Chapter 2
Fundamentals of R

> When we say that a historian or a linguist is 'innumerate' we mean that he cannot even begin to understand what scientists and mathematicians are talking about
> Oxford English Dictionary, 2nd ed., 1989, s.v. *numeracy*.
> (cited from Keen 2010: 4)

1. Introduction and installation

In this chapter, you will learn about the basics of R that enable you to load, process, and store data as well as perform some simple data processing operations. Thus, this chapter prepares you for the applications in the following chapters. Let us begin with the first step: the installation of R.

1. The main R website is <http://www.r-project.org/>. From there you can go to the CRAN website at <http://cran.r-project.org/mirrors.html>. Click on the mirror Austria, then on the link(s) for your operating system;
2. for Windows you will then click on "base", and then on the link to the setup program to download the relevant setup program; for Mac OS X, you immediately get to a page with a link to a .pkg file; for Linux, you choose your distribution, maybe your distribution version, and then the relevant file(s) or, more conveniently, you may be able to install R and many frequently-used packages using a package manager such as Synaptic or Muon;
3. then, you run the installer;
4. start R by double-clicking on the icon on the desktop, the icon in the start menu, or the icon in the quick launch tool bar.

That's it. You can now start and use R. However, R has more to offer. Since R is an open-source software, there is a lively community of people who have written so-called packages for R. These packages are small additions to R that you can load into R to obtain commands (or functions, as we will later call them) that are not part of the default configuration.

5. In R, enter the following at the console `install.packages()¶` and then choose a mirror; I recommend always using *Austria*;
6. Choose all packages you think you will need; if you have a broadband connection, you could theoretically choose all of them, but that might be a bit of an overkill at this stage. I minimally recommend `amap`, `aod`, `car`, `cluster`, `effects`, `Hmisc`, `lattice`, `qcc`, `plotrix`, `rms`, `rpart`, and `vcd`. (You can also enter, say, `install.packages("car")¶` at the console to install said package and ideally do either with administrator/root rights; in Ubuntu, for example, start R with `sudo R¶`. On Linux systems, you will sometimes also need additional files such as gfortran, which you may need to install separately.)

Next, you should download the files with example files, all the code, exercises, and answer keys onto your hard drive. Ideally, you create one folder that will contain all the files from the book, such as <_sflwr> on your harddrive (for statistics for linguists with R). Then download all files from the companion website of this edition of the book (<http://tinyurl.com/StatForLingWithR>) and save/unzip them into:

– <_sflwr/_inputfiles>: this folder will contain all input files: text files with data for later statistical analysis, spreadsheets providing all files in a compact format, input files for exercises etc.; to unzip these files, you will need the password "hamste_R2";
– <_sflwr/_outputfiles>: this folder will contain output files from Chapters 2 and 5; to unzip these files, you will need the password "squi_R2rel";
– <_sflwr/_scripts>: this folder will contain all files with code from this book as well as the files with exercises and their answer keys; to unzip these files, you will need the password "otte_R2".

(By the way, I am using regular slashes here because you can use those in R, too, and more easily so than backslashes.) The companion website will also provide a file with errata. Lastly, I would recommend that you also get a text editor that has syntax highlighting for R or an IDE (integrated development environment). If you use a text editor, I recommend Notepad++ to Windows users and geany or the use of Notepad++ with Wine to Linux users. The probably best option, however, might be to go with RStudio (<http://www.rstudio.org/>), a truly excellent open source IDE for R, which offers easy editing of R code, sending code from the editor window

to the console with just using Ctrl+ENTER, plot histories, and many other things; you should definitely watch the screencast at RStudio's website.

After all this, you can view all scripts in <_scripts> with syntax-highlighting, which will make it easier for you to understand them. I strongly recommend to write all R scripts that are longer than, say, 2-3 lines in these editors / in the script window of the IDE and then paste them into R because the syntax high-lighting will help you avoid mistakes and you can more easily keep track of all the things you have entered into R.

R is not just a statistics program – it is also a programming language and environment which has at least some superficial similarity to Perl, Python, or Julia. The range of applications is breathtakingly large as R offers the functionality of spreadsheet software, statistics programs, a programming language, database functions etc. This introduction to statistics, however, is largely concerned with

- functions to generate and process simple data structures in R, and
- functions for probability distributions, statistical tests, and graphical evaluation.

We will therefore unfortunately not be able to deal with more complex data structures and many aspects of R as a programming language however interesting these may be. Also, I will not always use the simplest or most elegant way to perform a particular task but the way that is most useful from a pedagogical and methodological perspective (e.g., to highlight commonalities between different functions and approaches). Thus, this book is not really a general introduction to R, and I refer you to the recommendations for further study and the reference section for introductory books to R.

Now we have to address some typographical and other conventions. As already above, websites, folders, and files will be delimited by "<" and ">" as in, say, <_inputfiles/04-1-1-1_tense-aspect.csv>, where the numbering before the underscore refers to the section in which this file is used. Text you are supposed to enter into R is formatted like this `mean(c(1, 2, 3))¶`. This character "¶" instructs you to hit ENTER (I show these characters here because they can be important to show the exact structure of a line and because whitespace makes a big difference in character strings; the code files of course do not include those visibly unless you set your text editor to displaying them). Code will usually be given in grey blocks of several lines like this:

```
> a<-c(1, 2, 3)¶
> mean(a)¶
[1] 2
```

This also means for you: do not enter the two characters `>` . They are only provided for you to easily distinguish your input from R's output. You will also occasionally see lines that begin with "+". These plus signs, which you are not supposed to enter either, begin lines where R is still expecting further input before it begins to execute the function. For example, when you enter `2-¶`, then this is what your R interface will look like:

```
> 2-¶
+
```

R is waiting for you to complete the subtraction. When you enter the number you wish to subtract and press ENTER, then the function will be executed properly.

```
+ 3¶
[1] -1
```

Another example: if you wish to load the package `corpora` into R to access some of the functions that the computational linguists Marco Baroni and Stefan Evert contributed to the community, you can load this package by entering `library(corpora)¶`. (Note: this only works if you installed the package before as explained above.) However, if you forget the closing bracket, R will wait for you to complete the input:

```
> library(corpora¶
+ )¶
>
```

Unfortunately, R will not always be this forgiving. By the way, if you make a mistake in R, you often need to change only one thing in a line. Thus, rather than typing the whole line again, press the cursor-up key to get back to that line you wish to change or execute again; also, you need not move the cursor to the end of the line before pressing ENTER.

Corpus files or tables / data frames will be represented as in Figure 10, where "→" and "¶"denote tab stops and line breaks respectively. Menus, submenus, and commands in submenus in applications are given in italics in double quotes, and hierarchical levels within application menus are indicated with colons. So, if you open a document in, say, LibreOffice Writer,

you do that with what is given here as *File: Open* ...

```
PartOfSp  →  TokenFreq    →  TypeFreq   →  Class¶
ADJ       →  421          →  271        →  open¶
ADV       →  337          →  103        →  open¶
N         →  1411         →  735        →  open¶
CONJ      →  458          →  18         →  closed¶
PREP      →  455          →  37         →  closed¶
```

Figure 10. Representational format of corpus files and data frames

2. Functions and arguments

As you may remember from school, one often does not use numbers, but rather letters to represent variables that 'contain' numbers. In algebra class, for example, you had to find out from two equations such as the following which values *a* and *b* represent (here $a = {}^{23}/_{7}$ and $b = {}^{20}/_{7}$):

$a+2b = 9$ and
$3a-b = 7$

In R, you can solve such problems, too, but R is much more powerful, so variable names such as *a* and *b* can represent huge multidimensional elements or, as we will call them here, *data structures*. In this chapter, we will deal with the data structures that are most important for statistical analyses. Such data structures can either be entered into R at the console or, more commonly, read from files. I will present both means of data entry, but most of the examples below presuppose that the data are available in the form of a tab-delimited text file that has the structure discussed in the previous chapter and was created in a text editor or a spreadsheet software such as LibreOffice Calc. In the following sections, I will explain

- how to create data structures in R;
- how to load data structures into R and save them from R;
- how to edit data structures in R.

