library(graphics)
```

Currently there are thousands of packages available. The most reliable source of packages is CRAN, as only packages that pass strict tests and are actively maintained are included. In some cases you may need or want to install less stable code, and this is also possible. With package `devtools` it is even possible to install packages directly from Github, Bitbucket and a few other repos. These later installations are always installations from source (see below).

R packages can be installed either from source, or from already built 'binaries'. Installing from sources, depending on the package, may require quite a lot of additional software to be available. Under MS-Windows, very rarely the needed shell, commands and compilers are already available. Installing then is not too difficult (you will need RTools, and MiKTeX). For this reason it is the norm to install packages from binary .zip files. Under Linux most tools will be available, or very easy to install, so it is not unusual to install from sources. For OS X (Mac) the situation is somewhere in-between. If the tools are available, packages can be very easily installed from source from within RStudio.

The development of packages is beyond the scope of the current course, but it is still interesting to know a few things about packages. Using RStudio it is relatively easy to develop your own packages. Packages can be of very different sizes. Packages use a relatively rigid structure of folder for storing the different types of files, and there is a built-in help system, that one needs to use, so that the package documentation gets linked to the R help system when the package is loaded. In addition to R code, packages can call C, C++, FORTRAN, Java, etc. functions and routines, but some kind of 'glue' is needed, as data is stored differently. At least for C++, the recently developed Rcpp R package makes the gluing extremely easy.

One good way of learning how R works, is by experimenting with it, and whenever using a certain function looking at the help, to check what are all the available options.

4

R built-in functions

4.1 Loading data

To start with we need some data. Here we use `cars`, a data set included in base R.

```r
data(cars)
```

4.2 Looking at the data

```r
names(cars)
```

```
## [1] "speed" "dist"
```

```r
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

```r
tail(cars)
```

```
##    speed dist
## 45    23   54
## 46    24   70
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85
```

```
str(cars)

## 'data.frame': 50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

The `cars` data are stored as a data frame. Data frames consist in columns of equal length. The different columns can be different types (e.g. numeric and character). With `data()` we load it; with `names()` we obtain the names of the variables or columns. With head with can see the top several lines, and with tail the lines at the end. In general `data()` is used load R objects saved in a file format used by R. Text files con be read with functions `scan()`, `read.table()`, `read.csv()` and their variants.

It is also possible to 'import' data saved in files of *foreign* formats, defined by other programs. Packages such as 'foreign', 'readr', 'readxl', 'RNetCDF', 'jsonlite', etc. allow importing data from other statistic and data analysis applications and from standard data exchange formats. It is also good to keep in mind that in R urls are accepted as arguments to the `file` argument.

4.3 Plotting

The built-in generic function `plot` can be used to plot data. It is a generic function, that has suitable methods for different kinds of objects. In this section we only very briefly demonstrate the use of the most common base R's graphics functions.

