CHAPTER 2

Introduction to R

What you will learn from this chapter:

In this chapter you will learn to install the basic distribution of R, as well as add-on packages. The chapter also introduces the basics of R syntax and demonstrates how to perform simple operations with different R objects. Special attention is paid to importing and exporting your own data to and from R and saving your graphical output. You will also be able to interpret error messages and warnings that R may give you and search for additional information on R functions.

2.1 Why use R?

In the old days, statistical tests were performed with pencil and paper. Today we are lucky to have computers, which can do complex calculations with amazing speed. This book will show how to use R for statistical analyses and creation of graphs. Although R is also a programming language (that is, one can write their own functions and execute them in R), this aspect is not covered in this book. Its goal is to demonstrate how to use the most popular functions written by other programmers and statisticians and interpret the output.

Why use R? The reasons are manifold. First, R seems to have become the *de facto* standard tool in many areas of linguistics, especially corpus-based and computational studies. Many leading quantitative linguists use R. Second, R offers a wealth of various functions and packages for specific tasks written by professional statisticians and specialists in different fields (at the moment of writing, the number of downloadable packages in the CRAN repository was 6050). Although it takes some time to master the basics of R syntax, you will be rewarded by its amazing flexibility and wealth of useful information. In case of questions or problems, one can rely on the support of the vast and dynamic R community. And last but not least, R is freely available for download. Created and supported by thousands of enthusiasts all over the world, it is distributed under the GNU General Public License, which gives end users the right to use, study, share and modify the software.

2.2 Installation of the basic distribution and add-on packages

If you are a Windows or Mac OS X user, you can install R from http://cran.r-project.org/. CRAN is the Comprehensive R Archive Network. It is a collection of sites (mirrors) all over the world that carry identical material (packages, documentation, etc.). The purpose of mirrors is to reduce network load. Click on *Mirrors* in the main menu on the left and select the mirror site that is the closest to your location. Choose the binaries for base distribution that match your OS and download the Installer package. When the download is completed, double-click on the Installer and follow the directions of the setup wizard. If you use a 64-bit version of Windows, it is recommended to choose the 64-bit version of R when the installer asks you to make the choice between the 64-bit and 32-bit versions. This will make large computations run faster. In addition to the base packages, it is strongly recommended to install the documentation.

If you use Linux, you should check with your package management system, since R is part of many Linux distributions. However, such prebuilt copies of R may be outdated. To obtain the latest version, you may want to get the distribution directly from CRAN. Two scenarios are possible in that case. For some Unix and Unix-like platforms, you can find precompiled binary distributions. To install them, follow the directions available on the website or in the corresponding README files. The second option is to build the distribution package yourself from the source file (you will need a C compiler and a FORTRAN compiler or *f2c* to do that). For detailed instructions, see the manual on R Installation and Administration at http://cran.r-project.org/ and the corresponding FAQ section. In most cases, especially if you are a beginner, it makes more sense to install a precompiled distribution than to build R from scratch.

The basic distribution of R contains many useful functions developed by the R Core Team. There also exist add-on packages developed by statisticians all over the world for specific tasks. It is possible to install an add-on package by typing in the following command in the R console (see next section):

```
> install.packages("cluster")
```

The name of the package is enclosed in the quotation marks. Note that you need to have Internet access. First, R will prompt you to select the nearest CRAN mirror. Next, you will be able to select and install the package. However, you will not be able to access the data and functions from the package yet. First, you need to load it. For example, for the package `cluster`, the command will be as follows:

```
> library(cluster)
```

You will have to load all add-on packages that you need for your work every time you open your R workspace. If you work with the R GUI (Graphical User Interface), it is also possible to install, update and load packages via *Packages* in the main menu of the R console. Managing packages is also very easy with RStudio (see Section 2.5).

This textbook comes with a companion package `Rling`. It contains the datasets that are discussed in the textbook and a few functions. The package can be downloaded from the online platform at http://dx.doi.org/10.1075/z.195.website (file *Rling_1.0.tar.gz*) and installed on your computer by following the instructions provided in the file *read.me*. Next, it should be loaded:

```
> library(Rling)
```

The list of all packages that should be installed and loaded will be given at the beginning of every case study. To be able to access a dataset from an add-on package, use `data()` with the name of the dataset in parentheses, for example:

```
> data(ldt)
```

How to cite R and add-on R packages

It is important that you acknowledge the enormous work of R creators and maintainers by making proper references to the software. To obtain the citation information on your base distribution, type in the following:

```
> citation()

To cite R in publications use:

R Core Team (2015). R: A language and environment for statistical
computing. R Foundation for Statistical Computing, Vienna,
Austria. URL
http://www.R-project.org/.
[output omitted]
```

If you use someone's add-on package for your research, follow the example below:

```
> citation("cluster")

To cite the R package 'cluster' in publications use:

Maechler, M., Rousseeuw, P., Struyf, A., Hubert, M., Hornik,
K.(2015). cluster: Cluster Analysis Basics and Extensions. R
package version 2.0.1.
[output omitted]
```

2.3 First steps with R

2.3.1 Starting R

In Windows, R can be started by clicking on the *R* icon on your desktop (if you have agreed to create it during the setup) or from the *Start* menu. Users of OS X can click on the *R* icon in the *Applications* directory. This will open the R GUI, or R console, shown in Figure 2.1. If you work with Linux/Unix, type *R* to start an R session.