One of the most central things to understand about R is how you tell it to do something other than the simple calculations from above. A command in R virtually always consists of two elements: a *function* and, in parentheses, *arguments*. A function is an instruction to do something, and

the arguments to a function represent (i) what the instruction is to be applied to and (ii) how the instruction is to be applied to it. (Arguments can be null, in which case the function name is just followed by opening and closing parentheses.) Let us look at two simple arithmetic functions you know from school. If you want to compute the square root of 5 with R – without simply entering the instruction `5^0.5¶`, that is – you need to know the name of the function as well as how many and which arguments it takes. Well, the name of the function is `sqrt`, and it takes just one argument which R calls `x` by default, namely the figure of which you want the square root. Thus:

```
> sqrt(x=5)¶
[1] 2.236068
```

Note that R just outputs the result, but does not store it. If you want to store a result into a data structure, you must use the assignment operator `<-` (an arrow consisting of a less-than sign and a minus). The simplest way in the present example is to assign a name to the result of `sqrt(5)`. Note: R's handling of names, functions, and arguments is case-sensitive, and you can use letters, numbers, periods, and underscores in names as long as the name begins with a letter or a period (e.g., `my.result` or `my_result` or …):

```
> a<-sqrt(x=5)¶
```

R does not return anything, but the result of `sqrt(5)` has now been assigned to a data structure that is called a vector, which is called `a`. You can test whether the assignment was successful by looking at the content of `a`. One function to do that is `print`, and its minimally required argument is the data structure whose content you want to see, but most of the time, it is enough to simply enter the name of the relevant data structure:

```
> print(a)¶
[1] 2.236068
> a¶
[1] 2.236068
```

Three final comments before we discuss various data structures in more detail. First, R ignores everything in a line after a pound/number sign or hash, which you can use to put comments into your lines (to remind you what that line is doing). Second, the assignment operator can also be used to assign a new value to an existing data structure. For example,

```
> a<-sqrt(x=9) # assign the value of 'sqrt(9)' to a¶
> a # print a¶
[1] 3
> a<-a+2 # assign the value of 'a+2' to a¶
> a # print a¶
[1] 5
```

If you want to delete or clear a data structure, you can use the function `rm` (for *remove*). You can remove just a single data structure by using its name as an argument to `rm`, or you can remove all data structures at once.

```
> rm(a) # remove/clear a¶
> rm(list=ls(all=TRUE)) # clear memory of all data¶
```

Third, it will be very important later on to know that functions have default orders of their arguments and that many functions have default settings for their arguments. The former means that, if you provide arguments in their default order, you don't have to name them. That is, instead of `sqrt(x=9)¶` you could just write `sqrt(9)¶` because the (only) argument `x` is in its 'default position'. The latter means that if you use a function without specifying all required arguments, then R will use default settings, if those are provided by that function. Let us explore this on the basis of the very useful function `sample`. This function generates random or pseudo-random samples of elements and can take up to four arguments:

- `x`: a data structure – typically a vector – containing the elements from which you want to sample;
- `size`: a positive integer giving the size of the sample;
- the assignment `replace=FALSE` (if each element of the vector can only be sampled once, the default setting) or `replace=TRUE` (if the elements of the vector can be sampled multiple times, sampling with replacement);
- `prob`: a vector with the probabilities of each element to be sampled; the default setting is `NULL`, which means that all elements are equally likely to be sampled.

Let us look at a few examples, which will make successively more use of default orders and argument settings. First, you generate a vector with the numbers from 1 to 10 using the function `c` (for *concatenate*); the colon here generates a sequence of integers between the two numbers:

```
> some.data<-c(1:10)¶
```

If you want to sample 5 elements from this vector equiprobably and with replacement, you can enter the following:[11]

```
> sample(x=some.data, size=5, replace=TRUE, prob=NULL)¶
[1] 5 9 9 9 2
```

But if you list the arguments of a function in their standard order (as we do here), then you can leave out their names:

```
> sample(some.data, 5, TRUE, NULL)¶
[1] 3 8 4 1 7
```

Also, `prob=NULL` is the default, so you can leave that out, too:

```
> sample(some.data, 5, TRUE)¶
[1]  2  1  9  9 10
```

With this, you sample 5 elements equiprobably *without* replacement:

```
> sample(some.data, 5, FALSE)¶
[1]  1 10  6  3  8
```

But since `replace=FALSE` is the default, you can leave that out, too:

```
> sample(some.data, 5)¶
[1] 10  5  9  3  6
```

Sometimes, you can even leave out the `size` argument, namely when you just want all elements of the given vector in a random order:

```
> some.data¶
[1]  1  2  3  4  5  6  7  8  9 10
> sample(some.data)¶
[1]  2  4  3 10  9  8  1  6  5  7
```

And if you only want the numbers from 1 to 10 in a random order, you can even do away with the vector `some.data`:

```
> sample(10)¶
[1]  5 10  2  6  1  3  4  9  7  8
```

11. Your results will be different, after all this is *random* sampling.

In extreme cases, the property of default settings may result in function calls without any arguments. Consider the function `q` (for *quit*). This function shuts R down and usually requires three arguments:

- `save`: a character string indicating whether the R workspace should be saved or not or whether the user should be prompted to make that decision (the default);
- `status`: the (numerical) error status to be returned to the operating system, where relevant; the default is 0, indicating 'successful completion';
- `runLast`: a logical value (`TRUE` or `FALSE`), stating whether a function called `Last` should be executed before quitting R; the default is `TRUE`.

Thus, if you want to quit R with these settings, you just enter:

```
> q()¶
```

R will then ask you whether you wish to save the R workspace or not and, when you answered that question, executes the function `Last` (only if one is defined), shuts down R and sends "0" to your operating system.

As you can see, defaults can be a very useful way of minimizing typing effort. However, especially at the beginning, it is probably wise to try to strike a balance between minimizing typing on the one hand and maximizing code transparency on the other. While this may ultimately boil down to a matter of personal preference, I recommend using more explicit code at the beginning in order to be maximally aware of the options your R code uses; you can then shorten your code as you become more proficient.

Recommendation(s) for further study

the functions `?` or `help`, which provide the help file for a function (try `?sample¶` or `help(sample)¶`), and the functions `args` and `formals`, which provide the arguments a function needs, their default settings, and their default order (try `formals(sample)¶` or `args(sample)¶`)

3. Vectors

3.1. Generating vectors

The most basic data structure in R is a vector. Vectors are one-dimensional, sequentially ordered sequences of elements (such as numbers or character

strings (such as words)). While it may not be completely obvious why vectors are important here, we must deal with them in some detail since many other data structures in R can ultimately be understood in terms of vectors. As a matter of fact, we have already used vectors when we computed the square root of 5:

```
> sqrt(5)¶
[1] 2.236068
```

The "[1]" before the result indicates that the first (and, here, only) element printed as the output is element number 1, namely 2.236068. You can test this with R: first, you assign the result of `sqrt(5)` to a data structure.

```
> a<-sqrt(5)¶
```

The function `is.vector` tests whether its argument is a vector or not and returns the result of its test, here R's version of "yes":

```
> is.vector(a)¶
[1] TRUE
```

And the function `length` returns the number of elements of the data structure provided as its argument:

```
> length(a)¶
[1] 1
```

Of course, you can also create vectors that contain character strings – the only difference is that the character strings are put into double quotes:

```
> a.name<-"John"; a.name¶
[1] "John"
```

In this book, we only deal with logical vectors as well as vectors of numbers or character strings. Vectors usually only become interesting when they contain more than one element. You already know the function to create such vectors, `c`, and the arguments it takes are just the elements to be concatenated in the vector, separated by commas. For example:

```
> numbers<-c(1, 2, 3); numbers¶
[1] 1 2 3
```

or

```
> some.names<-c("al", "bill", "chris"); some.names¶
[1] "al"    "bill"  "chris"
```

Note that, since individual numbers or character strings are also vectors (just vectors of length 1), the function `c` can not only combine individual numbers or character strings but also vectors with 2+ elements:

```
> numbers1<-c(1, 2, 3); numbers2<-c(4, 5, 6) # generate two
   vectors¶
> numbers1.and.numbers2<-c(numbers1, numbers2) # combine
   vectors¶
> numbers1.and.numbers2¶
[1] 1 2 3 4 5 6
```

A similar function is `append`, which takes two or three arguments:

- `x`: a vector to which something should be appended;
- `values`: the vector to be appended;
- `after`: the position in the first argument where the elements of the second argument are to be appended; the default setting is at the end.

Thus, with `append`, the above example would look like this:

```
> numbers1.and.numbers2<-append(numbers1, numbers2)¶
> numbers1.and.numbers2¶
[1] 1 2 3 4 5 6
```

An example of how `append` is more typically used is the following, where an existing vector is modified:

```
> evenmore<-c(7, 8)¶
> numbers1.and.numbers2<-append(numbers1.and.numbers2,
   evenmore)¶
> numbers¶
[1] 1 2 3 4 5 6 7 8
```

It is important to note that – unlike arrays in Perl – vectors can only store elements of one data type. For example, a vector can contain numbers *or* character strings, but not really both: if you try to force character strings into a vector together with numbers, R will change the data type of one kind of element to homogenize the kinds of vector elements, and since you

can interpret numbers as characters but not vice versa, R changes the numbers into character strings and then concatenates them into a vector of character strings:

```
> mixture<-c("al", 2, "chris"); mixture¶
[1] "al"    "2"     "chris"
```

and

```
> numbers.num<-c(1, 2, 3); numbers.char<-c("four", "five",
   "six")¶
> nums.and.chars<-c(numbers.num, numbers.char)¶
> nums.and.chars¶
[1] "1"    "2"    "3"    "four" "five" "six"
```

The double quotes around 1, 2, and 3 indicate that these are now understood as character strings, which means that you cannot use them for calculations anymore (unless you change their data type back). We can identify the type of a vector (or the data types of other data structures) with `str` (for "structure") which takes as an argument the name of a data structure:

```
> str(numbers.num)¶
 num [1:3] 1 2 3¶
> str(nums.and.chars)¶
 chr [1:6] "1" "2" "3" "four" "five" "six"
```

The first vector consists of three numerical elements, namely 1, 2, and 3. The second vector consists of the six character strings (from *character*) that are printed.

