Marek Gągolewski
Konstancja Bobecka-Wesołowska
Przemysław Grzegorzewski

Computer Statistics with R

1. An Introduction to R

Faculty of Mathematics and Information Science
Warsaw University of Technology
[Last update: December 9, 2012]

Contents

Info

These tutorials are likely to contain bugs and typos. In case you find any don't hesitate to *contact us*! Thanks in advance!

1.1. Preliminaries

The main purpose of these tutorials is to illustrate the use of R in a modern course on applied statistics. However, it doesn't aim to replace the typical academic lecture. The student is assumed to have some background theoretical knowledge on each presented topic. Note that R is only used as a supplementary tool here, and not as an objective on its own.

R is a free, powerful and extensible software environment for statistical computing and graphics. It provides a high level programming language that is very popular among the statistical community. Although the language syntax is roughly similar to the well-known C/C++, its semantics is based on the functional language Scheme.

R may be freely downloaded from [1]. The environment works under Windows, Linux and MacOS.

There is an extensive literature on R. We strongly encourage the reader to refer to, for example, [1–4].

R version 2.15.2 was used to generate this document. However, we hope that all presented code is valid in previous and later versions of the environment. It is because we use only the most basic language structures and functions here.

Hello Math!

After launching the application, R greets the user, gives some general (e.g. copyright) information and prints the *command prompt* which, by default, is marked as:

```
>
```

Hence, R is ready to accept our commands.

To break the ice, let us first use R as a (very powerful) calculator. We type the following command and press [ENTER] to execute it. The result will be printed immediately.

```
2 + 2
## [1] 4
```

In the C programming language, every statement should be ended with a semicolon (`;`). Here it is not necessary. However, a semicolon may be used to separate multiple commands in one line.

```
2+2;
## [1] 4
2*3.5; 5+4*2
## [1] 13
```

Task

All recently executed commands may be accessed via the [↑] and [↓] keys. We may also move the cursor freely to modify a statement as we type, just like in a typical text editor. Try to use [←], [→], [HOME] or [END] and see what happens.

It is advisable to get used to comment the code. Any characters appearing after the hash sign (`#`) are ignored by the interpreter:

[1] `http://www.r-project.org/`

```
2/4  # division (this is my first comment)
## [1] 0.5
```

Info

Working with "bare" R is often inconvenient. Try some more sophisticated GUI like *RStudio*.

We are now ready to learn about some important data types available in the R programming language.

1.2. Built-in data types

1.2.1. Atomic vectors

*Vector*s are the most commonly used data types in R.

1.2.1.1. Numeric vectors

The statement:

```
2 + 2
## [1] 4
```

is in fact an arithmetic operation on two numeric vectors of length one. It gives a numeric vector of the same length as a result.

The easiest way to create a numeric vector of arbitrary length is by *enumeration*. This is done by using the *concatenate* function, `c()`.

```
c(4, 6, 5, 3)
## [1] 4 6 5 3
```

Note

R is a case-sensitive programming language. It means that the above command written as `C()` (upper-case letter) will (most likely) end up with an error message.

Since each single numeric value is a vector, why not to try concatenating longer vectors into one?

```
c(1, 2, c(3,4,5), c(6,7), 8);   # it works!
## [1] 1 2 3 4 5 6 7 8
```

Most arithmetical operations on vectors are performed *element-wise*, i.e. given two sequences $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_n)$, the operation $\mathbf{a} \circledcirc \mathbf{b}$ results in a vector $(a_1 \circledcirc b_1, \ldots, a_n \circledcirc b_n)$.

```
c(1, 2, 3) * c(0.5, 0.5, 0.5)
## [1] 0.5 1.0 1.5
```

If two operands of different lengths are given (with no loss in generality let us assume $n_1 < n_2$), R conforms them using the so-called *recycling rule* — a shorter vector, say $(a_1, a_2, \ldots, a_{n_1})$, is replicated as many times as required to get the length of the latter, according to the following scheme: $(a_1, a_2, \ldots, a_{n_1}, a_1, a_2, \ldots)$. As a result of the operation, we get a sequence of length n_2.

For example, the following operation is equivalent to the one above.

```
c(1, 2, 3) * 0.5  # the same as above
## [1] 0.5 1.0 1.5
```

Other examples:

```
c(1, 2, 3, 4) + c(1, 0.5)
## [1] 2.0 2.5 4.0 4.5
2^c(0, 1, 2, 3, 4)  # exponentiation
## [1]  1  2  4  8 16
```

In case of vectors of incompatible lengths ($n_2 \mod n_1 \neq 0$), a warning is printed.

The table below lists all the available arithmetic operators.

Operator	Meaning
`-x`	sign inversion
`x + y`	addition
`x - y`	subtraction
`x * y`	multiplication
`x / y`	division
`x ^ y`	exponentiation
`x %/% y`	integer division
`x %% y`	integer division remainder

Some operations may generate special values: infinity (`Inf` keyword) and not-a-number (`NaN` keyword). Inf, NaN

```
1/0
## [1] Inf
0/0
## [1] NaN
-1/Inf
## [1] 0
Inf - Inf
## [1] NaN
NaN + 100
## [1] NaN
-Inf + 1e+11
## [1] -Inf
```

R has, of course, many built-in mathematical functions. When called on a vector, they are applied separately on every its element, i.e. $f((a_1, a_2, \ldots, a_n)) = (f(a_1), f(a_2), \ldots, f(a_n))$.

```
pi # a built-in constant
## [1] 3.142
sin(c(0.0, pi*0.5, pi, pi*1.5, pi*20))
```

```
## [1]  0.000e+00  1.000e+00  1.225e-16 -1.000e+00 -2.449e-15
round(sin(c(0.0, pi*0.5, pi, pi*1.5, pi*20)), 3); # round to 3 signif. digits
## [1]  0  1  0 -1  0
```

Info

`1.225e-16` is a result written in the so-called *scientific notation*. For example, `3.2e-2` is equivalent to $3.2 \cdot 10^{-2}$.

```
0.032
## [1] 0.032
```

Here: 10^{-16} means "almost 0" (we keep in mind the floating-point arithmetic accuracy issues from a course on numerical analysis/methods.