```
plot(dist ~ speed, data=cars)
```
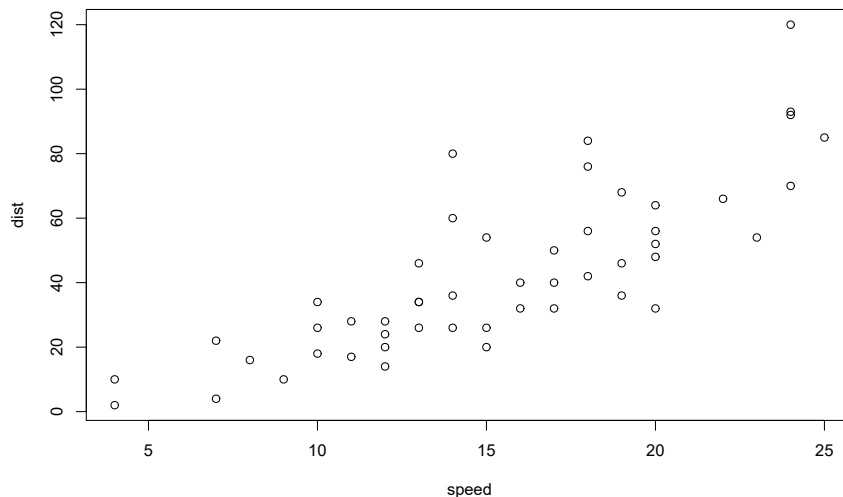
4.4 Fitting linear models

One important thing to remember is that model 'formulas' are used in different contexts: plotting, fitting of models, and tests like *t*-test. The basic syntax is rather consistently followed, although there are some exceptions.

4.4.1 Regression

The R function `lm` is used next to fit linear models. If the explanatory variable is continuous, the fit is a regression. In the example below, `speed` is a numeric variable (floating point in this case). In the ANOVA table calculated for the model fit, in this case a linear regression, we can see that the term for `speed` has only one degree of freedom (df) for the denominator.

We first fit the model and save the output as `fm1` (A name I invented to remind myself that this is the first fitted-model in this chapter.

```
fm1 <- lm(dist ~ speed, data=cars)
```

The next step is diagnosis of the fit. Are assumptions of the linear model procedure used reasonably fulfilled? In R it is most common to use plots to this end. We show here only one of the four plots normally produced. This quantile vs.

quantile plot allows to assess how much the residuals deviate from being normally distributed.

```
plot(fm1, which = 2)
```
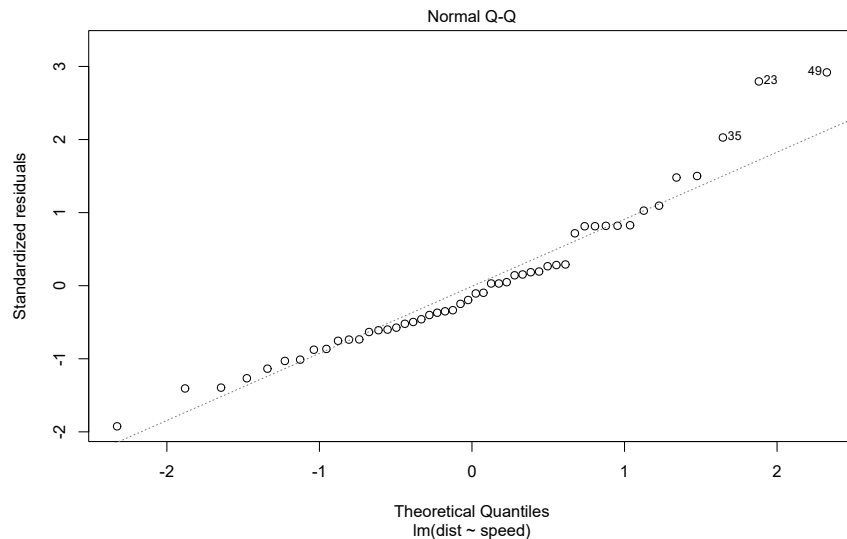

In the case of a regression, calling `summary()` with the fitted model object as argument is most useful as it provides a table of coefficient estimates and their errors. `anova()` applied to the same fitted object, returns the ANOVA table.

```
summary(fm1) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123
## speed         3.9324     0.4155   9.464 1.49e-12
##
## (Intercept) *
## speed       ***
```