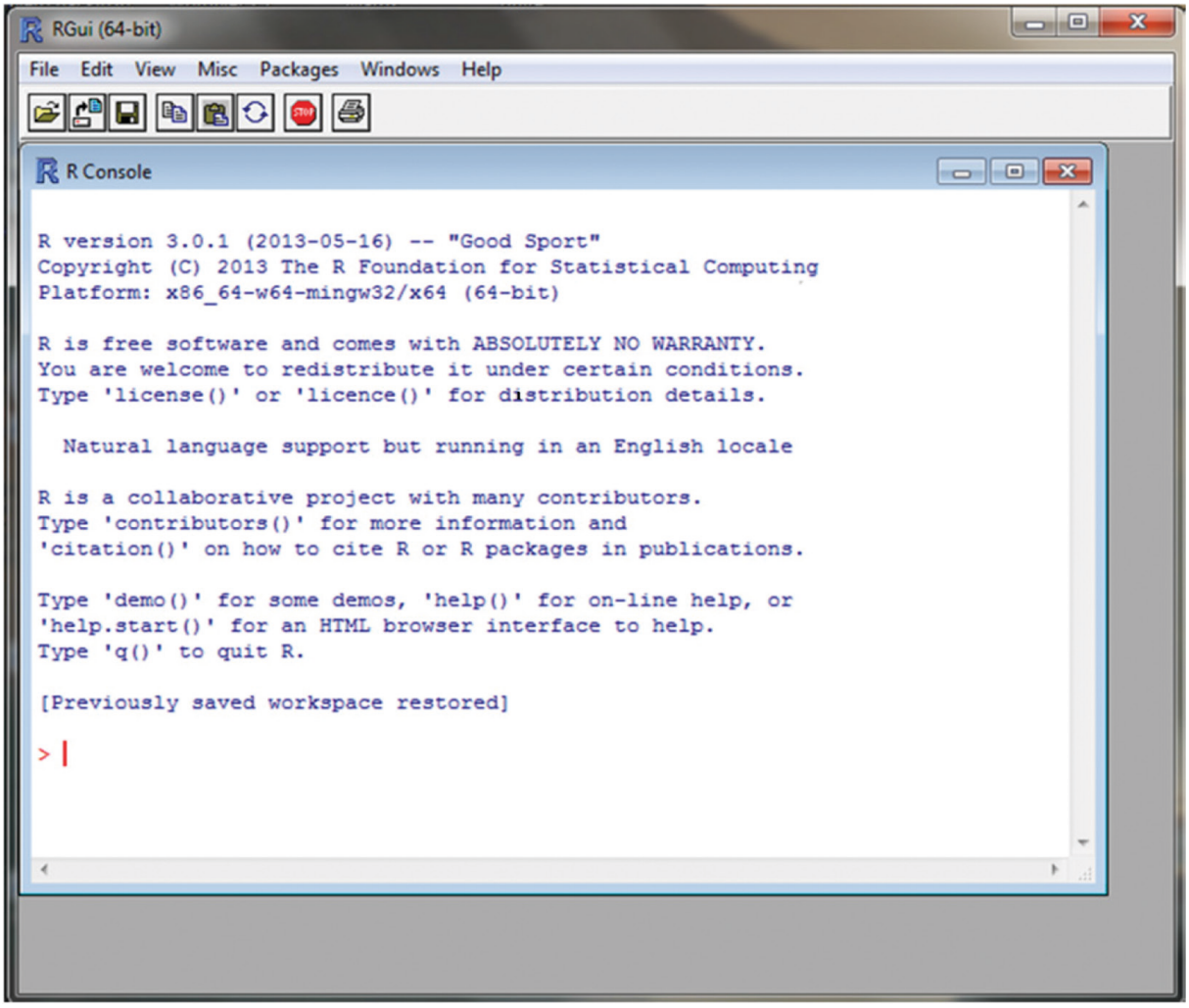


Figure 2.1. A screenshot of the R console

2.3.2 R syntax

After you have started R, you will see some text that gives you information about the R version and some useful commands. The > sign indicates the prompt line where you can type your code. Unlike some other statistical packages, R requires that the user types in his or her commands in the prompt line. Pressing *Enter* will get them executed. For example, one can type in an arithmetic expression and press *Enter* to get the result:

```
> 2 + 2
[1] 4
```