The most commonly used mathematical functions are listed below.

Function	Meaning
`abs(x)`	absolute value
`sqrt(x)`	square root
`cos(x)`	cosine
`sin(x)`	sine
`tan(x)`	tangent
`acos(x)`	arc-cosine
`asin(x)`	arc-sine
`atan(x)`	arc-tangent
`exp(x)`	e^x
`log(x, base = exp(1))`	logarithm of base `base`, the natural log. by default
`log10(x)`	logarithm of base 10
`log2(x)`	logarithm of base 2
`round(x, digits=0)`	rounding
`floor(x)`	floor function
`ceiling(x)`	ceiling function

Info

Note that by writing `log(x, base = exp(1))` we mean that `base` has a *default argument*. If omitted, its value will be established from a pre-defined setting, which is e in this case. Therefore `log(x)` stands for the natural logarithm (which can also be written as `log(x, exp(1))`) and `log(x, 10)` for the logarithm of base 10, $\log_{10}$.

Task

R has an exhaustive and intuitive help system. To get more information on any function, e.g. `sin()`, we call:

```
?sin
```

or, equivalently:

```
help(sin)
```

If we do not know the exact name of the element we are looking for, we may use the simple built-in search engine:

```
??"standard deviation"
```

or:

```
help.search("standard deviation")
```

Moreover, an automatic hinting/completion may be launched in the console by pressing the [TAB] key.

```
> ce      # press [TAB]...
> ceiling # ...and R completes the function name

> cos    # [TAB]

cos   cosh
```

The above implies that R „knows" two functions which names start with `cos`.

There are also many aggregation functions that return a single-number characteristic of a numeric vector. Besides them, auxiliary functions were defined to calculate some common mathematical expressions efficiently.

Function	Meaning	Expression
`sum(x)`	sum of all elements	$r_1 := \sum_{i=1}^{n} x_i$
`prod(x)`	product of all elements	$r_1 := \prod_{i=1}^{n} x_i$
`diff(x)`	differences of neighboring elements	$r_j := x_{j+1} - x_j; \quad j = 1, 2, \ldots, n-1$
`mean(x)`	arithmetic mean	$r_1 := \frac{1}{n} \sum_{i=1}^{n} x_i$
`var(x)`	variance	$r_1 := \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \texttt{mean(x)})^2$
`sd(x)`	standard deviation	$r_1 := \sqrt{\texttt{var(x)}} = \texttt{sqrt(var(x))}$
`sort(x)`	vector ordering	
`rank(x)`	sample ranks	
`min(x)`	minimum	$r_1 := \min_{i=1,2,\ldots,n} x_i$
`max(x)`	maximum	$r_1 := \max_{i=1,2,\ldots,n} x_i$
`cummin(x)`	cumulative minimum	$r_j := \min_{i=1,2,\ldots,j} x_i; \quad j = 1, 2, \ldots, n$
`cummax(x)`	cumulative maximum	$r_j := \max_{i=1,2,\ldots,j} x_i; \quad j = 1, 2, \ldots, n$
`cumsum(x)`	cumulative sum	$r_j := \sum_{i=1}^{j} x_i; \quad j = 1, 2, \ldots, n$
`cumprod(x)`	cumulative product	$r_j := \prod_{i=1}^{j} x_i; \quad j = 1, 2, \ldots, n$

Ex. 1.1. Knowing that $\lim_{n\to\infty} \sqrt{\sum_{i=1}^{n} \frac{6}{i^2}} = \pi$, calculate the approximate value of the circular constant for $n = 100, 1000, 10000$, and 100000.

Solution.
Each value may be calculated straightforwardly. Here we solely use built-in arithmetic operators: the `sum()` and `sqrt()` function:

```
sqrt(sum(6/(1:100)^2))
## [1] 3.132
sqrt(sum(6/(1:1000)^2))
```

```
## [1] 3.141
sqrt(sum(6/(1:10000)^2))
## [1] 3.141
sqrt(sum(6/(1:1e+05)^2))
## [1] 3.142
```

⊡

Details

The above-mentioned functions and operators are optimized to work efficiently on vectors. In such cases it is not needed to use any loops (for example `for`/`while` which is a natural way to solve this type of exercises in `C`). Therefore we are not going to introduce such control-flow structures in this chapter — we do not want you to get into bad habits!

Here is another way to create numeric vectors. It is very easy to generate arithmetic progressions. An arithmetic sequence of common difference equal to 1 or -1 is created by the colon operator (`:`).

```
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
1.5:6
## [1] 1.5 2.5 3.5 4.5 5.5
-1:10  # let us check the priority of the operators
##  [1] -1  0  1  2  3  4  5  6  7  8  9 10
(-1):10  # the same as above
##  [1] -1  0  1  2  3  4  5  6  7  8  9 10
-(1:10)
##  [1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
5:0
## [1] 5 4 3 2 1 0
```

Arithmetic sequences of arbitrary common difference are constructed with the `seq()` function.

```
seq(0, 10, 2); # from 0 to 10 by 2, similarly:
## [1]  0  2  4  6  8 10
seq(11.5, -3, by=-3.7);
## [1] 11.5  7.8  4.1  0.4
```

A sequence of given length, with the difference calculated automatically:

```
seq(0, 1, length = 5)
## [1] 0.00 0.25 0.50 0.75 1.00
```

We are also able to create vectors by *repetition*. Consider the following examples:

```
rep(1, 5)
## [1] 1 1 1 1 1
rep(c(1, 2), 5)
```

```
##  [1] 1 2 1 2 1 2 1 2 1 2
rep(c(1, 2), each = 5)
##  [1] 1 1 1 1 1 2 2 2 2 2
rep(c(1, 2), each = c(5, 4))
## Warning:  first element used of 'each' argument
##  [1] 1 1 1 1 1 2 2 2 2 2
rep(c(1, 2), c(5, 4))
## [1] 1 1 1 1 1 2 2 2 2
```

Task

Study the function usage details by calling `?rep`.