As you will see later, it is often necessary to create quite long vectors in which (sequences of) elements are repeated. Instead of typing those into R manually, you can use two very useful functions, `rep` and `seq`. In a simple form, the function rep (for *repetition*) takes two arguments: the element(s) to be repeated, and the number of repetitions. To create, say, a vector `x` in which the number sequence from 1 to 3 is repeated four times, you enter:

```
> numbers<-c(1, 2, 3)¶
> x<-rep(numbers, 4)¶
```

or

```
> x<-rep(c(1, 2, 3), 4); x¶
[1] 1 2 3 1 2 3 1 2 3 1 2 3
```

To create a vector in which the numbers from 1 to 3 are individually repeated four times – not in sequence – then you use the argument `each`:

```
> x<-rep(c(1, 2, 3), each=4); x¶
[1] 1 1 1 1 2 2 2 2 3 3 3 3
```

(The same would be true of vectors of character strings.) With whole numbers, you can also often use the `:` as a range operator:

```
> x<-rep(c(1:3), 4)¶
```

The function `seq` (for *sequence*) is used a little differently. In one form, `seq` takes three arguments:

- `from`: the starting point of the sequence;
- `to`: the end point of the sequence;
- `by`: the increment of the sequence.

Thus, instead of entering `numbers<-c(1:3)¶`, you can also write:

```
> numbers<-seq(1, 3, 1)¶
```

Since 1 is the default increment, the following would suffice:

```
> numbers<-seq(1, 3)¶
```

In fact, you can even just write this:

```
> numbers<-seq(3)¶
```

If the numbers in the vector to be created do not increment by 1, you can set the increment to whatever value you need. The following lines generate a vector `x` in which the even numbers between 1 and 10 are repeated six times in sequence. Try it out (and look at `x`):

```
> numbers<-seq(2, 10, 2)¶
> x<-rep(numbers, 6)¶
```

or

```
> x<-rep(seq(2, 10, 2), 6)¶
```

Finally, instead of providing the increment, you can also let R figure out it for you, as when you know how long your sequence should be and just want equal increments everywhere. You can then use the argument `length.out`. The following generates a 7-element sequence from 1 to 10 with equal increments and assigns it to `numbers`:

```
> numbers<-seq(1, 10, length.out=7); numbers¶
[1] 1.0  2.5  4.0  5.5  7.0  8.5 10.0
```

With `c`, `append`, `rep`, and `seq`, even long and complex vectors can often be created fairly easily. Another useful feature is that you can not only name vectors, but also elements of vectors:

```
> numbers<-c(1, 2, 3); names(numbers)<-c("one", "two",
   "three")¶
> numbers¶
  one   two three
    1     2     3
```

Before we turn to loading and saving vectors, let me briefly mention an interactive way to enter vectors into R. If you assign to a data structure just `scan()¶` (for numbers) or `scan(what=character(0))¶` (for character strings), then you can enter the numbers or character strings separated by ENTER until you complete the data entry by pressing ENTER twice:

```
> x<-scan()¶
1: 1¶
2: 2¶
3: 3¶
4:¶¶
Read 3 items
> x¶
[1] 1 2 3
```

Recommendation(s) for further study
the functions `as.numeric` and `as.character` to change the type of vectors

3.2. Loading and saving vectors

Since data for statistical analysis will usually not be entered into R manually, we now turn to reading vectors from files. First a general remark: R can read data of different formats, but we only discuss data saved as text files,

i.e., files that often have the extension: <.txt> or <.csv>. Thus, if the data file has not been created with a text editor but a spreadsheet software such as LibreOffice Calc, then you must first export these data into a text file (with *File: Save As ...* and *Save as type: Text CSV (.csv)*).

A very powerful function to load vector data into R is the function `scan`, which we already used to enter data manually. This function can take many different arguments so you should list arguments with their names. The most important arguments of `scan` for our purposes together with their default settings are as follows:

- `file=""`: the path of the file you want to load as a character string, e.g. `"_inputfiles/02-3-2_vector1.txt"`, but most of the time it is probably easier to just use the function `file.choose()`, which will prompt you to choose the relevant file directly; note, the `file` argument can also be `"clipboard"`;
- `what=""`: the kind of input `scan` is supposed to read. The most important settings are `what=double(0)` (for numbers, the omissible default) and `what=character(0)` (for character strings);
- `sep=""`: the character that separates individual entries in the file. The default setting, `sep=""`, means that any whitespace character will separate entries, i.e. spaces, tabs (represented as `"\t"`), and newlines (represented as `"\n"`). Thus, if you want to read in a text file into a vector such that each line is one element of the vector, you write `sep="\n"`;
- `dec=""`: the decimal point character; `dec="."` is the default; if you want to use a comma instead of the default period, just enter that here as `dec=","`.

To read the file <_inputfiles/02-3-2_vector1.txt>, which contains what is shown in Figure 11, into a vector `x`, you could enter this.

```
1¶
2¶
3¶
4¶
5¶
```

Figure 11. An example file

```
> x<-scan(file=file.choose(), sep="\n")¶
Read 5 items
```

Then you can print out the contents of x:

```
> x¶
[1] 1 2 3 4 5
```

Reading in a file with character strings (like the one in Figure 12) is just as easy; here you just have to tell R that you are reading in a file of character strings and that the character strings are separated by spaces:

```
alpha·bravo·charly·delta·echo¶
```

Figure 12. Another example file

```
> x<-scan(file.choose(), what=character(0), sep=" ")¶
```

You get:

```
> x¶
[1] "alpha"  "bravo"  "charly" "delta"  "echo"
```

Now, how do you save vectors into files. The required function – basically the reverse of `scan` – is `cat` and it takes very similar arguments:

- the vector(s) to be saved;
- `file=""`: the path to the file into which the vector is to be saved or again just `file.choose()`);
- `sep=""`: the character that separates the elements of the vector from each other: `sep=""` or `sep=" "` for spaces (the default), `sep="\t"` for tabs, `sep="\n"` for newlines;
- `append=TRUE` or `append=FALSE` (the default): if the output file already exists and you set `append=TRUE`, then the output will be appended to the output file, otherwise the output will overwrite the existing file.

Thus, to append two names to the vector x and then save it under some other name, you can enter the following:

```
> x<-append(x, c("foxtrot", "golf"))¶
> cat(x, file=file.choose())¶
```

Recommendation(s) for further study
the function `write`, `save`, and `dput` to save vectors (or other structures)

3.3. Editing vectors

Now that you can generate, load, and save vectors, we must deal with how you can edit them. The functions we will be concerned with allow you to access particular parts of vectors to output them, to use them in other functions, or to change them. First, a few functions to edit numerical vectors. One such function is `round`. Its first argument is the vector with numbers to be rounded, its second the desired number of decimal places. (Note, R rounds according to an IEEE standard: 3.55 does not become 3.6, but 3.5.)

```
> a<-seq(3.4, 3.6, 0.05); a¶
[1] 3.40 3.45 3.50 3.55 3.60
> round(a, 1)¶
[1] 3.4 3.4 3.5 3.5 3.6
```

The function `floor` returns the largest integers not greater than the corresponding elements of the vector provided, `ceiling` returns the smallest integers not less than the corresponding elements of the vector provided, and `trunc` simply truncates the elements toward 0:

```
> floor(c(-1.8, 1.8))¶
[1] -2  1
> ceiling(c(-1.8, 1.8))¶
[1] -1  2
> trunc(c(-1.8, 1.8))¶
[1] -1  1
```

The most important way to access parts of a vector (or other data structures) in R involves *subsetting* with square brackets. In the simplest form, this is how you access an individual vector element (here, the third):

```
> x<-c("a", "b", "c", "d", "e")¶
> x[3]¶
[1] "c"
```

Since you already know how flexible R is with vectors, the following uses of square brackets should not come as big surprises:

```
> y<-3; x[y]¶
[1] "c"
> z<-c(1, 3); x[z]¶
[1] "a" "c"
> z<-c(1:3); x[z]¶
[1] "a" "b" "c"
```

With negative numbers, you can leave out elements:

```
> x[-2]¶
[1] "a" "c" "d" "e"
```

However, there are many more powerful ways to access parts of vectors. For example, you can let R determine which elements of a vector fulfill a certain condition. One way is to present R with a *logical expression*:

```
> x=="d"¶
[1] FALSE FALSE FALSE  TRUE FALSE
```

This means, R checks for each element of `x` whether it is "d" or not and returns its findings. The only thing requiring a little attention here is that the logical expression uses two equal signs, which distinguishes logical expressions from assignments such as `file=""`. Other logical operators are:

&	and	|	or
>	greater than	<	less than
>=	greater than or equal to	<=	less than or equal to
!	not	!=	not equal to

Here are some examples:

```
> x<-c(10:1)¶
> x¶
[1] 10  9  8  7  6  5  4  3  2  1
> x==4¶
[1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
   FALSE
> x<=7¶
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
   TRUE
> x!=8¶
[1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
   TRUE
> (x>8 | x<3)¶
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
   TRUE
```

Since `TRUE` and `FALSE` in R correspond to 1 and 0, you can easily determine how often a particular logical expression is true in a vector:

```
> sum(x==4)¶
[1] 1
```

```
> sum(x>8 | x<3)¶
[1] 4
```