```
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511,Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12

anova(fm1) # we calculate an ANOVA

## Analysis of Variance Table
##
## Response: dist
##           Df Sum Sq Mean Sq F value    Pr(>F)
## speed      1  21186 21185.5  89.567 1.49e-12 ***
## Residuals 48  11354   236.5
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Let's look at each argument separately: `dist    speed` is the specification of the model to be fitted. The intercept is always implicitly included. To 'remove' this implicit intercept from the earlier model we can use `dist    speed - 1`. In what follows we fit a straight line through the origin ($x = 0$, $y = 0$).

```
fm2 <- lm(dist ~ speed - 1, data=cars)
plot(fm2, which = 2)
summary(fm2)

##
## Call:
## lm(formula = dist ~ speed - 1, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -26.183 -12.637  -5.455   4.590  50.181
##
## Coefficients:
##       Estimate Std. Error t value Pr(>|t|)
## speed   2.9091     0.1414   20.58   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 16.26 on 49 degrees of freedom
## Multiple R-squared:  0.8963,Adjusted R-squared:  0.8942
## F-statistic: 423.5 on 1 and 49 DF,  p-value: < 2.2e-16

anova(fm2)
```

```
## Analysis of Variance Table
##
## Response: dist
##            Df Sum Sq Mean Sq F value    Pr(>F)
## speed       1 111949  111949  423.47 < 2.2e-16 ***
## Residuals  49  12954     264
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```
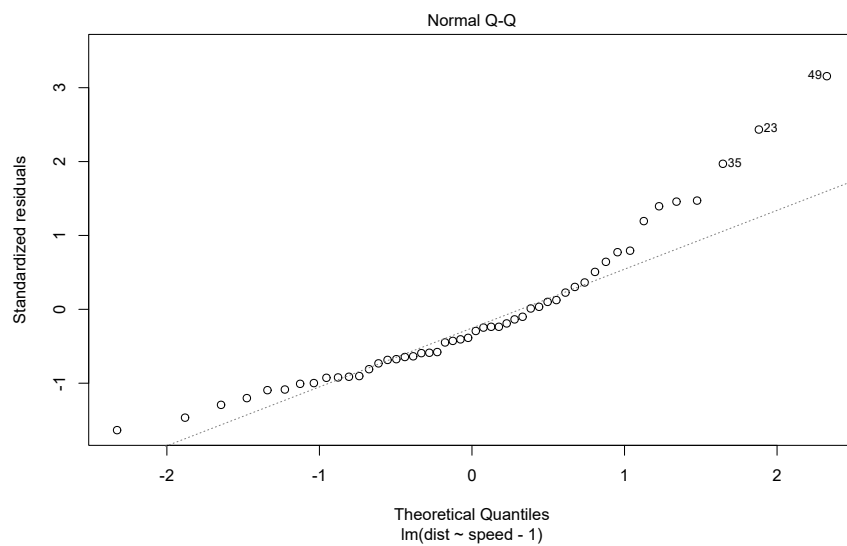

We now we fit a second degree polynomial.

```
fm3 <- lm(dist ~ speed + I(speed^2), data=cars) # we fit a model, and then save the re
plot(fm3, which = 3) # we produce diagnosis plots
summary(fm3) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ speed + I(speed^2), data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -28.720  -9.184  -3.188   4.628  45.152
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.47014   14.81716   0.167    0.868
```

```
## speed          0.91329    2.03422    0.449    0.656
## I(speed^2)    0.09996    0.06597    1.515    0.136
##
## Residual standard error: 15.18 on 47 degrees of freedom
## Multiple R-squared:  0.6673,Adjusted R-squared:  0.6532
## F-statistic: 47.14 on 2 and 47 DF,  p-value: 5.852e-12
```