1.2.1.2. Logical vectors

We have four keywords, denoting the logical constants: `TRUE` (or `T` for short) and `FALSE` (or `F`). Logical vectors may be created using `c()` or `rep()`.

```
c(T, F, T)
## [1]  TRUE FALSE  TRUE
rep(FALSE, 3)
## [1] FALSE FALSE FALSE
```

Logical operators

The following logical operators are available. They work in much the same way as arithmetic operators, that is element-wise and conforming to the recycling rule.

Operator	Meaning
`!  x`	negation
`x & y`	logical AND (conjunction)
`x | y`	logical OR (disjunction)
`xor(x, y)`	logical XOR (exclusive disjunction)

Details

Please note that the conjunction and disjunction operators in `C` are written as `&&` and `||`, respectively. In R they may be also used, but their rule of operation is different. They return a single value.

Try to find out how they work.

Some examples:

```
!c(T, F)
## [1] FALSE  TRUE
c(T, F, F, T) | c(T, F, T, F)
## [1]  TRUE FALSE  TRUE  TRUE
c(T, F, F, T) & c(T, F, T, F)
## [1]  TRUE FALSE FALSE FALSE
```

```
xor(c(T, F, F, T), c(T, F, T, F))
## [1] FALSE FALSE  TRUE  TRUE
```

Relational operators may be used to test relations between two operands, e.g. their equality or inequality. They may be applied to numeric vectors. They operate element-wise, returning a logical vector as a result.

Operator	Meaning
`x < y`	less than
`x > y`	more than
`x <= y`	less than or equal to
`x >= y`	more than or equal to
`x == y`	equal to
`x != y`	not equal to

Check those out:

```
2 == 2
## [1] TRUE
(1:3) <= c(1, 3, 5)
## [1] TRUE TRUE TRUE
(1:10) <= c(3, 7)  # recycling rule!
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
(1:10)%%2 == 0
##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
rep(c(TRUE, FALSE), 4) != rep(T, 8)
## [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

1.2.1.3. Character vectors

Another important data type are *character vectors*. Its elements (character strings, labels) may be defined either with double or, equivalently, single quote characters: `"..."` or `'...'`, e.g.:

```
'mary had a little lamb' # a vector of length one
## [1] "mary had a little lamb"
c("OMG", "she really had", "a lamb")
## [1] "OMG"            "she really had" "a lamb"
rep("a", 4);
## [1] "a" "a" "a" "a"
```

Info

The `paste()` function may be used to merge (concatenate) strings into one object. Consecutive strings are separated by a space character by default.

```
paste("Mary", "had", "a lamb");
## [1] "Mary had a lamb"
paste("Mary", "had", "a lamb", sep="!!"); # change default separator
## [1] "Mary!!had!!a lamb"
paste("Mary", "had", "a lamb", sep="");
## [1] "Maryhada lamb"
paste(c("Mary", "had", "a lamb")); # how about that!
## [1] "Mary"   "had"    "a lamb"
paste(c("Mary", "had", "a lamb"),
c("and", "the lamb", "had...")); # element-wise
## [1] "Mary and"      "had the lamb"  "a lamb had..."
paste(c("Mary", "had", "a lamb"), collapse=" ");
## [1] "Mary had a lamb"
paste(c("Mary", "had", "a lamb"),
c("and", "the lamb", "had..."), sep="!", collapse="?");
## [1] "Mary!and?had!the lamb?a lamb!had..."
```

Details

We have some good news for the worshipers of C's `printf`. There is a similar function in R to create character strings. It gives us great amount of control over the formatting issues.

```
sprintf("%e:%7.3f!!%10s, %d", sin(pi), sin(pi), "sin(pi)", floor(100*pi));
## [1] "1.224647e-16:  0.000!!   sin(pi), 314"
cat("Mary", "had", "a lamb"); # consult the manual
## Mary had a lamb
cat(sprintf("%.2f\n%.3f\n%.4f", pi, pi, pi))
## 3.14
## 3.142
## 3.1416
```

Up to now, we have only used vectors consisting either of real numbers, logical values, complex numbers, or character strings. An interesting question arises: are we allowed to mix elements of different types in a single vector? Let's check this out.

```
c(TRUE, 1, 1+1i, "one");
## [1] "TRUE" "1"    "1+1i" "one"
c("one", 1, 1+1i, TRUE); # does the order matter?
## [1] "one"  "1"    "1+1i" "TRUE"
c(TRUE, 1, 1+1i);
## [1] 1+0i 1+0i 1+1i
c(TRUE, 1);
## [1] 1 1
```

1.2.1.4. Variable declaration. Assignment operators

A programming language would be very limited if it did not provide any mechanisms to store objects in the computer's memory. Here such a functionality is provided with *variables*. Each variable is identified by *name*, which starts with a letter. It may consist of any combination of letters, digits and underscore signs (_).

Note

Please bear in mind that R is case-sensitive.

To assign a value to a name (symbol), we use the *assignment operators*: `<-`, `=`, `->`. The first and the second (`<-` is always preferred) need a variable name on their left sides and an expression on the right. The last one — inversely.

