Chapter 2
Fundamentals of R

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 first take the most important step: the installation of R (first largely for Windows).

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 .dmg file; for Linux, you choose your distribution and maybe your distribution version and then the relevant file;[12]
3. choose "complete installation" into the suggested standard directory;
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 choose all of them, but I minimally

12 Depending on your Linux distribution, you may also be able to install R and many frequently-used packages using a distribution-internal package manager).

recommend `amap`, `car`, `Design`, `Hmisc`, `lattice`, `MASS`, `qcc`, `rpart`, and `vcd`. (You can also enter, say, `install.packages("car")¶` at the console to install said package and ideally do that with administrator/root rights; that is, 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.)

As a next step, you should download the files with example files, all the code, exercises, and answer keys onto your hard drive. Create a folder such as <C:/_sflwr/> on your harddrive (for statistics for linguists with R). Then download all files from website of the Google group "StatForLing with R" hosting the companion website of this book (<http://groups.google.com/group/statforling-with-r/web/statistics-for-linguists-with-r> or the mirror at <http://www.linguistics.ucsb.edu/faculty/stgries/research/sflwr/sflwr.html>) and save them into the right folders:

– <C:/_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_R";
– <C:/_sflwr/_outputfiles>: this folder will contain output files from Chapters 2 and 5; to unzip these files, you will need the password "squi_Rrel";
– <C:/_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_R".

(By the way, I am using regular slashes here because you can use those in R, too, and more easily so than backslashes.) On Linux, you may want to use your main user directory as in <home/user/sflwr>.) 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. As a Windows user, you can use Tinn-R or Notepad++; the former already has syntax highlighting for R; the latter can be easily configured for the same functionality. As a Mac user, you can use R.app. As a Linux user, I use gedit (with the Rgedit plugin) or actually configure Notepad++ with Wine.

After all this, you can view all scripts in <C:/_sflwr/_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 and then paste them into R because the syntax highlighting 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 or Python. 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 statistical tests, graphical evaluation, and probability distributions.

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 a general introduction to R, and I refer you to the recommendations for further study and 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, <C:/_sflwr/_inputfiles/04-1-1-1_tense-aspect.txt>, 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))¶` and will usually be given in grey blocks of several lines. This character "·" represents a space (which are nearly always optional) and 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):

```
> 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 distinguish your input from R's output more easily. 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.

```
> 2-¶
+ 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 13, where "→" and "¶" again 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, OpenOffice.org 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 13. 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 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 OpenOffice.org 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*. Arguments can be null, in which case the function name is just followed by opening and closing parentheses. The 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. 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, namely the figure of which you want the square root. Thus:

```
> sqrt(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(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. Thus,

```
> print(a)¶
[1]·2.236068
```

Most of the time, it is enough to simply enter the name of the relevant data structure:

```
> 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. Second, the assignment operator can also be used to assign a new value to an existing data structure. For example,

```
> a<-sqrt(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·
      structures¶
```

Third, it will be very important later on to know that many functions have default settings for their arguments. This means that if you use a function without specifying all the necessary arguments, then R will use the default settings of that function. Let us explore this on the basis of the very useful function `sample`. This function generates random or pseudorandom samples of elements and can take up to four arguments:

- `x`: a data structure – typically a vector – containing the elements of which you want a sample;
- `size`: a positive integer giving the size of the sample;
- the assignment `replace=TRUE` (if the elements of the vector can be sampled multiple times, sampling with replacement) or `replace=FALSE` (if each element of the vector can only be sampled once, the default setting);
- `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 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)·#·or·just·some.data<-1:10¶
```

If you want to sample 5 elements from this vector equiprobably and with replacement, you can enter the following:[13]

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

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:

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

With the following line, 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. If you do that, R assumes you 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 to 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
```

13. 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 you defined one), 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 hand. While this may ultimately boil down to a matter of personal preferences, 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 function `formals`, which provides the arguments a function needs, their default settings, and their default order (try `formals(sample)¶`)

3. Vectors

3.1. Generating vectors in R

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 nearly all 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 result of `sqrt(5)` is a vector that is one element long and whose first (and only) element is 2.236068. You can test this with R: first, you assign the result of `sqrt(5)` to a data structure called `a`.

```
> a<-sqrt(5)¶
```

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

```
> is.vector(a)¶
[1]·TRUE
```