The very useful function `table` counts how often vector elements (or combinations of vector elements) occur. For example, with `table` we can immediately determine how many elements of `x` are greater than 8 or less than 3. (Note: `table` ignores missing data – if you want to count those, too, you must write `table(…, exclude=NULL)`.)

```
> table(x>8 | x<3)¶
FALSE  TRUE
    6     4
```

It is, however, obvious that the above examples are not particularly elegant ways to identify the position(s) of elements. However many elements of `x` fulfill a logical condition, you always get 10 logical values (one for each element of `x`) and must locate the `TRUE`s by hand – what do you do when a vector contains 10,000 elements? Another function can do that for you, though. This function is `which`, and it takes a logical expression of the type discussed above:

```
> which(x==4) # which elements of x are 4?¶
[1] 7
```

As you can see, this function looks nearly like English: you ask R "which element of `x` is 4?", and you get the response 'the seventh'. The following examples are similar to the ones above but now use `which`:

```
> which(x<=7) which elements of x are <= 7?¶
[1]  4  5  6  7  8  9 10
> which(x!=8) # which elements of x are not 8?¶
[1]  1  2  4  5  6  7  8  9 10
> which(x>8 | x<3) which elements of x are >8 or <3?¶
[1]  1  2  9 10
```

It should go without saying that you can assign such results to data structures, i.e. vectors:

```
> y<-which(x>8 | x<3); y¶
[1]  1  2  9 10
```

Note: do not confuse the *position of an element* in a vector with the *element* of the vector. The function `which(x==4)¶` does not return the element

4, but the position of the element 4 in x, which is 7; and the same is true for the other examples. You can probably guess how you can now get the elements themselves and not just their positions. You only need to remember that R uses vectors. The data structure you just called y is also a vector:

```
> is.vector(y)¶
[1] TRUE
```

Above, you saw that you can use vectors in square brackets to access parts of a vector. Thus, when you have a vector x and do not just want to know where to find numbers which are larger than 8 or smaller than 3, but also which numbers these are, you first use which and then square brackets, or you immediately combine these two steps:

```
> y<-which(x>8 | x<3)¶
> x[y]¶
[1] 10  9  2  1
> x[which(x>8 | x<3)]¶
[1] 10  9  2  1
```

Or you use this, which uses the fact that, when you subset with a logical vector of TRUEs and FALSEs, R returns the elements subset by TRUEs:

```
> x[x>8 | x<3]¶
[1] 10  9  2  1
```

You use a similar approach to see how often a logical expression is true:

```
> length(which(x>8 | x<3))¶
[1] 4
```

Sometimes you may want to test for several elements at once (e.g., the numbers 1, 6, and 11), which which can't do, but you can use the very useful operator %in%:

```
> c(1, 6, 11) %in% x¶
[1]  TRUE  TRUE FALSE
```

The output of %in% is a logical vector which says for each element of the vector before %in% whether it occurs in the vector after %in%. If you also would like to know the exact position of the first (!) occurrence of each of the elements of the first vector in the second, you can use match:

```
> match(c(1, 6, 11), x)¶
[1] 10  5 NA
```

That is to say, the first element of the first vector – the 1 – occurs the first (and only) time at the tenth position of `x`; the second element of the first vector – the 6 – occurs the first (and only) time at the fifth position of `x`; the last element of the first vector – the 11 – does not occur in `x`.

I hope it becomes obvious that the fact that much of what R does involves vectors is a big strength of R. Since nearly everything we have done so far is based on vectors (often of length 1), you can use functions flexibly and even embed them into each other freely. For example, now that you have seen how to access parts of vectors, you can also change those. Maybe you would like to change the values of `x` that are greater than 8 into 12:

```
> x # show x again¶
 [1] 10  9  8  7  6  5  4  3  2  1
> y<-which(x>8)¶
> x[y]<-12¶
> x¶
[1] 12 12  8  7  6  5  4  3  2  1
```

As you can see, since you want to replace more than one element in `x` but provide only one replacement (12), R recycles the replacement as often as needed (cf. below for more on that feature). This is a shorter way to do the same thing:

```
> x<-10:1¶
> x[which(x>8)]<-12¶
> x¶
 [1] 12 12  8  7  6  5  4  3  2  1
```

And this one is even shorter:

```
> x<-10:1¶
> x[x>8]<-12¶
> x¶
 [1] 12 12  8  7  6  5  4  3  2  1
```

R also offers several set-theoretical functions – `setdiff`, `intersect`, and `union` – which take two vectors as arguments. The function `setdiff` returns the elements of the first vector that are not in the second vector. The function `intersect` returns the elements of the first vector that are also in the second vector. And the function `union` returns all elements that occur in at least one of the two vectors.

```
> x<-c(10:1); y<-c(2, 5, 9, 12)¶
> setdiff(x, y)¶
[1] 10  8  7  6  4  3  1
> setdiff(y, x)¶
[1] 12
> intersect(x, y)¶
[1] 9 5 2
> intersect(y, x)¶
[1] 2 5 9
> union(x, y)¶
[1] 10  9  8  7  6  5  4  3  2  1 12
> union(y, x)¶
[1]  2  5  9 12 10  8  7  6  4  3  1
```

Another useful function is `unique`, which can be explained particularly easily to linguists: `unique` goes through all the elements of a vector (tokens) and returns all elements that occur at least once (types).

```
> x<-c(1, 2, 3, 2, 3, 4, 3, 4, 5)¶
> unique(x)
[1] 1 2 3 4 5
```

In R you can also very easily apply a mathematical function or operation to many or all elements of a numerical vector. Mathematical operations that are applied to a vector are applied to all elements of the vector:

```
> x<-c(10:1)¶
> x¶
[1] 10  9  8  7  6  5  4  3  2  1
> y<-x+2¶
> y¶
[1] 12 11 10  9  8  7  6  5  4  3
```

If you add two vectors (or multiply them with each other, or …), three different things can happen. First, if the vectors are equally long, the operation is applied to all pairs of corresponding vector elements:

```
> x<-c(2, 3, 4); y<-c(5, 6, 7)¶
> x*y¶
[1] 10 18 28
```

Second, the vectors are not equally long, but the length of the longer vector can be divided by the length of the shorter vector without a remainder. Then, the shorter vector will again be recycled as often as is needed to perform the operation in a pairwise fashion; as you saw above, often the length of the shorter vector is 1.

```
> x<-c(2, 3, 4, 5, 6, 7); y<-c(8, 9)¶
> x*y¶
[1] 16 27 32 45 48 63
```

Third, the vectors are not equally long and the length of the longer vector is not a multiple of the length of the shorter vector. In such cases, R will recycle the shorter vector as necessary, but will also issue a warning:

```
> x<-c(2, 3, 4, 5, 6); y<-c(8, 9)¶
> x*y¶
[1] 16 27 32 45 48
Warning message:
In x * y : longer object length is not a multiple of shorter
   object length
```

Finally, two functions to change the ordering of elements of vectors. The first of these functions is called `sort`, and its most important argument is of course the vector whose elements are to be sorted; another important argument defines the sorting style: `decreasing=FALSE` (the default) or `decreasing=TRUE`.

```
> x<-c(1, 3, 5, 7, 9, 2, 4, 6, 8, 10)¶
> y<-sort(x)¶
> z<-sort(x, decreasing=TRUE)¶
> y; z¶
[1]  1  2  3  4  5  6  7  8  9 10
[1] 10  9  8  7  6  5  4  3  2  1
```

The second function is `order`. It takes one or more vectors as arguments as well as the argument `decreasing=…` – but it returns something that may not be immediately obvious. Can you see what `order` does?

```
> z<-c("a", "c", "e", "d", "b")¶
> order(z, decreasing=FALSE)¶
[1] 1 5 2 4 3
```

THINK
BREAK

The output of `order` when applied to a vector `z` is a vector which tells you in which order to put the elements of `z` to sort them as specified. Let us clarify this rather opaque characterization: If you want to sort the values of

`z` in increasing order, you first have to take `z`'s first value. Thus, the first value of `order(z, decreasing=FALSE)¶` is 1. The next value you have to take is the fifth value of `z`. The next value you take is the second value of `z`, etc. (If you provide `order` with more than one vector, additional vectors are used to break ties.) As we will see below, this function will turn out to be useful when applied to data frames.

Recommendations for further study

- the functions `any` and `all` to test whether any or all elements of a vector fulfill a particular condition
- the function `abs` to obtain the absolute values of a numerical vector
- the functions `min` and `max` to obtain the minimum and the maximum values of numeric vectors respectively

4. Factors

At a superficial glance at least, factors are similar to vectors of character strings. Apart from the few brief remarks in this section, they will mainly be useful when we read in data frames and want R to recognize that some of their columns are nominal or categorical variables.

4.1. Generating factors

As I just mentioned, factors are mainly used to code nominal or categorical variables, i.e. in situations where a variable has two or more (but usually not very many) qualitatively different levels. The simplest way to create a factor is to generate a vector and then change it into a factor using the function `factor`. That function usually takes one or two arguments. The first is mostly the vector you wish to change into a factor. The second argument is `levels=…` and will be discussed in more detail in Section 2.4.3 below.

```
> rm(list=ls(all=TRUE))¶
> x<-c(rep("male", 5), rep("female", 5))¶
> y<-factor(x); y¶
 [1] male   male   male   male   male   female female female
   female female
Levels: female male
> is.factor(y)
 [1] TRUE
```

When you output a factor, you can see one difference between factors and vectors because the output includes a by default alphabetically sorted list of all levels of that factor.