```
summy <- 2+2; Summy <- 3+3
abs(summy - Summy - 1) -> dif_f_erence
dif_f_erence
## [1] 3
```

We note that R does not require us to declare a variable before use. Assignment operations do not print out their results. Nevertheless, putting the whole expression in round brackets, forces R to do so:

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

1.2.1.5. Indexing operator

We are certainly curious about captions "`[1]`" printed before the results. In the above example we noted that different ones also appeared. These labels identify indices of vector elements, at which each line starts. Therefore, in the first line we see the elements numbered $1, 2, \ldots, 24$ and in the second $25, 26, \ldots$, etc.

To check the number of elements of a sequence, use `length()`.

```
length(-3:3)
## [1] 7
```

Note

The elements of vectors are numbered from 1. Note that in `C`, the first element has index 0.

Indexing operator

Any vector element of interest may be extracted with the *indexing operator* `[·]`.

```
w <- -10:10
w[3]  # the third element
## [1] -8
w[c(3, 5)]  # the third and the fifth element /so confy!/
## [1] -8 -6
```

```
idx <- 4:10
w[idx]
## [1] -7 -6 -5 -4 -3 -2 -1
```

We may also exclude some arbitrary elements from the vector using negative indices:

```
abc <- 1:10
abc <- abc[-4]
abc
## [1]  1  2  3  5  6  7  8  9 10
abc[-c(3, 6)]
## [1]  1  2  5  6  8  9 10
```

Vectors not only can be indexed with numeric vectors. We may also use logical vectors for that purpose. They define which elements should be included in (`TRUE`) and which should be excluded from (`FALSE`) the resulting sequence. The `[·]` operator's argument requires a parameter of length equal to the vector's length. If a shorter logical vector is given, the recycling rule is applied.

```
x <- 1:10
x[c(T, F, T, F, T, T, F, F, F, F)]
## [1] 1 3 5 6
x[c(T, F)]  # recycling rule
## [1] 1 3 5 7 9

x > 5  # a logical vector as a result
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
x[x > 5]  # wow!
## [1]  6  7  8  9 10

(iLikeRVeryMuch <- x%%3)
##  [1] 1 2 0 1 2 0 1 2 0 1
x[iLikeRVeryMuch == 0]  # or equivalently: x[x %% 3 == 0]
## [1] 3 6 9
```

1.2.1.6. Missing data

There is also a special keyword to mark missing, unknown or *not available* data: `NA`. It is very important when we deal with large data sets that store e.g. the results of experiments or surveys. More on that can be found in the tutorial on Descriptive Statistics. Here are some examples on the usage, search and exclusion of missing data.

```
incompl1 <- c(T, NA, F, NA, T, T)
c(3, 4, 5, NA, 2, 3, 1)  # NA can be used with any type
## [1]  3  4  5 NA  2  3  1
is.na(incompl1)
## [1] FALSE  TRUE FALSE  TRUE FALSE FALSE
incompl1[!is.na(incompl1)]  # similarly:
## [1]  TRUE FALSE  TRUE  TRUE
na.omit(incompl1)
## [1]  TRUE FALSE  TRUE  TRUE
## attr(,"na.action")
```

```
## [1] 2 4
## attr(,"class")
## [1] "omit"
incompl1[is.na(incompl1)] <- FALSE  # replace
incompl1
## [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

Info

We may sometimes be interested in finding whether a value is not-a-number or infinity. The following functions return logical vectors:

```
is.nan(Inf/Inf)
## [1] TRUE
is.finite(-Inf)
## [1] FALSE
is.infinite(c(Inf/Inf, Inf * Inf, 1/Inf, Inf/1))
## [1] FALSE  TRUE FALSE  TRUE
```

1.2.1.7. Type hierarchy. Type conversion

From the above example we may easily conclude that these primary data types form a well defined hierarchy. All elements of any vector are of the same type. When creating a vector by concatenating elements of different types, the resulting type is chosen according to the following prioritization:

1. character,
2. numeric,
3. logical.

Details

Formally, there are separate numeric types — *integer* and *floating point*. However, numbers entered in the console are almost always treated as floating points by default, even if we do not use a decimal point explicitly.

On almost all implementations of R the range of representable integers is restricted to ca. $\pm 2 \cdot 10^9$. The floating point (`double`) type can hold much larger integers exactly.

During this course, without much loss of precision, we will only use the term *numeric type*. The interested reader is referred to the literature for more details and discussion.

Note

It is worth noting that `TRUE` is always converted to the value 1 and `FALSE` to 0. The solution to the example below is therefore a “piece of cake”.

Ex. 1.2. Given a logical vector, check how much truth is in it (i.e. how many `TRUE`s are there).

Solution.
The above note suggests that converting a logical vector to numeric and summing its elements should suffice here then. However, it turns out that `sum()` does the conversion on-the-fly!

```
sum(c(T, F, T, F, F, F, T, T, F, F))
## [1] 4
```

⊡

Details

We may check the type of an object with the `class()` function.

```
class(T);
## [1] "logical"
class(1);
## [1] "numeric"
class(1.0);
## [1] "numeric"
class("1");
## [1] "character"
class(1:5); # that's interesting!
## [1] "integer"
class(c(1,2,3,4,5));
## [1] "numeric"
class(seq(1,5,by=1));
## [1] "numeric"
```

An implicit conversion to a different type may be done via `as.logical()`, `as.numeric()`, `as.character()`, and so on.

```
as.numeric(T);
## [1] 1
as.numeric(F);
## [1] 0
as.logical(1);
## [1] TRUE
as.logical(0);
## [1] FALSE
as.logical(2); # non-zero == TRUE
## [1] TRUE
as.logical(-1);
## [1] TRUE
```

```
as.logical(1i);
## [1] TRUE
as.numeric("4242");
## [1] 4242
as.numeric("Mary");
## Warning:  NAs introduced by coercion
## [1] NA
```