```
anova(fm3) # we calculate an ANOVA
```

```
## Analysis of Variance Table
##
## Response: dist
##            Df  Sum Sq Mean Sq F value    Pr(>F)
## speed       1 21185.5 21185.5  91.986 1.211e-12
## I(speed^2)  1   528.8   528.8   2.296    0.1364
## Residuals  47 10824.7   230.3
##
## speed      ***
## I(speed^2)
## Residuals
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```
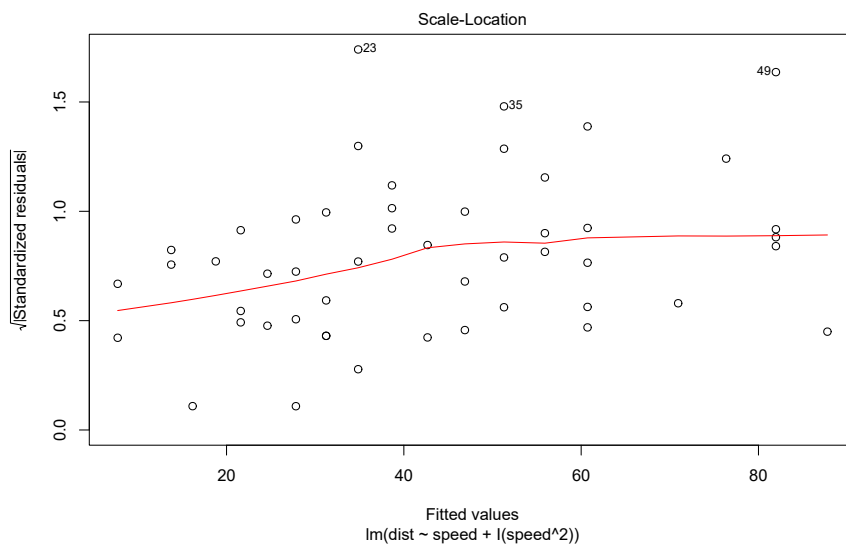

The "same" fit using an orthogonal polynomial. Higher degrees can be obtained by supplying as second argument to `poly()` a different positive integer value.

```
fm3a <- lm(dist ~ poly(speed, 2), data=cars) # we fit a model, and then save the resul
plot(fm3a, which = 3) # we produce diagnosis plots
summary(fm3a) # we inspect the results from the fit

##
## Call:
## lm(formula = dist ~ poly(speed, 2), data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -28.720  -9.184  -3.188   4.628  45.152
##
## Coefficients:
##                Estimate Std. Error t value
## (Intercept)      42.980      2.146  20.026
## poly(speed, 2)1 145.552     15.176   9.591
## poly(speed, 2)2  22.996     15.176   1.515
##                Pr(>|t|)
## (Intercept)     < 2e-16 ***
## poly(speed, 2)1 1.21e-12 ***
## poly(speed, 2)2   0.136
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.18 on 47 degrees of freedom
## Multiple R-squared:  0.6673,Adjusted R-squared:  0.6532
## F-statistic: 47.14 on 2 and 47 DF,  p-value: 5.852e-12

anova(fm3a) # we calculate an ANOVA

## Analysis of Variance Table
##
## Response: dist
##               Df Sum Sq Mean Sq F value
## poly(speed, 2)  2  21714 10857.1  47.141
## Residuals      47  10825   230.3
##                 Pr(>F)
## poly(speed, 2) 5.852e-12 ***
## Residuals
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Scale-Location

lm(dist ~ poly(speed, 2))
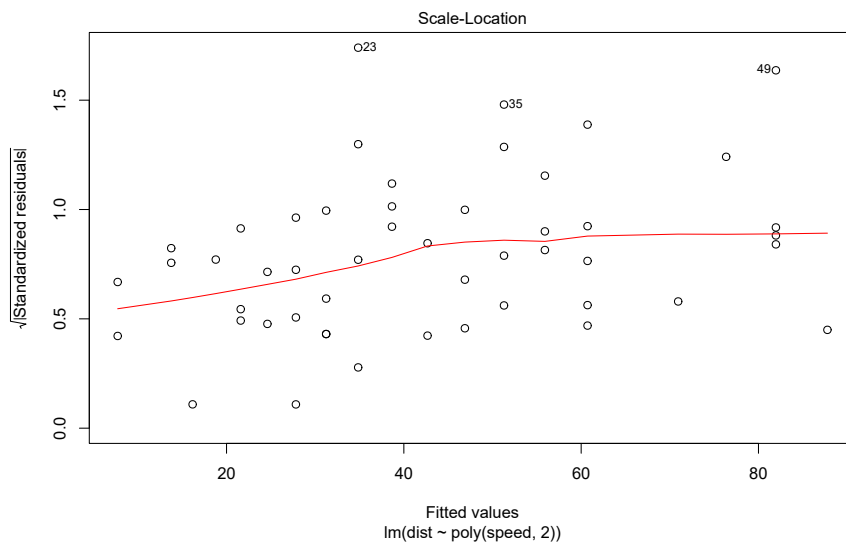
We can also compare two models, to test whether one of models describes the data better than the other.

