Notes on using R

Pedro J. Aphalo

Git: tag ‘none’, committed with hash ‘none’ on ‘none’

Contents

Preface

This series of Notes cover different aspects of the use of R. They are meant to be use as a complement to a course or book, as explanations are short and terse. We do not discuss here statistics, just R as a tool and language for data manipulation and display. The idea is a bit like how children learn a language: they work-out what the rules are simply by listening to people speak. 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.

This is work-in-progress. I will appreciate suggestions for further examples, notification of errors and unclear things and any bigger 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 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 resources. R can be made to run on the Raspberry Pi, a Linux micro-controller board with the processing power of a modest mobile phone. At the other end of the spectrum on really powerful servers like at CSC 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 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 help page available. In addition, if you use RStudio, auto-completion is available as well as balloon help on the possible arguments to functions. For scripts, there is syntax checking of the source code before its execution available in RStudio: *possible* mistakes and even formatting style problems are highlighted in the editor window. Error messages tend to be terse, 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 too aspects: 1) how readable to humans are the instructions given to the computer as part of the 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.

1 R as a powerful calculator

1.1 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 they can be stored as human- and computer readable text files that keep a record of all the steps used and that in most cases make 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 data analysis.

1.2 Working at the R console

I assume that you are already familiar with RStudio. These examples use only the console window, and results are printed to the console. The values stored in the different variables are also 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.

1.3 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 `vector`s 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`.

```
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.

```
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 $-\infty$. 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

is.double(1L / 3L)

## [1] TRUE

is.numeric(1L / 3L)

## [1] TRUE
```

1.4 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
```

1.5 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 14. 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
```

1.6 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.

1.7 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"
```

1.8 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.`.

```
as.character(1)

## [1] "1"

as.character(3.0e10)

## [1] "3e+10"

as.numeric("1")

## [1] 1

as.numeric("5E+5")

## [1] 5e+05

as.numeric("A")

## Warning:  NAs introduced by coercion

## [1] NA

as.numeric(TRUE)

## [1] 1

as.numeric(FALSE)

## [1] 0

TRUE + TRUE

## [1] 2

TRUE + FALSE

## [1] 1

TRUE * 2

## [1] 2

FALSE * 2

## [1] 0
```

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

## [1] TRUE

as.logical("t")

## [1] NA

as.logical("TRUE")

## [1] TRUE

as.logical("true")

## [1] TRUE

as.logical(100)

## [1] TRUE

as.logical(0)

## [1] FALSE

as.logical(-1)

## [1] TRUE
```

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

## [1] 1 2 3

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()`.

1.9 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]

## 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.

```
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.

1.10 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`.

```
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.

```
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.

1.11 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`.

```
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)]]`.

1.12 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 work-sheets, 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

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.

1.13 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
```

1.14 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.

2 Further reading about R

2.1 Introductory texts

2.2 Texts on specific aspects

2.3 Advanced texts

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`.

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

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`.

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/`.

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.

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`.

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`.

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

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`.

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`.

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

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

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

Wickham, H. (2015). *R Packages*. O'Reilly Media. ISBN: 9781491910542. URL: `https://books.google.fi/books?id=eqOxBwAAQBAJ`.

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`.

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`.