The result appears on the line below the command. The number in square brackets `[1]` shows the index number of the element in the output. This information can be useful when one has a long string of elements that does not fit in one line. For example, if you type in `month.name`, you can get the following output (the actual output on your computer depends on the size of the active screen):

```
> month.name
[1] "January" "February" "March"  "April"      "May"
[6] "June"    "July" "August"     "September" "October"
[11] "November" "December"
```

It is also possible to type several commands on one line. The commands should be separated by a semicolon:

```
> 2+2;4+4
[1] 4
[1] 8
```

Common arithmetic operations in R

Addition	`2 + 2`
Subtraction	`10 - 5`
Multiplication	`5*5`
Division	`10/2`
Raising to the power	`3^4`
Quadratic root	`16^(1/2)` or `sqrt(16)`
Cubic root	`27^(1/3)`
Natural logarithm $ln(x)$ to the base $e \approx 2.178$	`log(10)`
Logarithm to the base 10 $log_{10}(x)$	`log(100, 10)` or `log10(100)`

An example of combining several operations:

```
> -10 + (56 + 76)/12 + 3^4 - 16^(1/2) + 200*0.01
[1] 80
```

(*Continued*)

Do not forget to use parentheses to specify the order of operations where necessary:

```
> 16^(1/4)*(12+68)/(211 + 9)
[1] 0.7272727
```

Importantly, R can store objects created by you. This is convenient when you have complex analyses involving many steps. As a very simple illustration, we will create an object `a` that equals 3:

```
> a <- 3
```

This time R does not give any output. It simply 'remembers' the new object. To call a stored object, simply type its name:

```
> a
[1] 3
```

The new object can be used in other operations:

```
> a + 5
[1] 8
```

It is also possible to assign a value to a variable with the help of =.

```
> a = 3
> a
[1] 3
```

However, in some cases, this may create problems, so it is recommended to use <- for assignment, as shown above.

Double equal sign

In the following chapters you will also see the double equal sign ==. Unlike the single equal sign, which can be used for assignment, the double sign is used for testing if all elements of an object meet a certain logical condition. Consider the following example:

```
> a = 3 # creates an object a with the value 3, an alternative
to a <- 3
> a == 3 # tests if a equals 3
[1] TRUE
```

```
> a == 10 # tests if a equals 10
[1] FALSE
```

You can also perform different operations with the stored objects and combine them with other objects. Be careful with capital and small letters. R interprets them as different symbols.

```
> b <- 7
> a + b
[1] 10
> a + B
Error: object 'B' not found
```

The list of all objects that have been created can be accessed as follows:

```
> ls()
[1] "a"       "b"
```

To remove an object, you can use the `rm()` function:

```
> rm(b)
> ls()
[1] "a"
```

The code below removes ALL objects, so you should be very careful when using it:

```
> rm(list = ls())
```

Such expressions as `sqrt()`, `rm()`, `ls()` are called **functions**. To use them, one can provide arguments in parentheses, e.g. `sqrt(16)`, or use the default ones. Every function has its unique arguments, such as numbers, characters, previously created R objects, other functions, etc. To check which arguments are needed, one can use Help files (see Section 2.3.4).

How to save time typing your R commands

You can always go back to your previous commands by using the *Up* and *Down* arrows on your keyboard. This will make the previous code appear on the command line. Instead of typing in your code again, you can simply use your previous commands. You can also edit them if necessary. It is recommended to copy and paste elaborate code in a separate file, so that it can be easily accessed again.

2.3.3 Exiting from R or terminating a process

In Windows and OS X, you can exit the R GUI by selecting *File* → *Exit* from the main menu, or by clicking on the red cross in the top right corner of the window frame. If you use Linux/Unix, press *Ctrl-D*. In all systems, you can always exit from R by typing in the following command:

```
> q()
```

R will ask you if you want to save your workspace. If you choose this option, R will save your workspace with all user-defined objects to an .RData file. This is the recommended option if you plan to continue working with the created objects in the future. By default, the file will be stored in your current working directory. Note that the previous .RData file will be overwritten. To identify your current working directory, type in

```
> getwd()
[1] "C:/Users/YourName/YourDirectory"
```

When you want to open the saved workspace, you can do so by opening R or by double-clicking on the .RData file in your working directory. If you have several workspaces stored at different locations, you can open the one that you need either by double-clicking on the .RData file in the corresponding directory, or by typing in the path to the file in your R console:

```
> load("C:/Users/YourName/YourDirectory/yourFile.RData")
```

RStudio offers a convenient environment for managing several projects (see Section 2.5). Another useful file that you will find in the working directory after saving your workspace is .Rhistory, which contains the commands that you have entered.