One other very useful way in which one sometimes generates factors is based on the function `cut`. In its simplest implementation, it takes a numeric vector as its first argument (`x`) and a number of intervals as its second (`breaks`), and then it divides `x` into `breaks` intervals:

```
> cut(1:9, 3)¶
[1] (0.992,3.66] (0.992,3.66] (0.992,3.66] (3.66,6.34]
   (3.66,6.34]  (3.66,6.34]  (6.34,9.01]  (6.34,9.01]
   (6.34,9.01]
Levels: (0.992,3.66] (3.66,6.34] (6.34,9.01]
```

As you can see, the vector with the numbers from 1 to 9 has now been recoded as a factor with three levels that provide the intervals R used for cutting up the numeric vector.

- $0.992 <$ interval/level 1 ≤ 3.66;
- $3.66 <$ interval/level 1 ≤ 6.34;
- $6.34 <$ interval/level 1 ≤ 9.01.

This function has another way of using `breaks` and some other useful arguments so you should explore those in more detail: `?cut`.

4.2. Loading and saving factors

We do not really need to discuss how you load factors – you do it in the same way as you load vectors, and then you convert the loaded vector into a factor as illustrated above. Saving a factor, however, is a little different. Imagine you have the following factor `a`.

```
> a<-factor(c("alpha", "charly", "bravo")); a¶
[1] alpha  charly bravo
Levels: alpha bravo charly
```

If you now try to save this factor into a file as you would with a vector,

```
> cat(a, sep="\n", file=file.choose())¶
```

your output file will look like Figure 13.

```
1¶
3¶
2¶
```

Figure 13. Another example file

This is because R represents factors internally in the form of numbers (which represent the factor levels), and therefore R also only outputs these numbers into a file. Since you want the words, however, you simply force R to treat the factor as a vector, which will produce the desired result.

```
> cat(as.vector(a), sep="\n", file=file.choose())¶
```

4.3. Editing factors

Editing factors is similar to editing vectors, but a bit more cumbersome when you want to introduce new levels. Let's create a factor `x`:

```
> x<-factor(rep(c("long", "intmed", "short"), 1:3)); x¶
[1] long   intmed intmed short  short  short
Levels: intmed long short
```

Note how the alphabetical ordering of the levels is not particularly useful since it does not coincide with an ascending or descending order of the meaning of the levels. The easiest thing you may have to do is change the first level from the alphabetically first level to another one (which will be important in Chapters 4 and 5). For example, you may want to make `short` the first level. For that, you can use the function `relevel`, which requires the factor to be changed and the new reference level:

```
> x<-relevel(x, "short"); x¶
[1] long   intmed intmed short  short  short
Levels: short intmed long
```

As you can see, the factor content per se has not changed, only the order in which the levels are listed and now we have a nice ordering of the levels.

If you want to change the order of levels more substantively – for instance reversing their order – you can use the function `factor` again and assign the levels in the desired way. Again, the content of the factor is the same, but the order of the levels is now reversed.

```
> x<-factor(x, levels=levels(x)[3:1]); x¶
[1] long   intmed intmed short  short  short
Levels: long intmed short
```

Now, what about changing the content of factors? You may want to just chang the name of a level in that factor to something else. You can do that by just setting a new level, e.g., changing `intmed` to `intermed`:

```
> levels(x)[2]<-"intermed"; x¶
[1] long     intermed intermed short    short    short
Levels: long intermed short
```

Then, you may wish to change a particular element to some other level that is already attested in the factor. In that case, you can treat factors as you would vectors:

```
> x[3]<-"short"; x¶
[1] long     intermed short    short    short    short
Levels: long intermed short
```

A difficulty arises when you want to assign a brand new level:

```
> x[6]<-"supershort"¶
Warning message:
In `[<-.factor`(`*tmp*`, 6, value = "supershort") :
  invalid factor level, NAs generated
> x¶
[1] long     intermed short    short    short    <NA>
Levels: long intermed short
```

Thus, if you want to assign a new level, you must proceed differently: Let's re-create `x` and then first define the new (fourth) level with `levels`:

```
x<-factor(rep(c("long", "intermed", "short"), c(1, 1, 4)),
   levels=c("long", "intermed", "short"))¶
> x<-factor(x, levels=c(levels(x), "supershort")); x¶
[1] long     intermed short    short    short    short
Levels: long intermed short supershort
```

Note: the factor content has not changed yet, you only have one more level than before. This also illustrates a factor can have levels that are not attested in its content. However, now that `x` has all the levels you need, you can proceed as above and assign the new value as you would with a vector:

```
> x[6]<-"supershort"; x¶
[1] long       intermed   short      short      short
   supershort
Levels: long intermed short supershort
```

Now, let's just assume you changed your mind and changed the sixth data point back to `short`:

```
> x[6]<-"short"; x¶
 [1] long     intermed short    short    short    short
Levels: long intermed short supershort
```

Now it is of course not nice to have this level `supershort` listed in the levels if it is not really attested especially because later we will use functions that would return output for these levels even if they are unattested. Thus, let us get rid of this level. Thankfully, that is easy: you can just apply the function `factor` again, which will then drop unused levels:

```
> x<-factor(x); x # also see ?droplevels¶
[1] long     intermed short    short    short    short
Levels: long intermed short
```

Sometimes one wants to conflate factor levels, e.g. to test whether all levels of a factor that corpus data were annotated for are actually required. Let's assume, you decide that you really only want to distinguish 'short' from 'not short'. This is how you change the levels and the factor accordingly, essentially by overwriting the first two levels with the new level.

```
> levels(x)<-c("not_short", "not_short", "short"); x¶
[1] not_short not_short short     short     short     short
Levels: not_short short
```

Finally, as I mentioned above, R stores factors as numbers and there are situations (esp. in the context of plotting, see Ch. 5) where it is useful to have access to these numbers. The function `as.numeric` provides these:

```
> as.numeric(x)¶
[1] 1 1 2 2 2 2
```

Recommendation(s) for further study

– the function `is.factor` to test whether a data structure is a factor
– the functions `gl` and `reorder` to create factors and reorder levels

5. Data frames

The data structure that is most relevant to nearly all statistical methods in this book is the data frame. The data frame, basically what we would colloquially call a table, is actually only a specific type of another data structure, a list, but since data frames are the single most frequent input format for statistical analyses (within R, but also for other statistical programs and of course spreadsheet software), we will concentrate only on data frames per se and disregard lists for now.

5.1. Generating data frames

Given the centrality of vectors in R, you can generate data frames easily from vectors (and factors). Imagine you collected three different kinds of information for five parts of speech and wanted to generate the data frame in Figure 14:

- the variable TOKENFREQUENCY, i.e. the frequency of words of a particular part of speech in a corpus X;
- the variable TYPEFREQUENCY, i.e. the number of different words of a particular part of speech in the corpus X;
- the variable CLASS, which represents whether the part of speech is from the group of open-class words or closed-class words.

```
POS     →     TOKENFREQ     →     TYPEFREQ     →     CLASS¶
adj      →    421          →     271          →     open¶
adv      →    337          →     103          →     open¶
n       →     1411        →      735          →     open¶
conj     →    458          →     18           →     closed¶
prep     →    455          →     37           →     closed¶
```

Figure 14. An example data frame

Step 1: you generate four vectors, one for each column:

```
> rm(list=ls(all=TRUE))¶
> POS<-c("adj", "adv", "n", "conj", "prep")¶
> TOKENFREQ<-c(421, 337, 1411, 458, 455)¶
> TYPEFREQ<-c(271, 103, 735, 18, 37)¶
> CLASS<-c("open", "open", "open", "closed", "closed")¶
```

Step 2: The first row in the desired table does not contain data points but the header with the column names. You must now decide whether the first column contains data points or also 'just' the names of the rows. In the first case, you can just create your data frame with the function `data.frame`, which takes as arguments the relevant vectors; the order of vectors determines the order of columns. Now you can look at the data frame.

```
> x<-data.frame(POS, TOKENFREQ, TYPEFREQ, CLASS)¶
> x¶
   POS TOKENFREQ TYPEFREQ  CLASS
1  adj       421      271   open
2  adv       337      103   open
3    n      1411      735   open
4 conj       458       18 closed
5 prep       455       37 closed
> str(x)¶
'data.frame': 5 obs. of  4 variables:
 $ POS      : Factor w/ 5 levels "adj","adv",..: 1 2 4 3 5
 $ TOKENFREQ: num  421 337 1411 458 455
 $ TYPEFREQ : num  271 103 735 18 37
 $ CLASS    : Factor w/ 2 levels "closed","open": 2 2 2 1 1
```

Within the data frame, R has changed the vectors of character strings into factors and represents them with numbers internally (e.g., `closed` is 1 and `open` is 2). It is very important in this connection that R only changes variables into factors when they contain character strings (and not just numbers). If you have a data frame in which nominal or categorical variables are coded with numbers, then R will neither know nor guess that these are factors and will treat the variables as numeric and thus as interval/ratio variables in statistical analyses. Thus, you should either use meaningful character strings as factor levels in the first place (as recommended in Chapter 1 anyway) or must characterize the relevant variable(s) as factors at the point of time you create the data frame: `factor(vectorname)`. Also, you did not define row names, so R automatically numbers the rows. If you want to use the parts of speech as row names, you need to say so explicitly:

```
> x<-data.frame(TOKENFREQ, TYPEFREQ, CLASS, row.names=POS)
> x¶
     TOKENFREQ TYPEFREQ  CLASS
adj        421      271   open
adv        337      103   open
n         1411      735   open
conj       458       18 closed
prep       455       37 closed
> str(x)¶
'data.frame': 5 obs. of  3 variables:
 $ TOKENFREQ: num  421 337 1411 458 455
```

```
 $ TYPEFREQ : num  271 103 735 18 37
 $ CLASS    : Factor w/ 2 levels "closed","open": 2 2 2 1 1
```

As you can see, there are now only three variables left because `POS` now functions as row names. Note that this is only possible when the column with the row names contains no element twice.

