1 An introduction to R

In order to learn to work with R, you have to learn to speak its language, the S language, developed originally at Bell Laboratories (Becker *et al.*, 1988). The grammar of this programming language is beautiful and easy to learn. It is important to master its basics, as this grammar is designed to guide you towards the appropriate way of thinking about your data and how you might want to carry out your analysis.

When you begin to use R on an Apple Macintosh or a Windows PC, you will start R either through a menu guiding you to applications, or by clicking on R's icon. As a result, a graphical user interface is started up, with as its central part a window with a prompt (>), the place where you type your commands. On UNIX or LINUX systems, the same window is obtained by opening a terminal and typing R at its prompt.

The sequence of commands in a given R session and the objects created are stored in files named `.Rhistory` and `.RData` when you quit R and respond positively to the question of whether you want to save your workspace. If you do so, then your results will be available to you the next time you start up R. If you are using a graphical user interface, this `.RData` file will be located by default in the folder where R has been installed. In UNIX and LINUX, the `.RData` file will be created in the same directory as where R was started up.

You will often want to use R for different projects, located in different directories on your computer. On UNIX and LINUX systems, simply open a terminal in the desired directory, and start R. When using a graphical user interface, you have to use the `File` drop-down menu. In order to change to another directory, select `Change dir`. You will also have to load the `.RData` and `.Rhistory` using the options `Load Workspace` and `Load History`.

Once R is up and running, you need to install a series of packages, including the package that comes with this book, `languageR`. This is accomplished with the following instruction, to be typed at the R prompt:

```
install.packages(c("rpart", "chron", "Hmisc", "Design",
"Matrix", "lme4", "coda", "e1071", "zipfR", "ape",
"languageR"), repos = "http://cran.r-project.org")
```

Packages are installed in a folder named `library`, which itself is located in R's home directory. On my system, R's home is `/home/harald/R-2.4.0`, so packages are found in `/home/harald/R-2.4.0/library`, and the code of the

main examples in this book is located in `/home/harald/R-2.4.0/library/languageR/scripts`.

I recommend that you create a file named `.Rprofile` in your home directory. This file should contain the line,

```
library(languageR)
```

telling R that upon startup it should attach `languageR`. All data sets and functions defined in `languageR`, and some of the packages that we will need, will be automatically available. Alternatively, you can type `library(languageR)` at the R prompt yourself after you have started R. All examples in this book assume that the `languageR` package has been attached.

The way to learn a language is to start speaking it. The way to learn R, and the S language that it is built on, is to start using it. Reading through the examples in this chapter is not enough to become a confident user of R. For this, you need to actually try out the examples by typing them at the R prompt. You have to be very precise in your commands, which requires a discipline that you will master only if you learn from experience, from your mistakes and typos. Don't be put off if R complains about your initial attempts to use it, just carefully compare what you typed, letter by letter and bracket by bracket, with the code in the examples.

If you type a command that extends over separate lines, the standard prompt > will change into the special continuation prompt +. If you think your command is completed, but still have a continuation prompt, there is something wrong with your syntax. To cancel the command, use either the escape key, or hit CONTROL-C. Appendix B provides an overview of operators and functions, grouped by topic, that you may find useful as a complement to the example-by-example approach followed in the main text of this book.

1.1 R as a calculator

Once you have an R window, you can use R simply as a calculator. To add 1 and 2, type,

```
> 1 + 2
```

and hit the RETURN (ENTER) key, and R will display:

```
[1] 3
```

The `[1]` preceding the answer indicates that 3 is the first element of the answer. In this example, it is also the only element. Other examples of arithmetic operations are:

```
> 2 * 3                    # multiplication
[1] 6
> 6 / 3                    # division
[1] 2
> 2 ^ 3                    # power
```

```
[1] 8
> 9 ^ 0.5                 # square root
[1] 3
```

The hash mark # indicates that the text to its right is a comment that should be ignored by R. Operators can be stacked, in which case it may be necessary to make explicit by means of parentheses the order in which the operations have to be carried out:

```
> 9 ^ 0.5 ^ 3
[1] 1.316074
> (9 ^ 0.5) ^ 3
[1] 27
> 9 ^ (0.5 ^ 3)
[1] 1.316074
```

Note that the evaluation of exponentiation proceeds from right to left, rather than from left to right. Use parentheses whenever you are not absolutely sure about the order in which R evaluates stacked operators.