And the function `length` determines and returns the number of elements of the data structure provided as an 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·#·several·functions·in·one·line·
      are·separated·by·semicolons¶
[1]·"John"
```

(Actually, there are six different vector types, but 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 (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. This function takes at least two and maximally three arguments (and as usual the different arguments are separated by commas):

– x: a vector to which something should be appended;
– values: the vector to be appended;
– after: the position in the data structure of 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)·#·combine·
      vectors¶
> 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)¶
> numbers<-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"·····"caesar"
```

and

```
> numbers<-c(1,·2,·3);·names.of.numbers<-c("four",·"five",·
       "six")·#·generate·two·vectors¶
> names.and.names.of.numbers<-c(numbers,·names.of.numbers)·#·
       combine·vectors¶
> names.and.names.of.numbers¶
[1]·"1"····"2"····"3"····"four"·"five"·"six"
```

The double quotes around 1, 2, and 3 indicate that these are 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·[1:3]·1·2·3¶
> str(numbers.and.names.of.numbers)
·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 elements or sequences of elements are repeated. Instead of typing those into R, 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)¶
```

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:

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

or

```
> x<-rep(seq(2,·10,·2),·6)¶
```

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 a vector of numbers) or `scan(what=character(0))¶` (for vectors of 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 in R

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>. Thus, if the data file has not been created with a text editor but a spreadsheet software such as OpenOffice.org 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;
- `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)`;
- `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 <C:/_sflwr/_inputfiles/02-3-2_vector1.txt>, which contains what is shown in Figure 14, into a vector `x`, you enter this.

```
1¶
2¶
3¶
4¶
5¶
```

Figure 14. An example file

```
> x<-scan(file="C:/_sflwr/_inputfiles/02-3-2_vector1.txt",·
      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 15) is just as easy:

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

Figure 15. Another example file

```
> x<-scan(file="C:/_sflwr/_inputfiles/02-3-2_vector2.txt",·
      what=character(0),·sep="·",·quiet=TRUE)¶
```

The argument `quiet=TRUE` suppresses the output of how many elements were read. You get:

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

On a Windows-based PC, you can also use the function `choose.files()` without arguments; R will prompt you to choose one or more files in a file explorer window:

```
> x<-scan(file=choose.files(),·what=character(0),·sep="·",·
      quiet=TRUE)·#·and·then·choose·<C:/_sflwr/_inputfiles/
      02-3-2_vector2.txt>¶
> x¶
[1]·"alpha"··"bravo"··"charly"·"delta"··"echo"
```

If you use R on another operating system, you can either use the function `file.choose()`, which only allows you to choose one file, or you can proceed as follows: After you entered the following line into R,

```
> choice<-select.list(dir(scan(nmax=1,·what=character(0)),·
      full.names=TRUE),·multiple=TRUE)¶
```

you first enter the path to the directory in which the relevant file is located, for example “C:/_sflwr/_inputfiles”. Then R will show to you all the files in that directory and you can choose one (or more) of these and then load it with the following line:[14]

14. Below and in the scripts I will mostly use `choose.files` with the argument `default="…"`; that argument provides the path to the required file. On PCs running Microsoft Windows – for some reason certainly still the most widely used operating system – this is more convenient than `file.choose()` and allows you to access the relevant file immediately just by pressing ENTER (if you have stored the files in the recommended directories, that is). As a Mac- or Linux User you (i) change the `file=choose.files(…)` argument into `file=file.choose()` and then enter a path when promoted to read in, or write into, an already existing file.

```
> x<-scan(choice,·what=character(0),·sep="·")¶
```

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 file into which the vector is to saved (or, on Windows PCs, `choose.files()`);
– `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=choose.files())·#·and·then·choose·<C:/_sflwr/
      _outputfiles/02-3-2_vector3.txt>¶
```

Recommendation(s) for further study
the functions `write` as a wrapper for `cat` to save vectors

3.3. Editing vectors in R

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 as its argument, `ceiling` returns the smallest integers not less than the corresponding elements of the vector provided as an argument, 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
```

That also means you can round in the 'traditional' way by using `floor` as follows:

```
> floor(a+0.5)¶
[1]·3·3·4·4·4
```

Or, in a more abstract, but also more versatile, way:

```
> digits<-0
> floor(a*10^digits+0.5*10^digits)¶
[1]·3·3·4·4·4
> digits<-1
> floor(a*10^digits+0.5)/10^digits¶
[1]·3.4·3.5·3.5·3.6·3.6
```