Sometimes one needs to stop a computation process, which has been running for a long time. In this case, you can press *Esc* or click on the red *STOP* sign (Windows or OS X), or press *Ctrl-C* (Linux/Unix). By doing so, you will stop the process without exiting from R.

2.3.4 Getting help

Very often it is necessary to check which arguments are required by a function, or to search for a relevant function. R documentation provides a lot of useful information. For instance, if you want to find out how to use the function `cor()`, which returns a correlation coefficient, you can type in the following command:

```
> help(cor)
```

Alternatively, you can use

```
> ?cor
```

Very often, however, one does not remember or know the name of specific function. Suppose you want to learn how to measure correlation between two variables. In that case, one can use the following code:

```
> help.search("correlation")
```

or, more simply

```
> ??correlation
```

This will return a list of all functions available in your documentation, which contain the expression.

Important guidance is also provided in warnings and error messages. An error message indicates that the command was not successful. For example, the following code did not work because of a typo in the function name:

```
> corr.test(x, y) #do not run; only provided as an example
Error: could not find function "corr.test"
```

In this situation, one can correct the code by typing `cor.test(x, y)`. Unlike errors, warning messages do not mean that the operation was unsuccessful. Instead, they indicate some problematic issues that need to be double-checked. Consider an example with a data object `d`:

```
> chisq.test(d)# do not run
  Pearson's Chi-squared test with Yates' continuity correction

data: d
X-squared = 3.865, df = 1, p-value = 0.0493

Warning message:
In chisq.test(d): Chi-squared approximation may be incorrect
```

The chi-squared (χ^2) test is not robust when at least one expected value in the table is smaller than 5 (see Chapter 9). It is recommended to use another test (e.g. the Fisher exact test) in such cases.

If the documentation is not sufficient or clear, one can look for help in the global R community. The easiest way is googling for specific keywords. The chances are that someone has already been struggling with a similar issue. One can also post a question on mailing lists, e.g. the R-help list. See https://stat.ethz.ch/mailman/listinfo/r-help for further instructions.

2.4 Main types of R objects

Uni-, bi- and multivariate data can be represented in R by different objects. Single variables are most commonly represented by vectors and factors. **Vectors** can be of two main types:

numeric (sequences of numbers) and character (sequences of character strings). Below is a numeric vector called `vnum`, a sequence of integers from 1 to 5, which was generated with the help of the colon operator.

```
> vnum <- 1:5
> vnum
[1] 1 2 3 4 5
```

Another way of creating a vector is by using the function `c()`, which combines individual elements:

```
> fibonacci10 <- c(1, 1, 2, 3, 5, 8, 13, 21, 34, 55)
> fibonacci10
[1] 1 1 2 3 5 8 13 21 34 55
```

The resulting vector is a sequence of ten Fibonacci numbers, where each subsequent number is the sum of the previous two.

Now consider an example of a character vector, a sequence of words in Gertrude Stein's famous sentence:

```
> stein <- c("a", "rose", "is", "a", "rose", "is", "a", "rose")
> stein
[1] "a"      "rose" "is"   "a"    "rose" "is"   "a"    "rose"
```

Character strings should be enclosed in quotation marks. It does not matter whether you use double or single quotation marks in R, as long as you use the same type in the beginning and at the end of your character string.

Character vectors will be rarely used in this book, however. Nominal variables are best represented by **factors**. One can create a factor from a character vector as follows:

```
> stein.fac <- factor(stein)
> stein.fac
[1] a rose is a      rose is       a      rose
Levels: a is rose
```

The factor levels represent the categories `a`, `is` and `rose`. Note that the levels are arranged alphabetically by default.

The frequencies of factor levels can be represented in a **table**:

```
> table(stein.fac)
stein.fac
   a  is  rose
   3  2   3
```

The result of this operation is a one-dimensional table with frequencies of each level. In the corpus linguistics terminology, these are the **token frequencies** of the words in this small text, whereas the factor levels (words) represent **types**.

Several numeric vectors can be combined in a **matrix** (mathematically speaking, a two-dimensional array). Consider an example:

```
> m <- cbind(1:5, 10:6)
> m
     [,1] [,2]
[1,] 1    10
[2,] 2    9
[3,] 3    8
[4,] 4    7
[5,] 5    6
```

The function `cbind()` combines two or more vectors or factors as columns. Its counterpart is `rbind()`, which combines different rows.