1.2.2. Factors

A *factor* is a special data type inheriting from vector (yes, R is an object-oriented programming language!). It is used to store categorical (qualitative) data. The domain of its elements is a finite (often small), predefined set of *levels* (classes, categories). Each level may be identified with a number and/or a descriptive name.

```
wek <- c("Joe", "Sam", "Jane", "Joe", "Joe", "Jane", "Joe") # only 3 distinct levels
fek <- factor(wek) # convert the character vector to a factor
fek
## [1] Joe  Sam  Jane Joe  Joe  Jane Joe
## Levels: Jane Joe Sam
levels(fek) # a string vector with level names
## [1] "Jane" "Joe"  "Sam"
levels(fek)[1] <- "Michael"; # change level name
fek
## [1] Joe     Sam     Michael Joe     Joe     Michael Joe
## Levels: Michael Joe Sam
(fek2 <- factor(rep(1:5, 2)))
##  [1] 1 2 3 4 5 1 2 3 4 5
## Levels: 1 2 3 4 5
levels(fek2) <- c("A", "B", "C", "D", "E"); fek2
##  [1] A B C D E A B C D E
## Levels: A B C D E
```

Task

Note the result returned by `table()`.

```
table(fek)
## fek
## Michael     Joe     Sam
##       2       4       1
```

`table()` requires a factor argument.

1.2.3. Lists

As we noted earlier, vectors may store only elements of the same type. This is the reason why we call them *atomic* types. Such a restriction does not apply to `lists`.

```
L <- list(1, "string", TRUE, 5:10)
L
## [[1]]
## [1] 1
##
## [[2]]
## [1] "string"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1]  5  6  7  8  9 10
```

We can access individual elements of any list using the double-square-brace operator:

```
L[[1]]
## [1] 1
L[[4]]
## [1]  5  6  7  8  9 10
L[[4]][3]  # the 3rd element of a vector which is the 4th element of the list
## [1] 7
```

Each list element may be named (that is, identified with a name):

```
el <- list(1, greeting = "hi there!", TRUE, values = 5:10)
el
## [[1]]
## [1] 1
##
## $greeting
## [1] "hi there!"
##
## [[3]]
## [1] TRUE
##
## $values
## [1]  5  6  7  8  9 10
```

Therefore the second element of `el` may be accessed by any of the following:

```
el[[2]]
## [1] "hi there!"
el$greeting  # note the dollar-sign ($) operator
## [1] "hi there!"
el[["greeting"]]
## [1] "hi there!"
```

The element names may be modified with the `names()` attribute. It returns the identifiers of all list elements.

```
names(el)
## [1] ""         "greeting" ""         "values"
names(el)[1:2] <- c("one", "greet")
el
## $one
## [1] 1
```

```
##
## $greet
## [1] "hi there!"
##
## [[3]]
## [1] TRUE
##
## $values
## [1]  5  6  7  8  9 10
```

Info

Vector elements may be named in a similar way.

1.2.4. Data frames

A *data frame* is a special kind of list. It stores vectors of the same length. The data are printed as a two-dimensional table.

```
kidnames <- c("Mary", "Mike", "Billy", "Natasha")
age <- c(8, 5, 3, 9)
likeIcecream <- c(T, T, F, T)

chldrn <- data.frame(kidnames, age, likeIcecream)
chldrn
```

```
##   kidnames age likeIcecream
## 1     Mary   8         TRUE
## 2     Mike   5         TRUE
## 3    Billy   3        FALSE
## 4  Natasha   9         TRUE
```

By default, the column names are taken from the argument identifiers. However, they can easily be changed.

```
chldrn <- data.frame(name = kidnames, age, likeIcecream)
chldrn
```

```
##      name age likeIcecream
## 1    Mary   8         TRUE
## 2    Mike   5         TRUE
## 3   Billy   3        FALSE
## 4 Natasha   9         TRUE
```

```
names(chldrn)[3] <- "likesIceCream"
chldrn
```

```
##      name age likesIceCream
## 1    Mary   8          TRUE
## 2    Mike   5          TRUE
## 3   Billy   3         FALSE
## 4 Natasha   9          TRUE
```

Here is how we can access the elements of a data frame:

```
chldrn[1, 1]  # first row, first column
```

```
## [1] Mary
## Levels: Billy Mary Mike Natasha
```

```
chldrn[2:4, c(1, 3)]
```

```
##      name likesIceCream
## 2    Mike          TRUE
## 3   Billy         FALSE
## 4 Natasha          TRUE
```

```
chldrn[1, ]  # first row  (whole range of columns)
```

```
##   name age likesIceCream
## 1 Mary   8          TRUE
```

```
chldrn[, 1]  # first column (whole range of rows)
```

```
## [1] Mary    Mike    Billy   Natasha
## Levels: Billy Mary Mike Natasha
```

```
chldrn[1]  # first column (named)
```

```
##      name
## 1    Mary
## 2    Mike
## 3   Billy
## 4 Natasha
```

```
chldrn$name  # first column (by name)
```

```
## [1] Mary    Mike    Billy   Natasha
## Levels: Billy Mary Mike Natasha
```

Info

The `name` column is of factor type. The character strings are converted automatically to factors during the data frame creation (for convenience). This default behavior may be suppressed by calling:

```
chldrn2 <- data.frame(kidnames, age, likeIcecream, stringsAsFactors = F)
chldrn2[, 1]  # a character vector
```

```
## [1] "Mary"    "Mike"    "Billy"   "Natasha"
```

Task

R function library is fantastic! Search for some interesting functions on your own, e.g. `na.omit()` for removing `NA`s (missing data). `unique()` for eliminating duplicate rows. `duplicated()` for listing duplicated row numbers. `merge()` for merging data frames.

1.2.5. Matrices

Last (but not least) data type to be described here is the `matrix`. Matrices can be created by using the `matrix()` function which needs at least two arguments: the vector of values and a desired size.

```
matrix(1:6, nrow = 2, ncol = 3)  # 2 rows, 3 columns
```

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

```
matrix(1:6, nrow = 2)  # calculate the number of columns automatically
```

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

```
matrix(1:6, nrow = 2, byrow = T)  # row-wise
```

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

A list of the most important matrix operators and functions is given below.