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

```
> x<-c("a",·"b",·"c",·"d",·"e")¶
> x[3]·#·access·the·3.·element·of·x¶
[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]·#·access·the·3.·element·of·x¶
[1]·"c"
```

and

```
> z<-c(1,·3);·x[z]·#·access·elements·1·and·3·of·x¶
[1]·"a"·"c"
```

and

```
> z<-c(1:3)¶
> x[z]·#·access·elements·1·to·3·of·x¶
[1]·"a"·"b"·"c"
```

With negative numbers, you can leave out elements:

```
> x[-2]·#·access·x·without·the·2.·element¶
[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)·#·generate·vector·with·the·numbers·from·10·to·1¶
> x¶
·[1]·10··9··8··7··6··5··4··3··2··1
> x==4·#·which·elements·of·x·are·4?¶
·[1]·FALSE·FALSE·FALSE·FALSE·FALSE·FALSE··TRUE·FALSE·FALSE·
      FALSE
> x<=7·#·which·elements·of·x·are·<=·7?¶
·[1]·FALSE·FALSE·FALSE··TRUE··TRUE··TRUE··TRUE··TRUE··TRUE··
      TRUE
> x!=8·#·which·elements·of·x·are·not·8?¶
·[1]··TRUE··TRUE·FALSE··TRUE··TRUE··TRUE··TRUE··TRUE··TRUE··
      TRUE
```

```
> (x>8·|·x<3)·#·which·elements·of·x·are·>·8·or·<·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 expressions is true in a vector:

```
> sum(x==4)¶
[1]·1
> sum(x>8·|·x<3)·#·an·example·using·or¶
[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 and must locate the TRUEs 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 its argument is the kind of logical expression 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 that the seventh element of `x` is a 4. 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:

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

Or you immediately combine these two steps:

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

or even

```
> 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))·#·or·sum(x>8·|·x<3)·as·above¶
[1]·4
```

Sometimes you may want to test for several elements at once, 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, 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 more and more obvious that the fact much of what R does happens in terms of 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)·#·store·the·positions·of·the·elements·
       of·x·that·are·larger·than·8·into·a·vector·y¶
> x[y]<-12·#·replace·these·elements·of·x·with·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·#·restore·x¶
> 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·#·restore·x¶
> 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:

```
> x<-c(10:1);·y<-c(2,·5,·9)·#·restore·x·and·y¶
> setdiff(x,·y)¶
[1]·10··8··7··6··4··3··1
> setdiff(y,·x)¶
numeric(0)
```

The function `intersect` returns the elements of the first vector that are also in the second vector.

```
> intersect(x,·y)¶
[1]·9·5·2
> intersect(y,·x)¶
[1]·2·5·9
```

The function `union` returns all elements that occur in at least one of the two vectors.

```
> union(x,·y)¶
[1]·10··9··8··7··6··5··4··3··2··1
> union(y,·x)¶
[1]··2··5··9·10··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 a set of 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)·#·restore·x¶
> 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 cannot be divided by the length of the shorter vector without a remainder. In such cases, R will recycle the shorter vectors as often as possible, but will also return a warning:

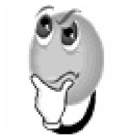
```
> x<-c(2,·3,·4,·5,·6);·y<-c(8,·9)¶
> x*y¶
[1]·16·27·32·45·48
Warning·message:
longer·object·length
········is·not·a·multiple·of·shorter·object·length·in:·x·*·y
```

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)·#·generate·a·vector·x¶
> y<-sort(x)·#·sort·x·in·ascending·order¶
> z<-sort(x,·decreasing=TRUE)·#·sort·x·in·descending·order¶
> 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(3,·5,·10,·1,·6,·7,·8,·2,·4,·9)¶
> order(z,·decreasing=FALSE)¶
[1]··4··8··1··9··2··5··6··7·10··3
```



THINK BREAK

The output of `order` when applied to a vector `z` is a vector which provides the order of the elements of `z` when they are sorted as specified. Let us clarify this rather opaque characterization by means of this example. If you wanted to sort the values of `z` in increasing order, you would first have to take `z`'s fourth value (which is the smallest value, 1). Thus, the first value of `order(z,·decreasing=FALSE)¶` is 4. The next value you would have to take is the eighth value of `z`, which is 2. The next value you would take is the first value of `z`, namely 3, etc. The last value of `z` you would take is its third one, 10, the maximum value of `z`. (If you provide `order` with more than vector, additional vectors are used to break ties.) As we will see below, this function will turn out to be very handy 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 `rev` to reverse the elements of a vector
- the function `abs` to obtain the absolute values of a numerical vector

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 tables and want R to recognize that some of the columns in tables are nominal or categorical variables.