The results of calculations can be saved and referenced by VARIABLES. For instance, we can store the result of adding 1 and 2 in a variable named x. There are three ways in which we can assign the result of our addition to x. We can use the equals sign as assignment operator,

```
> x = 1 + 2
> x
[1] 3
```

or we can use a left arrow (composed of < and -) or a right arrow (composed of - and >, as follows:

```
> x <- 1 + 2
> 1 + 2 -> x
```

The right arrow is especially useful in cases where you have typed a long expression and only then decide that you would like to save its output rather than have it displayed on your screen. Instead of having to go back to the beginning of the line, you can continue typing and use the right arrow as assignment operator. We can modify the value of x, for instance, by increasing its value by one:

```
> x = x + 1
```

Here we take x, add one, and assign the result (4) back to x. Without this explicit assignment, the value of x remains unchanged:

```
> x = 3
> x + 1   # result is displayed, not assigned to x
[1] 4
> x       # so x is unchanged
[1] 3
```

We can work with variables in the same way that we work with numbers:

```
> 4 ^ 3
[1] 64
> x = 4
```

```
> y = 3
> x ^ y
[1] 64
```

The more common mathematical operations are carried out with operators such as `+`, `-`, and `*`. For a range of standard operations, as well as for more complex mathematical calculations, a wide range of functions is available. Functions are commands that take some input, do something with that input, and return the result to the user. Above, we calculated the square root of 9 with the help of the `^` operator. Another way of obtaining the same result is by means of the `sqrt()` function:

```
> sqrt(9)
[1] 3
```

The argument of the square root function, 9, is enclosed between parentheses.

1.2 Getting data into and out of R

Bresnan *et al.* (2007) studied the dative alternation in English in the three-million-word Switchboard collection of recorded telephone conversations and in the Treebank *Wall Street Journal* collection of news and financial reportage. In English, the recipient can be realized either as an NP (*Mary gave John the book*) or as a PP (*Mary gave the book to John*). Bresnan and colleagues were interested in predicting the realization of the recipient (as NP or PP) from a wide range of potential explanatory variables, such as the animacy, the length in words, and the pronominality of the theme and the recipient. A subset of their data collected from the Treebank is available as the data set `verbs`. (Bresnan and colleagues studied many more variables, the full data set is available as `dative`, and we will study it in detail in later chapters.) You should have attached the `languageR` package at this point, otherwise `verbs` will not be available to you.

We display the first 10 rows of the `verbs` data with the help of the function `head()`. (Readers familiar with programming languages like C and Python should note that `R` numbering begins with 1 rather than with zero.)

```
> head(verbs, n = 10)
RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
1               NP  feed      animate      inanimate     2.6390573
2               NP  give      animate      inanimate     1.0986123
3               NP  give      animate      inanimate     2.5649494
4               NP  give      animate      inanimate     1.6094379
5               NP offer      animate      inanimate     1.0986123
6               NP  give      animate      inanimate     1.3862944
7               NP   pay      animate      inanimate     1.3862944
8               NP bring      animate      inanimate     0.0000000
9               NP teach      animate      inanimate     2.3978953
10              NP  give      animate      inanimate     0.6931472
```

When the option *n* is left unspecified, the first 6 rows will be displayed by default. Tables such as exemplified by `verbs` are referred to in R as DATA FRAMES. Each line in this data frame represents a clause with a recipient, and specifies whether this recipient was realized as an NP or as a PP. Each line also lists the verb used, the animacy of the recipient, the animacy of the theme, and the logarithm of the length of the theme. Note that each elementary observation — here the realization of the recipient as NP or PP in a given clause — has its own line in the input file. This is referred to as the LONG DATA FORMAT, where *long* highlights that no attempt is made to store the data more economically.