Operation	Meaning
`A %*% B`	matrix multiplication
`det(A)`	determinant
`t(A)`	transposition
`solve(A, b)`	linear equation system solving $\mathbf{Ax} = \mathbf{b}$
`diag(A)`	diagonal
`eigen(A)`	eigenvectors and eigenvalues

Some examples:

```
(A <- matrix(c(2, 0, 0, 2), nrow = 2))
```

```
##      [,1] [,2]
## [1,]    2    0
## [2,]    0    2
```

```
(B <- matrix(1:4, nrow = 2))
```

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

```
A + B  # elementwise
```

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

```
A * B  # elementwise
```

```
##      [,1] [,2]
## [1,]    2    0
## [2,]    0    8
```

```
A %*% B  # matrix multiplication
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

```
det(B)
```

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

```
eigen(B)
```

```
## $values
## [1]  5.3723 -0.3723
##
## $vectors
##         [,1]    [,2]
## [1,] -0.5658 -0.9094
## [2,] -0.8246  0.4160
```

```
t(B)
```

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

1.3. Functions

The most simple function in R may be declared in the following way:

```
f <- function() NULL
```

This function just returns a null-value (e.g. "nothing"). Let's call it.

```
f()
## NULL
```

Some more sophisticated examples:

```
f <- function(x) {
    if (length(x) == 0)
        return(NA)

    y <- sum(x)
    y^2  # equivalent to return(y^2)
}

f(1:10)
## [1] 3025
```

Info

Note that if the end of a function is reached without calling `return()`, the value of the last evaluated expression is returned as result.

Note

Curly braces `{ ... }` are used to group multiple statements into blocks, just like in the `C` programming language.

"Anonymous" functions may also be created. They are useful when they are passed to other functions as arguments.

```
g <- function(x, h, p=1) # p is a default parameter
   h(x^p)

g(1:5, function(y) { y[1] })
## [1] 1
g(1:5, sum, 2)
## [1] 55
```

1.4. Data handling

Data sets to be analyzed may come from many different sources. For example, they can represent the results of:

– an experiment, such as testing drug effects on animal behavior,

– a physical measurement, such as daily flow rate of a river,

– a survey, such as a research on social attitudes towards a political party, etc.

During our course we will not discuss methods of research (experiment) planning and data collecting — that is rather of concern e.g. to the methodology of science. Let us then "just" assume we are given *a* data set and our task is to analyze it.

First, we should *prepare* or preprocess the data set. Among the most important tasks here are:

– organizing data into variables (columns) and records/observations (rows),

– assuring consistency, i.e. all observations grouped into the same variables should be expressed in the same units, date/time format should be the same, etc.

– marking incomplete/incorrect data,

– correct variable (column) naming (simple alphanumeric identifiers, no white spaces in names), etc.

Concerning small-sized data sets, a spreadsheet application is probably the most convenient tool for the first-stage information preprocessing. A data set prepared in an external program can be stored in one of special file formats, and then be imported into R. The most common way is to use the so-called CSV file format, *comma-separated values*, a plain text file with a specific structure.

1.4.1. Importing tabular data into R

To read a CSV file, we use `read.table()` or one of its variants.

Consider the following file, named `example.csv`:

```
# This is a comment
# An exemplary CSV file
Name;Sex;BirthDate;Height
"Mary";F;"1987-05-09";172,3
"Kate";F;"1987-12-31";159,2
"John";M;"1988-04-01";181,9
"Bill";M;"1987-11-12";175,3
"Mike";M;"1986-01-26";192,2
```

The file may be imported into an R data frame by calling:

```
people <- read.table("../datasets/example.csv",
   comment.char="#",     # lines starting with '#' will be treated as comments
   sep=";",              # field separator
   dec=",",              # decimal point
   header=T              # first row contains column names
);
people
```

```
##   Name Sex  BirthDate Height
## 1 Mary   F 1987-05-09  172.3
## 2 Kate   F 1987-12-31  159.2
## 3 John   M 1988-04-01  181.9
## 4 Bill   M 1987-11-12  175.3
## 5 Mike   M 1986-01-26  192.2
```

```
class(people)
```

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

Info

R not only reads local files. We may also open so-called "connections", for example to read a file from a Web server via HTTP.

`read.table()` has many parameters which should be set according to the file structure. However, as some settings appear more often than others, few aliases to the function were created. The table below lists default settings for `read.table()` and its variants.

Function	header	sep	quote	dec	comment.char
read.table()	FALSE	none	" or '	.	#
read.csv()	TRUE	,	"	.	none
read.csv2()	TRUE	;	"	,	none
read.delim()	TRUE	\t	"	.	none
read.delim2()	TRUE	\t	"	,	none

Task

Study the manual page of the `read.table()` function.

Other file formats (we will not use them during this course) may require a special library to be loaded:

```
library("foreign")
```

The most popular functions are: `read.spss()` (SPSS), `read.mtp()` (Minitab), `read.xport()` (SAS). We may also connect to external database servers, e.g. via the `RODBC` package.

1.4.2. Organizing data

1.4.2.1. Date and time conversion

R has built-in formats to store date/time information. The `POSIXct` class represents the (signed) number of seconds since the UNIX Epoch (1 Jan 1970). `POSIXlt` is a named list of vectors representing human-readable details of date/time information. `POSIXt` inherits from both of the classes: it is used to allow operations such as subtraction to mix the two classes.

See the `Sys.time()`, `strptime()`, `format()` functions documentation for more details. See also: `?DateTimeClasses`.