In real research, data are often represented as **data frames**. A data frame is a list of vectors and/or factors of various types, or other objects. It is usually displayed as a table with rows, which normally correspond to individual cases, and columns, which represent variables. Below is an imaginary example of experimental data. The cases (rows) are four participants in an experiment, and the variables (sex and reaction time in milliseconds), are columns. The data frame is constructed from two vectors, one of which is a character vector with information about the participants' sex, and the other is a numeric vector with reaction times:

```
> sex <- c("f", "m", "m", "f")
> sex
[1] "f" "m" "m" "f"

> rt <- c(455, 773, 512, 667)
> rt
[1] 455 773 512 667

> df <- data.frame(sex, rt)
> df
  sex  rt
1  f   455
2  m   773
3  m   512
4  f   667
```

When building a data frame, R automatically turns a character vector (`sex`) into a factor, as can be seen from the output of `str()`, which can display the internal structure of any R object.

```
> str(df)
'data.frame':       4 obs. Of 2 variables:
$ sex: Factor w/ 2 levels "f","m": 1 2 2 1
$ rt: num 455 773 512 667
```

R offers many convenient tools to explore, edit and subset data frames. For example, one can easily select a column in a data frame as follows:

```
> df$sex
[1] f m m f
Levels: f m
```

For more information on how to manipulate these and other R objects, see Appendix 1. Finally, if you do not know which type an object belongs to, use the function `is()`, for example:

```
> is(m)
[1] "matrix"  "array"       "structure" "vector"
```

2.5 RStudio

RStudio is an integrated environment for R. It can run both locally and on a web server. It is very convenient for managing one's numerous research projects. RStudio enables one to easily access the R console, R code, data, plots, packages, etc., which are represented in separate panels and tabs (see Figure 2.2). You can view and edit your data, execute your code directly from the code editor, search in your R history, and do many other useful things. RStudio is freely downloadable from http://www.rstudio.com/ and available for different platforms.

2.6 Importing and exporting your data and saving your graphs

2.6.1 Importing your data to R

When doing your own research, it is crucial to be able to import and export your data to and from R. Many linguists store their datasets in Excel or similar spreadsheets. Consider an example, a fictitious dataset in Figure 2.3. The dataset contains a list of imaginary subjects, their sex, dialect (American or British English) and their reaction times in an experiment (in milliseconds). The first row contains the names of variables.