A second way of creating data frames that is much less flexible, but extremely important for Chapter 5 involves the function `expand.grid`. In its simplest use, the function takes several vectors or factors as arguments and returns a data frame the rows of which contain all possible combinations of vector elements and factor levels. Sounds complicated but is very easy to understand from this example and we will use this many times:

```
> expand.grid(COLUMN1=c("a", "b"), COLUMN2=1:3)¶
  COLUMN1 COLUMN2
1       a       1
2       b       1
3       a       2
4       b       2
5       a       3
6       b       3
```

5.2. Loading and saving data frames

While you can generate data frames as shown above, this is certainly not the usual way in which data frames are entered into R. Typically, you will read in files that were created with a spreadsheet software. If you create a table in, say LibreOffice Calc and want to work on it within R, then you should first save it as a comma-separated text file. There are two ways to do this. Either you copy the whole file into the clipboard, paste it into a text editor (e.g., geany or Notepad++), and then save it as a tab-delimited text file, or you save it directly out of the spreadsheet software as a CSV file (as mentioned above with *File: Save As ...* and *Save as type: Text CSV (.csv)*; then you choose tabs as field delimiter and no text delimiter, and don't forget to provide the file extension. To load this file into R, you use the function `read.table` and some of its arguments:

- `file="…"`: the path to the text file with the table (on Windows PCs you can use `choose.files()` here, too; if the file is still in the clipboard, you can also write `file="clipboard"`;
- `header=TRUE`: an indicator of whether the first row of the file contains

column headers (which it should always have) or `header=FALSE` (the default);
- `sep=""`: between the double quotes you put the single character that delimits columns; the default `sep=""` means space or tab, but usually you should set `sep="\t"` so that you can use spaces in cells of the table;
- `dec="."` or `dec=","`: the decimal separator;
- `row.names=…`, where … is the number of the column containing the row names;
- `quote=…`: the default is that quotes are marked with single or double quotes, but you should nearly always set `quote=""`;
- `comment.char=…`: the default is that comments are separated by "#", but we will always set `comment.char=""`.

Thus, if you want to read in the above table from the file <_inputfiles/02-5-2_dataframe1.csv> – once without row names and once with row names – then this is what you could type:

```
> a1<-read.table(file.choose(), header=TRUE, sep="\t",
   quote="", comment.char="") # R numbers rows¶
```

or

```
> a2<-read.table(file.choose(), header=TRUE, sep="\t",
   quote="", comment.char="", row.names=1) # row names¶
```

By entering `a1¶` or `str(a1)¶` (same with `a2`), you can check whether the data frames have been loaded correctly.

While the above is the most explicit and most general way to load all sorts of different data frames, when you have set up your data as recommended above, you can often use a shorter version with `read.delim`:, which has `header=TRUE` and `sep="\t"` as defaults and should, therefore, work most of the time:

```
> a3<-read.delim(file.choose())¶
```

If you want to save a data frame from R, then you can use `write.table`. Its most important arguments are:

- `x`: the data frame you want to save;
- `file`: the path to the file into which you wish to save the data frame;

typically, using `file.choose()` is easiest;
- `append=FALSE` (the default) or `append=TRUE`: the former generates or overwrites the defined file, the latter appends the data frame to that file;
- `quote=TRUE` (the default) or `quote=FALSE`: the former prints factor levels with double quotes; the latter prints them without quotes;
- `sep=""`: between the double quotes you put the single character that delimits columns; the default `" "` means a space, what you should use is `"\t"`, i.e. tabs;
- `eol="\n"`: between the double quotes you put the single character that separates lines from each other (eol for *end of line*); the default `"\n"` means newline;
- `dec="."` (the default): the decimal separator;
- `row.names=TRUE` (the default) or `row.names=FALSE`: whether you want row names or not;
- `col.names=TRUE` (the default) or `col.names=FALSE`: whether you want column names or not.

Given these default settings and under the assumption that your operating system uses an English locale, you would save data frames as follows:

```
> write.table(a1, file.choose(), quote=FALSE, sep="\t",
   col.names=NA)¶
```

5.3. Editing data frames

In this section, we will discuss how you can access parts of data frames and then how you can edit and change data frames.

Further below, we will discuss many examples in which you have to access individual columns or variables of data frames. You can do this in several ways. The first of these you may have already guessed from looking at how a data frame is shown in R. If you load a data frame with column names and use `str` to look at the structure of the data frame, then you see that the column names are preceded by a "$". You can use this syntax to access columns of data frames, as in this example using the file <_inputfiles/02-5-3_dataframe.csv>.

```
> rm(list=ls(all=TRUE))¶
> a<-read.delim(file.choose())¶
> a¶
   POS TOKENFREQ TYPEFREQ  CLASS
```

```
1  adj       421      271   open
2  adv       337      103   open
3    n      1411      735   open
4 conj       458       18 closed
5 prep       455       37 closed
> a$TOKENFREQ¶
[1]  421  337 1411  458  455
> a$CLASS¶
[1] open   open   open   closed closed
Levels: closed open
```

You can now use these just like any other vector or factor. For example, the following line computes token/type ratios of the parts of speech:

```
> ratio<-a$TOKENFREQ/a$TYPEFREQ; ratio¶
[1]  1.553506  3.271845  1.919728 25.444444 12.297297
```

You can also use indices in square brackets for subsetting. Vectors and factors as discussed above are one-dimensional structures, but R allows you to specify arbitrarily complex data structures. With two-dimensional data structures, you can also use square brackets, but now you must of course provide values for both dimensions to identify one or several data points – just like in a two-dimensional coordinate system. This is very simple and the only thing you need to memorize is the order of the values – rows, then columns – and that the two values are separated by a comma. Here are some examples:

```
> a[2,3]¶
[1] 103
> a[2,]¶
  POS TOKENFREQ TYPEFREQ CLASS
2 adv       337      103  open
> a[,3]¶
[1] 271 103 735  18  37
> a[2:3,4]¶
[1] open open
Levels: closed open
> a[2:3,3:4]¶
   TYPEFREQ CLASS
2       103  open
3       735  open
```

Note that row and columns names are not counted. Also note that all functions applied to vectors above can be used with what you extract out of a column of a data frame:

```
> which(a[,2]>450)¶
```

```
[1] 3 4 5
> a[,3][which(a[,3]>100)]¶
[1] 271 103 735
> a[,3][ a[,3]>100]¶
[1] 271 103 735
```

The most practical way to access individual columns, however, involves the function `attach` (and gets undone with `detach`). I will not get into the ideological debate about whether one should use `attach` or rather `with`, etc. – if you are interested in that, go to the R-Help list or read `?with`… You get no output, but you can now access any column with its name:

```
> attach(a)¶
> Class¶
[1] open   open   open   closed closed
Levels: closed open
```

Note two things. First, if you attach a data frame that has one or more names that have already been defined as data structures or as columns of previously attached data frames, you will receive a warning; in such cases, make sure you are really dealing with the data structures or columns you want and consider using `detach` to un-attach the earlier data frame. Second, when you use `attach` you are strictly speaking using 'copies' of these variables. You can change those, but these changes do not affect the data frame they come from.

```
> CLASS[4]<-NA; CLASS¶
[1] open   open   open   <NA>   closed
Levels: closed open
> a¶
   POS TOKENFREQ TYPEFREQ  CLASS
1  adj       421      271   open
2  adv       337      103   open
3    n      1411      735   open
4 conj       458       18 closed
5 prep       455       37 closed
```

Let's change `CLASS` back to its original state:

```
> CLASS[4]<-"closed"¶
```

If you want to change the data frame `a`, then you must make your changes in `a` directly, e.g. with `a$CLASS[4]<-NA¶` or `a$TOKENFREQ[2]<-338¶`. Given what you have seen in Section 2.4.3, however, this is only easy with vector or with factors where you do not add a new level – if you

want to add a new factor level, you must define that level first.