4.1. Generating factors in R

As I just mentioned, factors are mainly used to code nominal or categorical variables, i.e. in situations where a variables has two or more (but usually not very many) qualitatively different levels. The simplest way to create a factor is to first generate a vector and then change that vector into a factor using the function `factor`.

```
> rm(list=ls(all=T))·#·clear·memory;·recall:·T·/·F·=·
       TRUE·/·FALSE¶
```

(While using `T` and `F` for `TRUE` and `FALSE` is a convenient shortcut, be aware that you can also define `T` and `F` yourself with anything you want. Thus, if you use `T` and `F` for `TRUE` and `FALSE`, make sure you never overwrite them with something else.)

```
> 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
```

The function `factor` 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 explained in Section 2.4.3 below.

When you output a factor, you can see one difference between factors and vectors because the output includes a list of all factor levels that occur at least once. It is not a perfect analogy, but you can look at it this way: `levels(FACTOR)¶` generates something similar to `unique(VECTOR)¶`.

4.2. Loading and saving factors in R

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 do with a vector, your output file will look like this:

```
> cat(a,·sep="\n",·file="C:/_sflwr/_outputfiles/02-
      42_factor1.txt")¶
```

```
1¶
3¶
2¶
```

Figure 16. 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 should simply force R to treat the factor as a vector for the output, which will produce the desired result.

```
> cat(as.vector(a),·sep="\n",·file="C:/_sflwr/_outputfiles/
      02-4-2_factor2.txt")¶
```

4.3. Editing factors in R

Editing factors is similar to editing vectors, but sometimes a small additional difficulty arises:

```
> x<-c(rep("long",·3),·rep("short",·3))¶
> x<-factor(x);·x¶
·[1]·long··long··long··short·short·short
Levels:·long·short
> x[2]<-"short"¶
> x¶
·[1]·long··short·long··short·short·short
Levels:·long·short
```

Thus, if your change only consists of assigning a level *that already exists in the factor* to another position in the factor, then you can treat vectors and factors alike. The difficulty arises when you assign a new level:

```
> x[2]<-"intermediate"¶
Warning·message:
In·`[<-.factor`(`*tmp*`,·2,·value·=·"intermediate")·:
··invalid·factor·level,·NAs·generated
> x¶
·[1]·long··<NA>··long·short·short·short
Levels:·long·short
```

Thus, if you want to assign a new level, you first must tell R that. You can do that with `factor`, but now you also must use the argument `levels`:

```
> x<-c(rep("long",·3),·rep("short",·3))·#·as·above¶
> x<-factor(x,·levels=c("long",·"short",·"intermediate"))·#·
      introducing·the·new·level¶
> x·#·x·has·not·changed·apart·from·the·levels¶
·[1]·long··long··long··short·short·short
Levels:·long·short·intermediate
> x[2]<-"intermediate"·#·now·you·can·use·the·new·level¶
> x¶
·[1]·long·········intermediate·long·········short·······short
      ·······short
Levels:·long·short·intermediate
```

Recommendation(s) for further study

– the function `gl` to create factors
– the function `reorder` and `relevel` to reorder the levels of a factor

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 a table, is actually only a specific type of another data structure, the 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.

5.1. Generating data frames in R

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:

– 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.

Imagine also the data frame or table you wanted to generate is the one in Figure 17. Step 1: you generate four vectors, one for each column of the table:

```
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 17. An example data frame

```
> rm(list=ls(all=T))·#·clear·memory¶
> PartOfSp<-c("ADJ",·"ADV",·"N",·"CONJ",·"PREP")¶
> TokenFrequency<-c(421,·337,·1411,·458,·455)¶
> TypeFrequency<-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:

```
> x<-data.frame(PartOfSp,·TokenFrequency,·TypeFrequency,·
      Class)¶
```