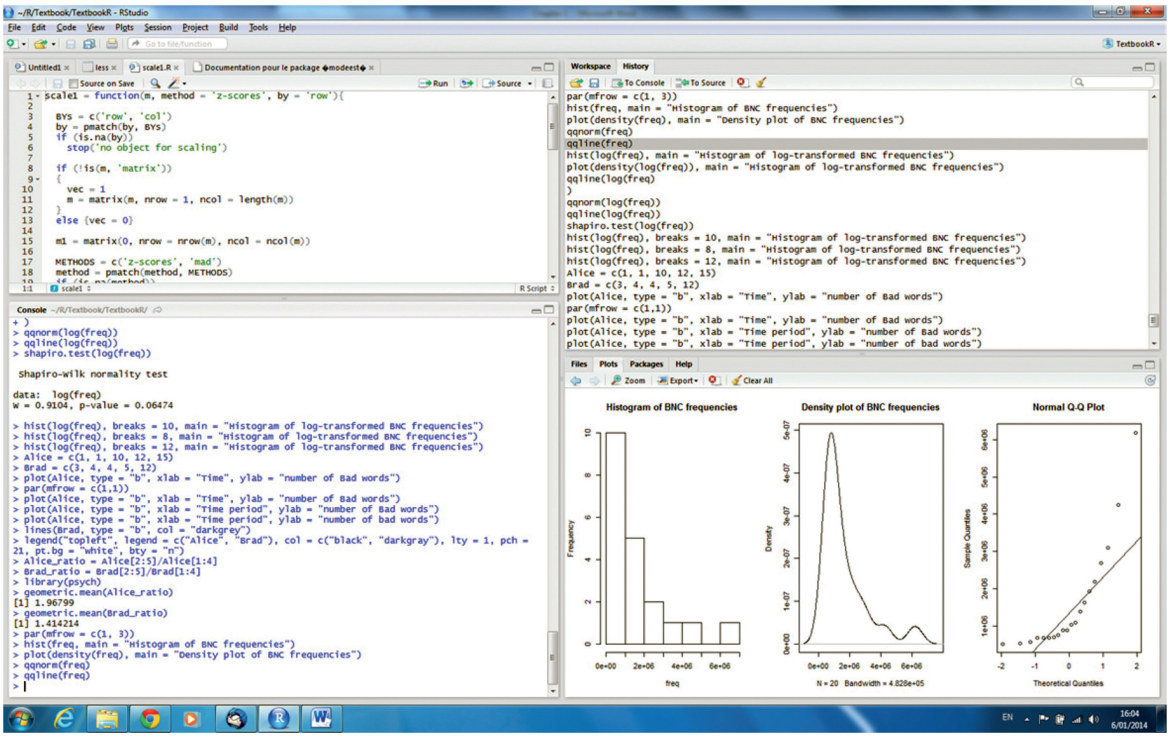
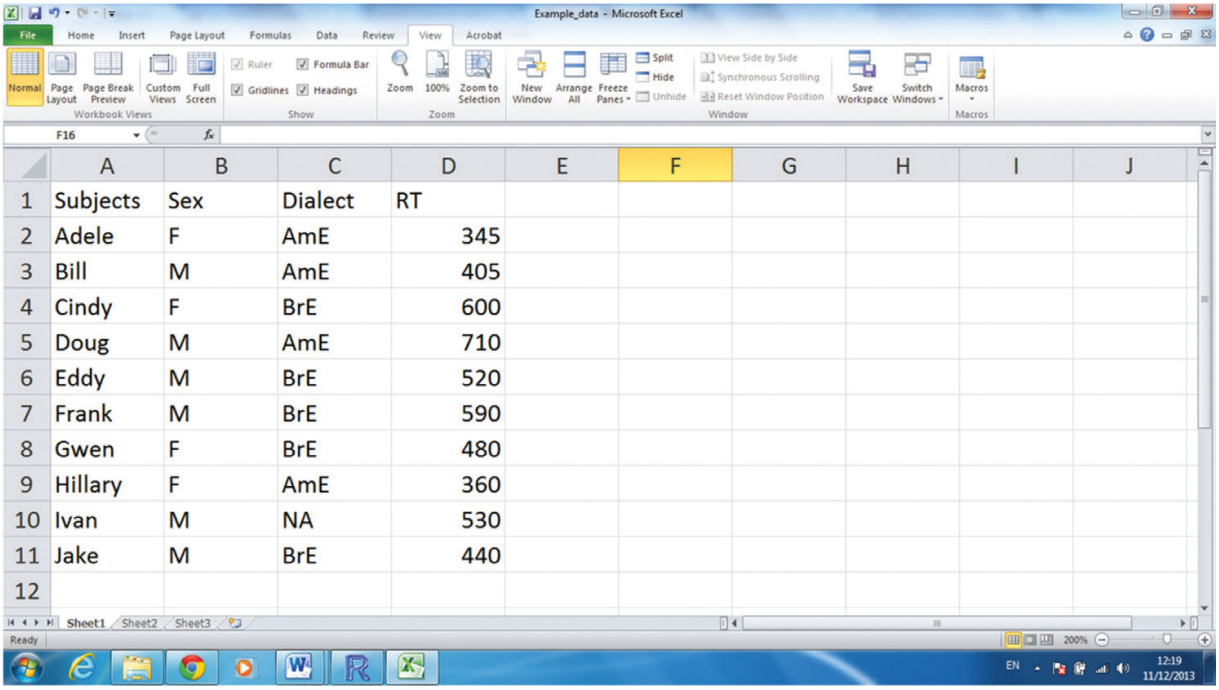


Figure 2.2. A screenshot of RStudio



	A	B	C	D
1	Subjects	Sex	Dialect	RT
2	Adele	F	AmE	345
3	Bill	M	AmE	405
4	Cindy	F	BrE	600
5	Doug	M	AmE	710
6	Eddy	M	BrE	520
7	Frank	M	BrE	590
8	Gwen	F	BrE	480
9	Hillary	F	AmE	360
10	Ivan	M	NA	530
11	Jake	M	BrE	440
12				

Figure 2.3. A fictitious dataset in an Excel spreadsheet

It is important that your table should not contain any empty cells. Note that Ivan has a missing value for *Dialect*. Missing values should be coded as "NA".

You can save your spreadsheet in several formats by using *File* → *Save As*. The most popular ones are the tab delimited text file (.txt) and the comma-separated values file (.csv). Let us assume you have used both options and created two files, *Example_data_tab.txt* and *Example_data_csv.csv*. See the screenshots in Figure 2.4. Note that the.csv file may also contain semicolons rather than commas, depending on the spreadsheet software you are using.