```
anova(fm2, fm1)

## Analysis of Variance Table
##
## Model 1: dist ~ speed - 1
## Model 2: dist ~ speed
##   Res.Df   RSS Df Sum of Sq      F  Pr(>F)
## 1     49 12954
## 2     48 11354  1    1600.3 6.7655 0.01232 *
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Or three or more models. But be careful, as the order of the arguments matters.

```
anova(fm2, fm1, fm3, fm3a)

## Analysis of Variance Table
##
## Model 1: dist ~ speed - 1
## Model 2: dist ~ speed
## Model 3: dist ~ speed + I(speed^2)
## Model 4: dist ~ poly(speed, 2)
##   Res.Df   RSS Df Sum of Sq      F  Pr(>F)
```

```
## 1      49 12954
## 2      48 11354  1   1600.26 6.9482 0.01133 *
## 3      47 10825  1    528.81 2.2960 0.13640
## 4      47 10825  0      0.00
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can use different criteria to choose the best model: significance based on *P*-values or information criteria (AIC, BIC). AIC and BIC penalize the resulting 'goodness' based on the number of parameters in the fitted model. In the case of AIC and BIC, a smaller value is better, and values returned can be either positive or negative, in which case more negative is better.

```
BIC(fm2, fm1, fm3, fm3a)

##      df       BIC
## fm2   2 427.5739
## fm1   3 424.8929
## fm3   4 426.4202
## fm3a  4 426.4202

AIC(fm2, fm1, fm3, fm3a)

##      df       AIC
## fm2   2 423.7498
## fm1   3 419.1569
## fm3   4 418.7721
## fm3a  4 418.7721
```

One can see above that these three criteria not necessarily agree on which is the model to be chosen.
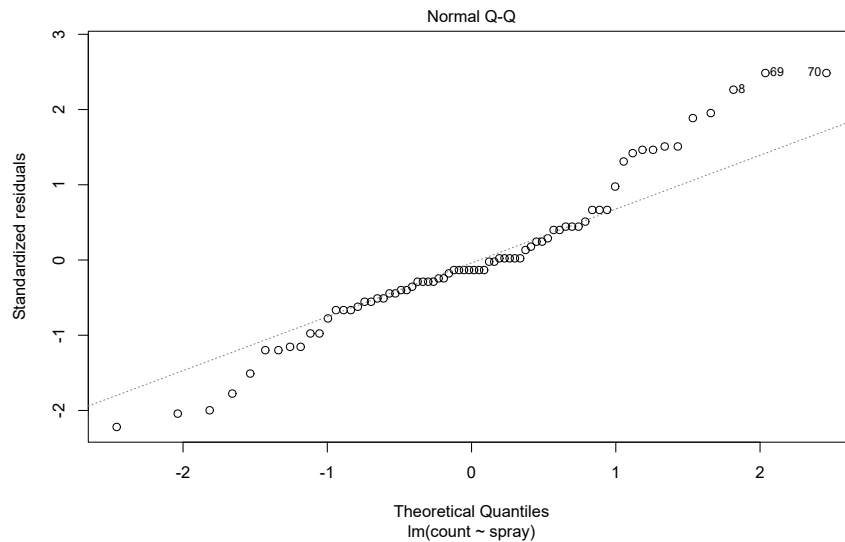
anova fm1

BIC fm1

AIC fm3

4.4.2 Analysis of variance, ANOVA

We use as the `InsectSpray` data set, giving insect counts in plots sprayed with different insecticides. In these data `spray` is a factor with six levels.