```
strptime(c("2008-12-01", "2008-02-01"), "%Y-%m-%d")  # parse string
## [1] "2008-12-01" "2008-02-01"
Sys.time()  # current time
## [1] "2012-12-09 17:24:38 CET"
Sys.time() - strptime("15:50:21", "%H:%M:%S")  # time difference
## Time difference of 1.572 hours
format(Sys.time(), "%a %b %d %X %Y %Z")  # POSIXlt to string with a fancy format
## [1] "nie gru 09 17:24:38 2012 CET"
```

Info

Moreover, `Date` is a class to represent calendar dates. Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates. See `?Date` for more details.

1.4.2.2. Subsets selection

We may select subsets of data frames either by specifying appropriate arguments to the indexing operator or by using the `subset()` function.

```
people[people$Height > 170, ]
```

```
##   Name Sex  BirthDate Height
## 1 Mary   F 1987-05-09  172.3
## 3 John   M 1988-04-01  181.9
## 4 Bill   M 1987-11-12  175.3
## 5 Mike   M 1986-01-26  192.2
```

```
subset(people, Height > 170)
```

```
##   Name Sex  BirthDate Height
## 1 Mary   F 1987-05-09  172.3
## 3 John   M 1988-04-01  181.9
## 4 Bill   M 1987-11-12  175.3
## 5 Mike   M 1986-01-26  192.2
```

```
subset(people, Height > 170 & Sex == "F", select = -BirthDate)
```

```
##   Name Sex Height
## 1 Mary   F  172.3
```

```
subset(people, Height >= 180 | Sex == "F", select = Sex:BirthDate)
```

```
##   Sex  BirthDate
## 1   F 1987-05-09
## 2   F 1987-12-31
## 3   M 1988-04-01
## 5   M 1986-01-26
```

1.4.3. Other functions

We can bind additional columns to an existing data frame with `cbind()`.

```
IsHappy <- c(T, F, F, T, T)
(people <- cbind(people, IsHappy))
```

```
##   Name Sex  BirthDate Height IsHappy
## 1 Mary   F 1987-05-09  172.3    TRUE
## 2 Kate   F 1987-12-31  159.2   FALSE
## 3 John   M 1988-04-01  181.9   FALSE
## 4 Bill   M 1987-11-12  175.3    TRUE
## 5 Mike   M 1986-01-26  192.2    TRUE
```

`rbind()` is a related function:

```
rbind(people, data.frame(Name = "Sasha", Sex = "F", BirthDate = "1982-01-21",
    Height = 200, IsHappy = F))
```

```
##    Name Sex  BirthDate Height IsHappy
## 1  Mary   F 1987-05-09  172.3    TRUE
## 2  Kate   F 1987-12-31  159.2   FALSE
```

```
## 3  John   M 1988-04-01  181.9   FALSE
## 4  Bill   M 1987-11-12  175.3    TRUE
## 5  Mike   M 1986-01-26  192.2    TRUE
## 6 Sasha   F 1982-01-21  200.0   FALSE
```

Vectors may be sorted with the `sort()` or `order()` functions.

```
(AnOrder <- order(people$Height))  # returns a permutation of indices
## [1] 2 1 4 3 5
people$Height[AnOrder]
## [1] 159.2 172.3 175.3 181.9 192.2
people[AnOrder, ]
##   Name Sex  BirthDate Height IsHappy
## 2 Kate   F 1987-12-31  159.2   FALSE
## 1 Mary   F 1987-05-09  172.3    TRUE
## 4 Bill   M 1987-11-12  175.3    TRUE
## 3 John   M 1988-04-01  181.9   FALSE
## 5 Mike   M 1986-01-26  192.2    TRUE
people[order(people$Height, decreasing = TRUE), ]
##   Name Sex  BirthDate Height IsHappy
## 5 Mike   M 1986-01-26  192.2    TRUE
## 3 John   M 1988-04-01  181.9   FALSE
## 4 Bill   M 1987-11-12  175.3    TRUE
## 1 Mary   F 1987-05-09  172.3    TRUE
## 2 Kate   F 1987-12-31  159.2   FALSE
```

A function may be applied to each row/column of a matrix or a data frame by calling `apply()` (see the manual).

We may apply a function to each group of values given by a unique combination of the levels of certain factors by calling `tapply()`, `aggregate()` or `by()` (see the manual).

A vector may be categorized with `cut()` or `split()`.

Details

To attach a data frame to R's search path, use `attach()`.

```
attach(people)
## The following object(s) are masked _by_ '.GlobalEnv':
##
##     IsHappy
```

Now all the variables are easily accessible. This is very convenient. From now, we do not need to specify the data set name any more when accessing its columns.

```
people$Name  # full path
## [1] Mary Kate John Bill Mike
## Levels: Bill John Kate Mary Mike
Name  # shortened
## [1] Mary Kate John Bill Mike
## Levels: Bill John Kate Mary Mike
```

After finishing our work with the data frame, type:

```
detach(people)
people$Name  # full path only now
## [1] Mary Kate John Bill Mike
## Levels: Bill John Kate Mary Mike
```

1.4.4. Exporting data

To export a data frame to a CSV file, we may consider `write.table()`, `write.csv()`, `write.csv2()`. Their usage is similar to `read.table()`.

1.5. Examples

Ex. 1.3. Add a column `Age` (in years) to the database stored in the `example.csv` file.

Solution.
First, we should convert the birth dates to a `POSIXct` format.

```
bdates <- strptime(people$BirthDate, "%Y-%m-%d");  # parse strings to date
Sys.time() - bdates;   # or:
## Time differences in days
## [1] 9347 9111 9019 9160 9815
## attr(,"tzone")
## [1] ""
difftime(Sys.time(), bdates, units="weeks");
## Time differences in weeks
## [1] 1335 1302 1288 1309 1402
## attr(,"tzone")
## [1] ""
```

We cannot pass `units="years"` as `difftime()` does not know it. Maybe we could solve this task this way:

```
as.POSIXlt(Sys.time())$year  # years since 1900
## [1] 112
as.POSIXlt(Sys.time())$year - as.POSIXlt(bdates)$year
## [1] 25 25 24 25 26
```

Unfortunately, this does not return proper results. So let us try the following.