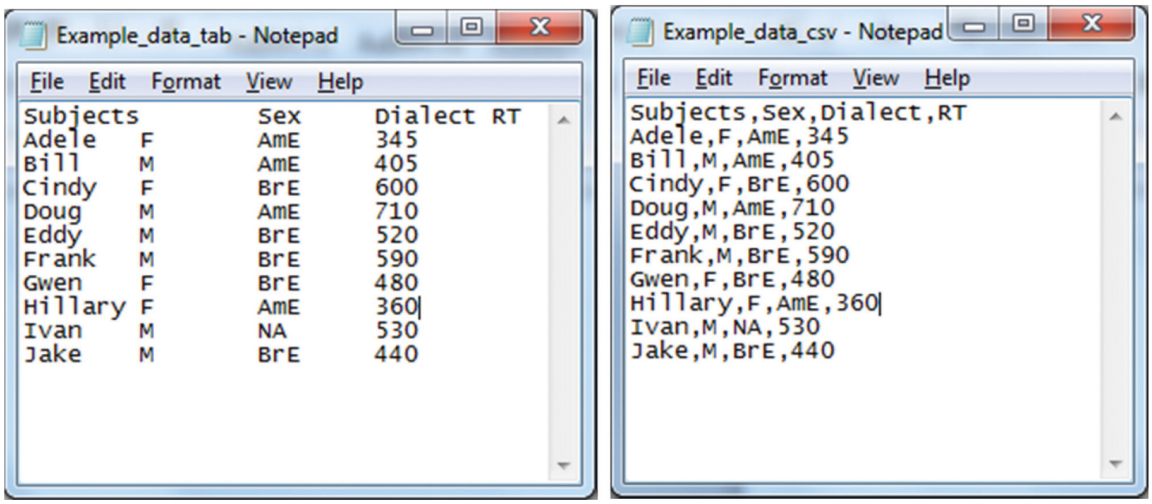


Figure 2.4. Screenshots of the files with tab-separated (left) and comma-separated values data (right)

Now you can import the data to R with the help of `read.table()` or `read.csv()`. You will have to specify the path (if you save the file in your working directory, then you can only provide the file name). For the tab-delimited file, the code will be as follows:

```
> data_tab <- read.table("C:/Your/Directory/Example_data_tab.txt",
header = TRUE)
```

The expression `header = TRUE` tells R to treat the first line as the names of variables.

For Windows users: Beware the backslash!

If you are a Windows user and you have to specify a file location in some directory, you should avoid using backslashes. For instance, if your path is

`C:\My\Directory\mydata.txt`,

you should use either double backslashes, as below:

`"C:\\My\\Directory\\mydata.txt"`,

or single forward slashes:

`"C:/My/Directory/mydata.txt"`

The function `head()` enables you to see the first six elements (here, rows in the table). R also provides names for rows: here, the row names are numbers from 1 to 10 (only the first six are shown).

```
> head(data_tab)
  Subjects  Sex  Dialect  RT
1 Adele     F    AmE      345
2 Bill      M    AmE      405
3 Cindy     F    BrE      600
4 Doug      M    AmE      710
5 Eddy      M    BrE      520
6 Frank     M    BrE      590
```

Let us use `str()` to examine the structure of the data frame:

```
> str(data_tab)
'data.frame': 10 obs. Of 4 variables:
$ Subjects:   Factor w/ 10 levels "Adele","Bill",..: 1 2 3 4 5 6 7
              8 9 10
$ Sex:        Factor w/ 2 levels "F","M": 1 2 1 2 2 2 1 1 2 2
$ Dialect:    Factor w/ 2 levels "AmE","BrE": 1 1 2 1 2 2 2 1 NA 2
$ RT:         int 345 405 600 710 520 590 480 360 530 440
```

Again, you can see that R has interpreted the subjects' names, sex and dialect variables as factors.

Non-ASCII characters in data frames

Using non-ASCII characters in R may be problematic. Consider an example. You collect experimental data about the linguistic development of several Russian children. Their names are written in Russian (the first column in your dataset). You save the data as a tab-delimited text file in Unicode, or UTF–8. When you import the data in R, you should specify the encoding:

```
> rus <- read.table("rus.txt", encoding = "UTF-8")# do not run
```

However, the data may be displayed unintelligibly when you try to look at the entire dataset:

```
> rus # do not run
V1 V2
1                      ⟨U+FEFF⟩⟨U+0410⟩⟨U+043D⟩⟨U+044F⟩ 28
2                      ⟨U+0412⟩⟨U+0430⟩⟨U+043D⟩⟨U+044F⟩ 16
3                      ⟨U+041C⟩⟨U+0438⟩⟨U+0448⟩⟨U+0430⟩ 22
4       <U+041D⟩⟨U+0430⟩⟨U+0442⟩⟨U+0430⟩⟨U+0448⟩⟨U+0430⟩ 31
5                      ⟨U+041F⟩⟨U+0435⟩⟨U+0442⟩⟨U+044F⟩ 30
6                             ⟨U+042F⟩⟨U+043D⟩⟨U+0430⟩ 25
```

Yet, if you look only at one variable, you can see the values (the children's names):

```
> rus[, 1] # do not run
[1] Аня      Ваня   Миша   Наташа Петя   Яна
Levels: Аня Ваня Миша Наташа Петя Яна
```

You can also see the original data if you transform the data into a matrix:

```
> as.matrix(rus) # do not run
      V1         V2
[1,]  "Аня"      "28"
[2,]  "Ваня"     "16"
[3,]  "Миша"     "22"
[4,]  "Наташа"   "31"
[5,]  "Петя"     "30"
[6,]  "Яна"      "25"
```

The output may also depend on the local settings of your OS. Unfortunately, data frames are easy to use if you have ASCII characters only.