```
fm4 <- lm(count ~ spray, data = InsectSprays)
```

```
plot(fm4, which = 2)
```



```
anova(fm4)
```

```
## Analysis of Variance Table
##
## Response: count
##           Df Sum Sq Mean Sq F value    Pr(>F)
## spray      5 2668.8  533.77  34.702 < 2.2e-16 ***
## Residuals 66 1015.2   15.38
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

4.4.3 Analysis of covariance, ANCOVA

When a linear model includes both explanatory factors and continuous explanatory variables, we say that analysis of covariance (ANCOVA) is used. The formula syntax is the same for all linear models, what determines the type of analysis is the nature of the explanatory variable(s). Conceptually a factor (an unordered categorical variable) is very different from a continuous variable.

4.5 Generalized linear models

Linear models make the assumption of normally distributed residuals. Generalized linear models are more flexible, and allow the assumed distribution to be selected as well as the link function. For the analysis of the `InsectSpray` data set, above (section 4.4.2 on page 66) the Normal distribution is not a good approximation as count data deviates from it. This was visible in the quantile–quantile plot above.

For count data GLMs provide a better alternative. In the example below we fit the same model as above, but we assume a quasi-Poisson distribution instead of the Normal.

```
fm10 <- glm(count ~ spray, data = InsectSprays, family = quasipoisson)
plot(fm10, which = 2)
anova(fm10, test = "F")

## Analysis of Deviance Table
##
## Model: quasipoisson, link: log
##
## Response: count
##
## Terms added sequentially (first to last)
##
##
##       Df Deviance Resid. Df Resid. Dev      F
## NULL                    71     409.04
## spray  5   310.71        66      98.33 41.216
##          Pr(>F)
## NULL
## spray < 2.2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```
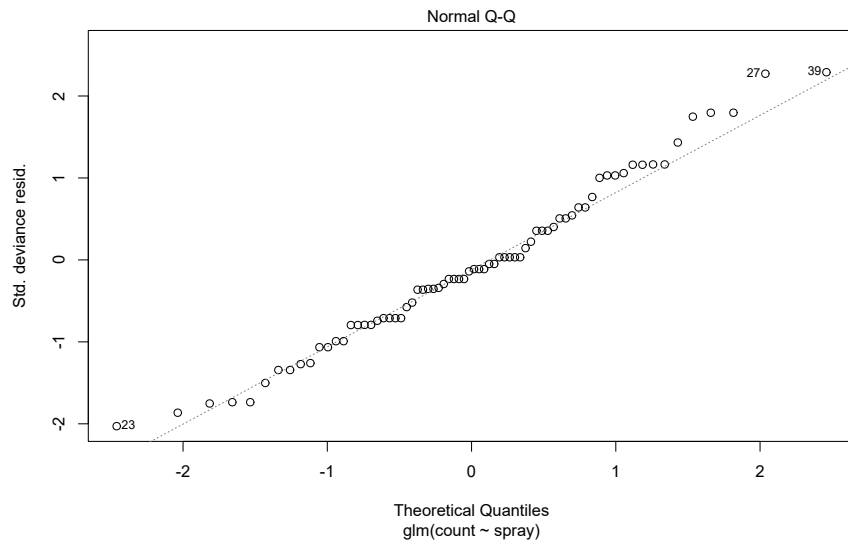
5

Further reading about R

5.1 Introductory texts

Dalgaard 2008; Teetor 2011; Zuur et al. 2009

5.2 Texts on specific aspects

Chang 2013; Everitt and Hothorn 2011; Faraway 2004; Faraway 2006; Fox 2002; Fox and Weisberg 2010

5.3 Advanced texts

Ihaka and Gentleman 1996; Matloff 2011; Murrell 2011; Pinheiro and Bates 2000; Wickham 2014, 2015; Xie 2013

Bibliography

Chang, W. (2013). *R Graphics Cookbook*. 1-2. Sebastopol: O'Reilly Media, p. 413. ISBN: 9781449316952. URL: `http://medcontent.metapress.com/index/A65RM03P4874243N.pdf` (cit. on p. 71).

Dalgaard, P. (2008). *Introductory Statistics with R*. Springer, p. 380. ISBN: 0387790543 (cit. on p. 71).