Sometimes you will need to investigate only a part of a data frame – maybe a set of rows, or a set of columns, or a matrix within a data frame. Also, a data frame may be so huge that you only want to keep one part of it in memory. As usual, there are several ways to achieve that. One uses indices in square brackets with logical conditions or `which`. Either you have already used `attach` and can use the column names directly or not:

```
> b<-a[CLASS=="open",]; b¶
  POS TOKENFREQ TYPEFREQ CLASS
1 adj       421      271  open
2 adv       337      103  open
3   n      1411      735  open

> b<-a[a[,4]=="open",]; b¶
  POS TOKENFREQ TYPEFREQ CLASS
1 adj       421      271  open
2 adv       337      103  open
3   n      1411      735  open
```

(Of course you can also write `b<-a[a$Class=="open",]¶`.) That is, you determine all elements of the column called `CLASS` / the fourth column that are `open`, and then you use that information to access the desired rows and all columns (hence the comma before the closing square bracket). There is a more elegant way to do this, though, the function `subset`. This function takes two arguments: the data structure of which you want a subset and the logical condition(s) describing which subset you want. Thus, the following line creates the same structure `b` as above:

```
> b<-subset(a, CLASS=="open")¶
```

The formulation "condition(s)" already indicates that you can of course use several conditions at the same time.

```
> b<-subset(a, CLASS=="open" & TOKENFREQ<1000); b¶
  POS TOKENFREQ TYPEFREQ CLASS
1 adj       421      271  open
2 adv       337      103  open
> b<-subset(a, POS %in% c("adj", "adv")); b¶
  POS TOKENFREQ TYPEFREQ CLASS
1 adj       421      271  open
2 adv       337      103  open
```

As I mentioned above, you will usually edit data frames in a spreadsheet software or, because the spreadsheet software does not allow for as many

rows as you need, in a text editor. For the sake of completeness, let me mention that R of course also allows you to edit data frames in a spreadsheet-like format. The function `fix` takes as argument a data frame and opens a spreadsheet editor in which you can edit the data frame; you can even introduce new factor levels without having to define them first. When you close the editor, R will do that for you.

Finally, let us look at ways in which you can sort data frames. Recall that the function `order` creates a vector of positions and that vectors can be used for sorting. Imagine you wanted to search the data frame `a` according to the column `CLASS` (in alphabetically ascending order), and within `Class` according to `TOKENFREQ` (in descending order). How can you do that?

THINK
BREAK

The problem is both sorting styles are different: one is `decreasing=FALSE`, the other is `decreasing=TRUE`. What you can do is apply `order` not to `TOKENFREQ`, but to the negative values of `TOKENFREQ`.

```
> order.index<-order(CLASS, -TOKENFREQ); order.index¶
[1] 4 5 3 1 2
```

After that, you can use the vector `order.index` to sort the data frame:

```
> a[order.index,]¶
   POS TOKENFREQ TYPEFREQ  CLASS
4 conj       458       18 closed
5 prep       455       37 closed
3    n      1411      735   open
1  adj       421      271   open
2  adv       337      103   open
```

Of course you can do that in just one line:[12]

```
> a[order(CLASS, -TOKENFREQ),]¶
```

You can now also use the function `sample` to sort the rows of a data frame randomly (for example, to randomize tables with experimental items;

12. Note that R is superior to many other programs here because the number of sorting parameters is in principle unlimited.

cf. above). You first determine the number of rows to be randomized (e.g., with `nrow` or `dim`) and then combine `sample` with `order`. Your data frame will probably be different because we used a random sampling.

```
> no.rows<-nrow(a)¶
> order.index<-sample(no.rows); order.index¶
[1] 3 4 1 2 5
> a[order.index,]¶
   POS TOKENFREQ TYPEFREQ  CLASS
3    n      1411      735   open
4 conj       458       18 closed
1  adj       421      271   open
2  adv       337      103   open
5 prep       455       37 closed

> a[sample(nrow(a)),] # in just one line¶
```

But what do you do when you need to sort a data frame according to several factors – some in ascending and some in descending order? You can of course not use negative values of factor levels – what would `-open` be? Thus, you first use the function `rank`, which rank-orders factor levels, and then you can use negative values of these ranks:

```
> order.index<-order(-rank(CLASS), -rank(POS))¶
> a[order.index,]¶
   POS TOKENFREQ TYPEFREQ  CLASS
3    n      1411      735   open
2  adv       337      103   open
1  adj       421      271   open
5 prep       455       37 closed
4 conj       458       18 closed
```

Recommendation(s) for further study

- the function `is.data.frame` to test if a data structure is a data frame
- the function `dim` for the number of rows and columns of a data frame
- the functions `read.csv` and `read.csv2` to read in tab-delimited files
- the function `save` to save data structures in a compressed binary format
- the function `with` to access columns of a data frame without `attach`
- the functions `cbind` and `rbind` to combine vectors and factors in a columnwise or rowwise way
- the function `merge` to combine different data frames
- the function `complete.cases` to test which rows of a data frame contain missing data / `NA`

6. Some programming: conditionals and loops

So far, we have focused on simple and existing functions but we have done little to explore the programming-language character of R. This section will introduce a few very powerful notions that allow you to make R decide which of two or more user-specified things to do and/or do something over and over again. In Section 2.6.1, we will explore the former, Section 2.6.2 then discusses the latter, but the treatment here can only be very brief and I advise you to explore some of the reading suggestions for more details.

6.1. Conditional expressions

Later, you will often face situations where you want to pursue one of several possible options in a statistical analysis. In a plot, for example, the data points for male subjects should be plotted in blue and the data points for female subjects should be plotted in pink. Or, you actually only want R to generate a plot when the result is significant but not, when it is not. In general, you can of course always do these things stepwise yourself: you could decide for each analysis yourself whether it is significant and then generate a plot when it is. However, a more elegant way is to write R code that makes decisions for you, that you can apply to any data set, and that, therefore, allows you to recycle code from one analysis to the next. Conditional expressions are one way – others are available and sometimes more elegant – to make R decide things. This is what the syntax can look like in a notation often referred to as pseudo code (so, no need to enter this into R!):

```
if (some logical expression testing a condition) {
   what to do if this logical expression evaluates to TRUE
   (this can be more than one line)
} else if (some other logical expression) {
   what to do if this logical expression evaluates to FALSE
   (this can be more than one line)
} else {
   what to do if all logical expressions above evaluate to
   FALSE
}
```

That's it, and the part after the first `}` is even optional. Here's an example with real code (recall, `"\n"` means 'a new line'):

```
> pvalue<-0.06¶
> if (pvalue>=0.05) {¶
```

```
+    cat("Not significant, p =", pvalue, "\n")¶
+ } else {¶
+    cat("Significant, p =", pvalue, "\n")¶
+ }¶
Not significant, p = 0.06
```

The first line defines a *p*-value, which you will later get from a statistical test. The next line tests whether that *p*-value is greater than or equal to 0.05. It is, which is why the code after the first opening `{` is executed and why R then never gets to see the part after `else`.

If you now set `pvalue` to 0.04 and run the `if` expression again, then this happens: Line 2 from above tests whether 0.04 is greater than or equal to 0.05. It is not, which is why the block of code between `{` and `}` before `else` is skipped and why the second block of code is executed. Try it.

A short version of this can be extremely useful when you have many tests to make but only one instruction for both when a test returns `TRUE` or `FALSE`. It uses the function `ifelse`, here represented schematically again:

```
ifelse(logical expression, what when TRUE, what when FALSE)
```

And here's an application:

```
> pvalues<-c(0.02, 0.00096, 0.092, 0.4)¶
> decisions<-ifelse (pvalues<0.05, "*", "ns")¶
> decisions¶
[1] "*"  "*"  "ns" "ns"
```

As you can see, `ifelse` tested all four values of `pvalues` against the threshold value of 0.05, and put the correspondingly required values into the new vector `decisions`. We will use this a lot to customize graphs.

6.2. Loops

Loops are useful to have R execute one or (many) more functions multiple times. Like many other programming languages, R has different types of loops, but I will only discuss `for`-loops here. This is the general syntax in pseudo code:

```
for (some.name in a.sequence) {
   what to do as often often as a.sequence has elements
   (this can be more than one line)
}
```

Let's go over this step by step. The data structure `some.name` stands for any name you might wish to assign to a data structure that is processed in the loop, and `a.sequence` stands for anything that can be interpreted as a sequence of values, most typically a vector of length 1 or more. This sounds more cryptic than it actually is, here's a very easy example:

```
> for (counter in 1:3) {¶
+    cat("This is iteration number", counter, "\n")¶
+ }¶
This is iteration number 1
This is iteration number 2
This is iteration number 3
```

When R enters the `for`-loop, it assigns to `counter` the first value of the sequence 1:3, i.e. 1. Then, in the only line in the loop, R prints some sentence and ends it with the current value of `counter`, 1, and a line break. Then R reaches the `}` and, because `counter` has not yet iterated over all values of `a.sequence`, re-iterates, which means it goes back to the beginning of the loop, this time assigning to `counter` the next value of `a.sequence`, i.e., 2, and so on. Once R has printed the third line, it exits the loop because `counter` has now iterated over all elements of `a.sequence`.