(The order of vectors is not really important, but determines the order of columns.) Now you can look at the data frame’s characteristics:

```
> x¶
··PartOfSp·TokenFrequency·TypeFrequency··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:
·$·PartOfSp······:·Factor·w/·5·levels·"ADJ","ADV","CONJ",..:·
       1·2·4·3·5
·$·TokenFrequency:·num···421··337·1411··458··455
·$·TypeFrequency·:·num··271·103·735·18·37
·$·Class·········:·Factor·w/·2·levels·"closed","open":·2·2·2·
       1·1
```

Within the data frame, R has changed the vectors of character strings in 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 not know or guess that these are in fact factors, 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 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(TokenFrequency,·TypeFrequency,·Class,·
       row.names=PartOfSp)¶
> x¶
·····TokenFrequency·TypeFrequency··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:
·$·TokenFrequency:·num···421··337·1411··458··455
·$·TypeFrequency·:·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 `PartOfSp` now functions as row names. Note that this is only possible when the column with the row names contains no element twice.

5.2. Loading and saving data frames in R

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 Openoffice.org 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., Tinn-R 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.[15] 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=T`: an indicator of whether the first row of the file contains column headers (which it should always have) or `header=F` (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 <C:/_sflwr/_inputfiles/02-5-2_dataframe1.txt> – once without row names and once with row names – then this is what you enter on a Windows PC:

```
> a<-read.table(choose.files(),·header=T,·sep="\t",·quote=""
        ,·comment.char="")·#·no·row·numbers:·R·numbers·rows¶
```

15. To do the same in Microsoft Excel, you save the file as a tab-delimited text file.

or

```
> a<-read.table(choose.files(),·header=T,·sep="\t",·quote=""
      ,·comment.char="",·row.names=1)·#·with·row·numbers:·
      R·does·not·number·rows¶
```

By entering `a¶` or `str(a)¶`, you can check whether the data frame has been loaded correctly. If you want to save a data frame from R, then you 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; on Windows PCs you can again use `choose.files()`;
- `append=F` (the default) or `append=T`: the former generates or overwrites the defined file, the latter appends the data frame to that file;
- `quote=T` (the default) or `quote=F`: the former prints factor levels with double quotes; the letter 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=T` (the default) or `row.names=F`: whether you want row names or not;
- `col.names=T` (the default) or `col.names=F`: whether you want column names or not.

Given the above default settings and under the assumption that your operating system uses an English locale, there are two most common ways to save such data frames: if you have a data frame *without* row names (i.e., the first version of `a` we looked at), you enter the following line: `write.table(a,·choose.files(),·quote=F,·sep="\t")¶`. If you have a data frame *with* row names (the second version of `a` we looked at), you add `col.names=NA` so that the column names stay in place:

```
> write.table(x,·file.choose(default="C:/_sflwr/_outputfiles/
      02-5-2_dataframe2.txt"),·quote=F,·sep="\t",·
      col.names=NA)¶
```

(Cf. <C:/_sflwr/_outputfiles/02-5-2_dataframe2.txt> for the output). The default setting for the decimal separator can be changed in OpenOffice.org Calc (*Tools: Options: Language Settings: Languages*) or in Microsoft Excel (*Tools: Options: International*) or – for the locale – in Windows (*Control panel: Regional and language options*).

5.3. Editing data frames in R

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 <C:/_sflwr/_inputfiles/02-5-3_dataframe.txt>.

```
> rm(list=ls(all=T))·#·clear·memory¶
> a<-read.table(choose.files(),·header=T,·sep="\t",·
      comment.char="",·quote="")¶
> a¶
··PartOfSp·TokenFrequency·TypeFrequency··Class
1······ADJ············421···········271···open
2······ADV············337···········103···open
3········N···········1411···········735···open
4·····CONJ············458············18·clossd
5·····PREP············455············37·closed
> a$TokenFrequency¶
[1]··421··337·1411··458··455
> a$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$TokenFrequency/a$TypeFrequency;·ratio¶
[1]··1.553506··3.271845··1.919728·25.444444·12.297297
```

You can also use indices in square brackets for subsetting. Above, we discussed how you can access parts of vectors or factors by putting the position of an element into square brackets. Vectors and factors 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]·#·the·value·in·row·2·and·column·3¶
[1]·103
> a[2,]·#·the·values·in·row·2,·since·no·column·is·defined¶
··PartOfSp·TokenFrequency·TypeFrequency·Class
2······ADV············337···········103··open
> a[,3]·#·the·values·in·column·3,·since·no·row·is·defined¶
[1]·271·103·735··18··37
> a[2:3,4]·#·values·2·and·3·of·column·4¶
[1]·open·open
Levels:·closed·open
> a[2:3,3:4]·#·values·2·and·3·of·column·3·and·4¶
··TypeFrequency·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
```

The most practical way to access individual columns, however, involves the function `attach` (and gets undone with `detach`). 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, however, that you now use 'copies' of these variables. You can change those, but these changes do not affect the data frame `a` they come from.