Everitt, B. and T. Hothorn (2011). *An Introduction to Applied Multivariate Analysis with R*. Springer, p. 288. ISBN: 1441996494. URL: `http://www.amazon.co.uk/Introduction-Applied-Multivariate-Analysis-Use/dp/1441996494` (cit. on p. 71).

Faraway, J. J. (2004). *Linear Models with R*. Boca Raton, FL: Chapman & Hall/CRC, p. 240. URL: `http://www.maths.bath.ac.uk/~jjf23/LMR/` (cit. on p. 71).

Faraway, J. J. (2006). *Extending the linear model with R: generalized linear, mixed effects and nonparametric regression models*. Chapman & Hall/CRC Taylor & Francis Group, p. 345. ISBN: 158488424X (cit. on p. 71).

Fox, J. (2002). *An {R} and {S-Plus} Companion to Applied Regression*. Thousand Oaks, CA, USA: Sage Publications. URL: `http://socserv.socsci.mcmaster.ca/jfox/Books/Companion/index.html` (cit. on p. 71).

Fox, J. and H. S. Weisberg (2010). *An R Companion to Applied Regression*. SAGE Publications, Inc, p. 472. ISBN: 141297514X. URL: `http://www.amazon.com/An-R-Companion-Applied-Regression/dp/141297514X` (cit. on p. 71).

Ihaka, R. and R. Gentleman (1996). 'R: A Language for Data Analysis and Graphics'. In: *J. Comput. Graph. Stat.* 5, pp. 299-314 (cit. on p. 71).

Matloff, N. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, p. 400. ISBN: 1593273843. URL: `http://www.amazon.com/The-Art-Programming-Statistical-Software/dp/1593273843` (cit. on p. 71).

Murrell, P. (2011). *R Graphics, Second Edition (Chapman & Hall/CRC The R Series)*. CRC Press, p. 546. ISBN: 1439831769. URL: `http://www.amazon.com/Graphics-Second-Edition-Chapman-Series/dp/1439831769` (cit. on p. 71).

Pinheiro, J. C. and D. M. Bates (2000). *Mixed-Effects Models in S and S-Plus*. New York: Springer (cit. on p. 71).

Teetor, P. (2011). *R Cookbook*. 1st ed. Sebastopol: O'Reilly Media, p. 436. ISBN: 9780596809157 (cit. on p. 71).

Wickham, H. (2014). *Advanced R*. Chapman & Hall/CRC The R Series. CRC Press. ISBN: 9781466586970. URL: `https://books.google.fi/books?id=G5PNBQAAQBAJ` (cit. on p. 71).

– (2015). *R Packages*. O'Reilly Media. ISBN: 9781491910542. URL: `https://books.google.fi/books?id=eqOxBwAAQBAJ` (cit. on p. 71).

Xie, Y. (2013). *Dynamic Documents with R and knitr (Chapman & Hall/CRC The R Series)*. Chapman and Hall/CRC, p. 216. ISBN: 1482203537. URL: `http://www.amazon.com/Dynamic-Documents-knitr-Chapman-Series/dp/1482203537` (cit. on p. 71).

Zuur, A. F., E. N. Ieno and E. Meesters (2009). *A Beginner's Guide to R*. 1st ed. Springer, p. 236. ISBN: 0387938362. URL: `http://www.amazon.com/Beginners-Guide-Use-Alain-Zuur/dp/0387938362` (cit. on p. 71).