```
age <- floor(as.numeric(difftime(Sys.time(), bdates, units = "weeks")/52))
(people <- cbind(people, Age = age))
##   Name Sex  BirthDate Height IsHappy Age
## 1 Mary   F 1987-05-09  172.3    TRUE  25
## 2 Kate   F 1987-12-31  159.2   FALSE  25
## 3 John   M 1988-04-01  181.9   FALSE  24
## 4 Bill   M 1987-11-12  175.3    TRUE  25
## 5 Mike   M 1986-01-26  192.2    TRUE  26
```

Is that correct?

⊡

Ex. 1.4. The height data in the `example.csv` file is stored in centimeters. Convert it to feet.

Solution.
The international foot is defined to be equal to 0.3048 meters.

```
people$Height  # [cm]
## [1] 172.3 159.2 181.9 175.3 192.2
people$Height/30.48  # [ft]
## [1] 5.653 5.223 5.968 5.751 6.306
people$Height <- people$Height/30.48
people
##   Name Sex  BirthDate Height IsHappy Age
## 1 Mary   F 1987-05-09  5.653    TRUE  25
## 2 Kate   F 1987-12-31  5.223   FALSE  25
## 3 John   M 1988-04-01  5.968   FALSE  24
## 4 Bill   M 1987-11-12  5.751    TRUE  25
## 5 Mike   M 1986-01-26  6.306    TRUE  26
```

A similar transformation may also be done via the `transform()` function. Imagine we consider the same data set one year later.

```
transform(people, Age = Age + 1, IsHappy = !IsHappy)
##   Name Sex  BirthDate Height IsHappy Age
## 1 Mary   F 1987-05-09  5.653   FALSE  26
## 2 Kate   F 1987-12-31  5.223    TRUE  26
## 3 John   M 1988-04-01  5.968    TRUE  25
## 4 Bill   M 1987-11-12  5.751   FALSE  26
## 5 Mike   M 1986-01-26  6.306   FALSE  27
```

⊡

Ex. 1.5. The (built-in) exemplary data frame `ToothGrowth` is due to an experiment on the effect of vitamin C on guinea pigs' tooth growth. Calculate the mean and variance of teeth lengths in each experimental group.

Solution.
Read the manual on the data set (`?ToothGrowth`) for its detailed description.

```
head(ToothGrowth); # preview a few rows
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
class(ToothGrowth$dose); # check data type
## [1] "numeric"
ToothGrowth$dose <- factor(ToothGrowth$dose); # factor is more convenient
```

Let us list the levels of each categorization variable.

```
levels(ToothGrowth$dose)
## [1] "0.5" "1"   "2"
levels(ToothGrowth$supp)
## [1] "OJ" "VC"
```

The mean tooth length for each supplement type:

```
tapply(ToothGrowth$len, ToothGrowth$supp, mean)
```

```
##    OJ    VC
## 20.66 16.96
```

The variance of lengths for each Vitamin dose:

```
tapply(ToothGrowth$len, ToothGrowth[, 3], var)
```

```
##   0.5     1     2
## 20.25 19.50 14.24
```

And for the combination of supplements and doses:

```
tapply(ToothGrowth[, 1], ToothGrowth[, 2:3], var)
```

```
##     dose
## supp    0.5      1      2
##   OJ 19.889 15.296  7.049
##   VC  7.544  6.327 23.018
```

Details

The mean and variance together may be applied to each level of the grouping variable by specifying a custom function.

```
MeanAndVar <- function(v) {
    c(mean(v), var(v))
}  # return a vector of size 2
tapply(ToothGrowth$len, ToothGrowth$dose, MeanAndVar)
```

```
## $`0.5`
## [1] 10.61 20.25
##
## $`1`
## [1] 19.73 19.50
##
## $`2`
## [1] 26.10 14.24
```

or by specifying anonymous function

```
tapply(ToothGrowth$len, ToothGrowth$dose, simplify = T, function(v) {
    c(MEAN = mean(v), VAR = var(v))
})
```

```
## $`0.5`
##  MEAN   VAR
## 10.61 20.25
##
## $`1`
##  MEAN   VAR
## 19.73 19.50
##
## $`2`
##  MEAN   VAR
## 26.10 14.24
```

Another example, using `by()`:

```
by(ToothGrowth$len, ToothGrowth[, 2:3], simplify = T, function(v) {
    c(MEAN = mean(v), VAR = var(v))
})
```

```
## supp: OJ
## dose: 0.5
##  MEAN   VAR
## 13.23 19.89
## ------------------------------------------------------------
## supp: VC
## dose: 0.5
##  MEAN   VAR
## 7.980 7.544
## ------------------------------------------------------------
## supp: OJ
## dose: 1
## MEAN  VAR
## 22.7 15.3
## ------------------------------------------------------------
## supp: VC
## dose: 1
##   MEAN    VAR
## 16.770  6.327
## ------------------------------------------------------------
## supp: OJ
## dose: 2
##   MEAN    VAR
## 26.060  7.049
## ------------------------------------------------------------
## supp: VC
## dose: 2
##  MEAN   VAR
## 26.14 23.02
```

⊡

Bibliography

[1] M.J. Crawley. *The R Book*. Wiley, 2007.

[2] O. Jones, R. Maillardet, and A. Robinson. *Introduction to Scientific Programming and Simulation Using R*. Chapman & Hall/CRC, 2009.

[3] N.S. Matloff. *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, 2011.

[4] W.N. Venables and B.D. Ripley. *S Programming*. Springer, 2000.