```
> Class[4]<-NA;·Class¶
[1]·open···open···open···<NA>···closed
Levels:·closed·open
> a¶
··PartOfSp·TokenFrequency·TypeFrequency··Class
1······ADJ············421···········271···open
2······ADV············337···········103···open
3········N···········1411···········735···open
4·····CONJ············458············18·clossd
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, for example with `a$TokenFrequency[2]<-338¶` or `a$Class[4]<-NA¶`. Given what you have seen in Section 2.4.3, however, this is only easy with factors where you do not add a new level or vectors – 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

```
> b<-a[Class=="open",];·b¶
··PartOfSp·TokenFrequency·TypeFrequency··Class
1······ADJ············421···········271···open
2······ADV············337···········103···open
3········N···········1411···········735···open
```

or not:

```
> b<-a[a[,4]=="open",];·b¶
··PartOfSp·TokenFrequency·TypeFrequency··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 (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 frame 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"·&·TokenFrequency<1000);·b¶
··PartOfSp·TokenFrequency·TypeFrequency··Class
1······ADJ············421···········271···open
2······ADV············337···········103···open
> b<-subset(a,·PartOfSp·%in%·c("ADJ",·"ADV"));·b¶
··PartOfSp·TokenFrequency·TypeFrequency··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 `TokenFrequency` (in descending order). How can you do that?

THINK BREAK

One problem here is that both sorting styles are different: one is `decreasing=F`, the other is `decreasing=T`. What you can do is this:

```
> order.index<-order(Class,·-TokenFrequency);·order.index¶
[1]·4·5·3·1·2
```

That is, you do not apply `order` to `TokenFrequency`, but to the negative values of `TokenFrequency`. Once that is done, you can use the vector `order.index` to sort the data frame:

```
> a[order.index,]¶
··PartOfSp·TokenFrequency·TypeFrequency··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:[16]

```
> a[order(Class,·-TokenFrequency),]¶
```

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; cf. above). You first determine the number of rows to be randomized (with `dim`) and then combine `sample` with `order`:

```
> no.rows<-dim(a)[1]·#·or·e.g.:·no.rows<-length(a$Class)¶
> order.index<-sample(no.rows);·order.index¶
[1]·1·4·2·3·5
> a[order.index,]¶
··PartOfSp·TokenFrequency·TypeFrequency··Class
1······ADJ············421···········271···open
4·····CONJ············458············18·clossd
2······ADV············337···········103···open
3········N···········1411···········735···open
5·····PREP············455············37·closed
```

You data frame will probably be different because we used a random sampling. Once more in just one line:

```
> a[sample(dim(a)[1]),]¶
```

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 use the function `rank`, which first rank-orders factor levels, and then you can use negative values of these ranks:

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

```
> order.index<-order(-rank(Class),·-rank(PartOfSp))¶
> a[order.index,]¶
··PartOfSp·TokenFrequency·TypeFrequency··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·clossd
```

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

Recommendation(s) for further study

- the functions `nrow` and `ncol` for the number of rows and columns of a data frame (as a more direct way than `dim(x)[1]` and `dim(x)[2]`)
- the argument `colClasses` of the function `read.table` to tell R which columns should be converted to vectors and which to factors
- the functions `read.csv` and `read.csv2` to read in simple Microsoft Excel files
- the function `read.delim` to load text files with useful default settings
- the function `read.spss` to load simple SPSS files
- 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 `with` to access columns of a data frame without `attach`
- the functions `NA`, `is.na`, `NaN`, `is.nan`, `na.action`, `na.omit`, and `na.fail` on how to handle missing data
- the function `complete.cases` to test which rows of a data frame contain missing data / `NA`
- the function `save` to save data structures in a compressed binary format
- Ligges (2005), Crawley (2007), Braun and Murdoch (2008), Spector (2008), Gentleman (2009), and Gries (2009) 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 (2009) on corpus-linguistic methods with R