To open the comma-separated file, use `read.csv()` instead of `read.table()`. In that case, you do not have to specify `header = TRUE` because it is the default option. If the file is saved with the semicolon as the delimiter, you can use `read.csv2()`.

Dealing with spaces in your data

In case you have a tab-delimited file and you have spaces in some of the values, you should tell R to use the tab as a delimiter; otherwise it will treat spaces as delimiters, as well. Let us add a new line to the data with the following values (shown with formatting signs):

Subjects → Sex → Dialect → RT¶
Kate·Emma → F → BrE → 570¶

"Kate Emma" is a double name written with a space, and it represents one subject in the experiment. If you now save the dataset as a tab-separated file *Example_data_space.txt* and try to read it in R, you will get an error message:

```
> data_space <- read.table("C:/Your/Directory/Example_data_
space.txt", header = TRUE)
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines,
na.strings,:
line 11 did not have 4 elements
```

(*Continued*)

This happens because line 11 has five elements instead of the expected four. To fix this, add `sep = "\t"` (the default separator is any 'white space', that is, one or more spaces, tabs, newlines or carriage returns):

```
> data_space <- read.table("C:/Your/Directory/Example_data_space.txt", header = TRUE, sep = "\t")
```

2.6.2 Exporting your data from R

After you have created or edited your dataset in R, you might want to export it. To export a data frame, you can use the function `write.table()`:

```
> write.table(data_tab, file = "C:/Your/Directory/Exported.txt", quote = FALSE, sep = "\t", row.names = FALSE)
```

The first argument is the name of the data frame. It is followed by the path where you want to save the file. The next argument is `quote = FALSE`, which tells R not to put the character strings or factor values in quotation marks (it does so by default). The code also says that the field separator is the tab character (the default separator is white space). Finally, `row.names = FALSE` tells R not to save the row names (numbers from 1 to 10). In this example, they are not very informative. Of course, if the row names are important, you should keep them. In that case, you should simply remove `row.names = FALSE` from the code. The default is to save the row names, as well as the column names. The new file can be easily opened in Excel or a similar application. See more options on the help page of `write.table()`. If you want to save your data in the comma-separated values format, you can use `write.csv()`.

Choosing files interactively

If you find it inconvenient to type long paths, you can choose files interactively with the help of `file = file.choose()`, e.g. `read.table(file = file.choose())`. You will see a file selection window, where you can choose the location and specify the name of the file.

2.6.3 Saving your graphs

R offers many functions that can be used for making graphs. After you have entered the code with those functions, an R graphical device window will be opened. If you use the R GUI, the easiest way to save your graphical output is to select *File* → *Save As* from the main menu and then choose the desired format. Another way of doing it, which is also available on Unix-like systems, would be as follows:

```
> png("C:/My/Directory/myplot.png") # opens a plot device. You
should specify the path and the name of the file. Alternatively, use
file = file.choose(), as shown in the previous section.
> plot(1:5, 11:15) # makes a scatter plot
> dev.off()# closes the graphics device
null device
          1
```

You can also use `jpeg()`, `tiff()`, `bmp()`, as well as `pdf()` to save your output in a specific format. See `help(jpeg)` for more information about how to control different graphical parameters, e.g. the size of the plot (arguments `width` and `height`) and background colour (`bg`). RStudio offers a convenient interface for copying and saving plots in different formats and sizes.

Dealing with other data formats

- If you have tabular or CSV data from the Web, HTML tables and MySQL databases, see Teetor (2011) for guidelines.
- To learn how one can read and write individual vectors, factors, lists and matrices, see the help pages of `scan()` and `cat()` or `write()`. See also Gries (2013: 69–71, 80–81).
- If you deal with more complex data structures, see Recipe 4.12 in Teetor (2011).
- If you want to store an R object for later use, or to transfer it from one computer to another, see Recipe 4.14 in Teetor (2011), as well as the help pages of `load()` and `save()`.

2.7 Summary

In this chapter you have learnt how to install the basic R distribution and add-on packages. We have discussed some operations with the main data types in R. You have also learnt how to import and export your own datasets to and from R. For further analyses discussed in this book, the author recommends that you use RStudio, although this is not necessary. You should be able now to install and load the add-on packages required for the subsequent case studies, including the companion package `Rling`. Now that you are equipped with the basic principles and tools, we can begin doing linguistics with R.

More on R basics

You can find more information on R basics and operations with various R objects in Crawley (2007: Ch. 1–7). Teetor's (2011) *R Cookbook* is a convenient task-based reference book. See also Appendix 1 for more operations with basic R objects.