Here is a more advanced example, but one that is typical of what we're going to use loops for later. Can you see what it does just from the code?

```
> some.numbers<-1:100¶
> collector<-vector(length=10)¶
> for (i in 1:10) {¶
+    collector[i]<-mean(sample(some.numbers, 50))¶
+ }¶
> collector¶
[1] 50.78 51.14 45.04 48.04 55.30 45.90 53.02 48.40 50.38
   49.88
```

THINK
BREAK

The first line generates a vector `some.numbers` with the values from 1 to 100. The second line generates a vector called `collector` which has 10 elements and which will be used to collect results from the looping. Line 3 begins a loop of 10 iterations, using a vector called `i` as the counter. Line 4 is the crucial one now: In it, R samples 50 numbers randomly without replacement from the vector `some.numbers`, computes the mean of these 50

numbers, and then stores that mean in the `i`-th slot of collector. On the first iteration, `i` is of course 1 so the first mean is stored in the first slot of `collector`. Then R iterates, `i` becomes 2, R generates a second random sample, computes its mean, and stores it in the – now – 2nd slot of `collector`, and so on, until R has done the sampling, averaging, and storing process 10 times and exits the loop. Then, the vector `collector` is printed on the screen.

In Chapter 4, we will use an approach like this to help us explore data that violate some of the assumptions of common statistical tests. However, it is already worth mentioning that loops are often not the best way to do things like the above in R: in contrast to some other programming languages, R is designed such that it is often much faster and more memory-efficient to do things not with loops but with members of the `apply` family of functions, which you will get to know a bit later. Still, being able to quickly write a loop and test something is often a very useful skill.

Recommendation(s) for further study
– the functions `next` and `break` to control behavior of/in loops

7. Writing your own little functions

The fact that R is not just a statistics software but a full-fledged programming language is something that can hardly be overstated enough. It means that nearly anything is possible: the limit of what you can do with R is not defined by what the designers of some other software thought you may want to do – the limit is set pretty much only by your skills and maybe your RAM/processor (which is one reason why I recommend using R for corpus-linguistic analyses, see Gries 2009a). One aspect making this particularly obvious is how you can very easily write your own functions to facilitate and/or automate tedious and/or frequent tasks. In this section, I will give a few very small examples of the logic of how to write your own functions, mainly because we haven't dealt with any statistical functions yet. Don't despair if you don't understand these programming issues immediately – for most of this book, you will not need them, but these capabilities can come in very handy when you begin to tackle more complex data. Also, in Chapter 3 and 4 I will return to this topic so that you get more practice in this and end up with a list of useful functions for your own work.

The first example I want to use involves looking at a part of a data structure. For example, let's assume you loaded a really long vector (let's

say, 10,000 elements long) and want to check whether you imported it into R properly. Just printing that onto the screen is somewhat tedious since you can't possibly read all 10,000 items (let alone at the speed with which they are displayed), nor do you usually need all 10,000 items – the first *n* are usually enough to see whether your data import was successful. The same holds for long data frames: you don't need to see all 1600 rows to check whether loading it was successful, maybe the first 5 or 6 are sufficient. Let's write a function `peek` that by default shows you the first 6 elements of each of the data structures you know about: one-dimensional vectors or factors and two-dimensional data frames.

One good way to approach the writing of functions is to first consider how you would solve that problem just for a particular data structure, i.e. outside of the function-writing context, and then make whatever code you wrote general enough to cover not just the one data structure you just addressed, but many more. To that end, let's first load a data frame for this little example (from <_inputfiles/02-7_dataframe1.csv>):

```
> into.causatives<-read.delim(file.choose())¶
> str(into.causatives)¶
'data.frame':1600 obs. of  5 variables:
$ BNC       : Factor w/ 929 levels "A06","A08","A0C",..:
   1 2 3 4 ...
$ TAG_ING   : Factor w/ 10 levels "AJ0-NN1","AJ0-VVG",..:
   10 7 10 ...
$ ING       : Factor w/ 422 levels "abandon-
   ing","abdicating",..: 354 49 382 ...
$ VERB_LEMMA: Factor w/ 208 levels "activate","aggravate",..:
   76 126 186 ...
$ ING_LEMMA : Factor w/ 417 levels "abandon","abdicate",..:
   349 41 377 ...
```

Now, you want to work with one-dimensional and two-dimensional vectors, factors, and data frames. How would you get the first six elements of each of these? That you already know. For vectors or factors you'd write:

```
vector.or.factor[1:6]
```

and for data frames you'd write:

```
data.frame[1:6,]
```

So, essentially you need to decide what the data structure is of which R is supposed to display the first *n* elements (by default 6) and then you subset with either `[1:6]` or `[1:6,]`. Since, ultimately, the idea is to have R –

not you – decide on the right way of subsetting (depending on the data structure), you use a conditional expression:

```
> if (is.data.frame(into.causatives)) {¶
>    into.causatives[1:6,]¶
> } else {¶
>    into.causatives[1:6]¶
> }¶
  BNC TAG_ING      ING VERB_LEMMA ING_LEMMA
1 A06     VVG speaking      force     speak
2 A08     VBG    being      nudge        be
3 A0C     VVG   taking       talk       tak
4 A0F     VVG   taking      bully      take
5 A0H     VVG   trying  influence       try
6 A0H     VVG thinking     delude     think
```

To turn this into a function, you wrap a function definition (naming the function `peek`) around this piece of code. However, if you use the above code as is, then this function will use the name `into.causatives` in the function definition, which is not exactly very general. As you have seen, many R functions use `x` for the main obligatory variable. Following this tradition, you could write this:

```
> peek<-function (x) {¶
>    if (is.data.frame(x)) {¶
>       x[1:6,]¶
>    } else {¶
>       x[1:6]¶
>    }¶
> }¶
> peek(into.causatives)¶
```

This means, R defines a function called `peek` that requires an argument, and that argument is function-internally called `x`. When you call `peek` with some argument – e.g., `into.causatives` – then R will take the content of that data structure and, for the duration of the function execution, assign it to `x`. Then, within the function R will carry out all of `peek` with `x` and return/output the result, which is the first 6 rows of `into.causatives`.

It seems like we're done. However, some things are missing. When you write a function, it is crucial you make sure it covers all sorts of possibilities or data you may throw at it. After all, you're writing a function to make your life easier, to allow you not to have to worry about stuff anymore after you have thought about it once, namely when you wrote the function. There are three ways in which the above code should be improved:

– what if the data structure you use `peek` with is not a vector or a factor or

a data frame?
- what if you want to be able to see not 6 but *n* elements?
- what if the data structure you use `peek` with has fewer than *n* elements or rows?

To address the first possibility, we just add another conditional expression. So far we only test whether whatever we use `peek` with is a data frame – now we also need to check whether, if it is not a data frame, whether it then is a vector or a factor, and ideally we return some warning if the data structure is none of the three.

To address the second possibility, we need to be able to tell the function flexibly how many parts of `x` we want to see, and the way we tell this to a function is of course by its arguments. Thus, we add an argument, let's call it `n`, that says how much we want to see of `x`, but we make 6 the default.

To address the final possibility, we have to make sure that R realizes how many elements `x` has: if it has more than `n`, R should show `n`, but if it has fewer than `n`, R should show as many as it can, i.e., all of them.

This version of `peek` addresses all of these issues:

```
> peek< function (x, n=6) {¶
>    if (is.data.frame(x)) {¶
>       return(x[1:min(nrow(x), n),])¶
>    } else if (is.vector(x) | is.factor(x)) {¶
>       return(x[1:min(length(x), n)])¶
>    } else {¶
>       cat("Not defined for other data structures ...\n")¶
>    }¶
> }¶
```

Issue number one is addressed by adding a second conditional with the `else if` test – recall the use of | to mean 'or' – and outputting a message if `x` is neither a vector, factor, or a data frame.

Issue number two is addressed by adding the argument `n` to the function definition and using `n` in the body of the function. The argument `n` is set to 6 by default, so if the user does not specify `n`, 6 is used, but the user can also override this with another number.

The final issue is addressed by tweaking the subsetting: instead of using just `n`, we use `1:` the minimum of `n` or the number of elements `x` has. Thus, if `x` has more than `n` elements, then `n` will be the minimum and we get to see `n` elements, and if `x` has less than `n` elements, then that number of elements will be the minimum and we get to see them all.

Finally, also note that I am now using the function `return` to specify

exactly what `peek` should return and output to the user when it's done. Try the following lines (output not shown here and see the comments in the code file) to see that it works:

```
> peek(into.causatives)¶
> peek(into.causatives, 3)¶
> peek(into.causatives, 9)¶
> peek(21:50, 10)¶
> peek(into.causatives$BNC, 12)¶
> peek(as.matrix(into.causatives))¶
```

While all this may not seem easy and worth the effort, we will later see that being able to write your own functions will facilitate quite a few statistical analyses below. Let me also note that this was a tongue-in-cheek example: there is actually already a function in R that does what `peek` does (and more, because it can handle more data structures) – look up `head` and also `tail` ;-).

Now you should do the exercise(s) for Chapter 2 …

Recommendation(s) for further study

- the functions `NA`, `is.na`, `NaN`, `is.nan`, `na.action`, `na.omit`, and `na.fail` on how to handle missing data
- Ligges (2005), Crawley (2007), Braun and Murdoch (2008), Spector (2008), Gentleman (2009), and Gries (2009a) for more information on R: Ligges (2005), Braun and Murdoch (2008), and Gentleman (2009) on R as a (statistical) programming language, Crawley as a very comprehensive overview, Spector (2008) on data manipulation in R, and Gries (2009a) on corpus-linguistic methods with R