It is good practice to spell out the elements in the columns of a data frame with sensible names. For instance, the first line with data specifies that the recipient was realized as an NP for the verb *to feed*, that the recipient was animate, and that the theme was inanimate. The length of the theme is listed in log units, for reasons that will become clear in later chapters. The actual length of the theme is 14, as shown when we undo the logarithmic transformation with its inverse, the exponential function `exp()`:

```
> exp(2.6390573)
[1] 14
> log(14)
[1] 2.639057
```

A data frame such as `verbs` can be saved outside R as an independent file with `write.table()`, enclosing the name of the file (including its path) between double quotes:

```
> write.table(verbs, file = "/home/harald/dativeS.txt")  # Linux
> write.table(verbs, file = "/users/harald/dativeS.txt") # MacOSX
> write.table(verbs, file = "c:stats/dativeS.txt")       # Windows
```

Users of Windows should note the use of the forward slash for path specification. Alternatively, on MacOS X or Windows, the function `file.choose()` may be used, replacing the file name, in which case a dialog box is provided.

External data in this tabular format can be loaded into R with `read.table()`. We tell this function that the file we just made has an initial line, its *header*, that specifies the column names:

```
> verbs = read.table("/home/harald/dativeS.txt", header = TRUE)
```

R handles various other data formats as well, including `sas.get()` (which converts SAS data sets), `read.csv()` (which handles comma-separated spreadsheet data), and `read.spss()` (for reading SPSS data files).

Data sets and functions in R come with extensive documentation, including examples. This documentation is accessed by means of the `help()` function. Many examples in the documentation can be also executed with the `example()` function:

```
> help(verbs)
> example(verbs)
```

1.3 Accessing information in data frames

When working with data frames, we often need to select or manipulate subsets of rows and columns. Rows and columns are selected by means of a mechanism referred to as subscripting. In its simplest form, subscripting can be achieved simply by specifying the row and column numbers between square brackets, separated by a comma. For instance, to extract the length of the theme for the first line in the data frame `verbs`, we type:

```
> verbs[1, 5]
[1] 2.639057
```

Whatever precedes the comma is interpreted as a restriction on the rows, and whatever follows the comma is a restriction on the columns. In this example, the restrictions are so narrow that only one element is selected, the one element that satisfies the restrictions that it should be on row 1 and in column 5. The other extreme is no restrictions whatsoever, as when we type the name of the data frame at the prompt, which is equivalent to typing:

```
> verbs[ , ]        # this will display all 903 rows of verbs!
```

When we leave the slot before the comma empty, we impose no restrictions on the rows:

```
> verbs[ , 5]                      # show the elements of column 5
 [1] 2.6390573 1.0986123 2.5649494 1.6094379 1.0986123
 [6] 1.3862944 1.3862944 0.0000000 2.3978953 0.6931472
 ...
```

As there are 903 rows in `verbs`, the request to display the fifth column results in an ordered sequence of 903 elements. In what follows, we refer to such an ordered sequence as a `vector`. Thanks to the numbers in square brackets in the output, we can easily see that 0.00 is the eighth element of the vector. Column vectors can also be extracted with the `$` operator preceding the name of the relevant column:

```
> verbs$LengthOfTheme                  # same as verbs[, 5]
```

When we specify a row number but leave the slot after the comma empty, we impose no restrictions on the columns, and therefore obtain a row vector instead of a column vector:

```
> verbs[1, ]                         # show the elements of row 1
  RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
1               NP feed      animate      inanimate      2.639057
```

Note that the elements of this row vector are displayed together with the column names.

Row and column vectors can be extracted from a data frame and assigned to separate variables:

```
> row1 = verbs[1,]
> col5 = verbs[ , 5]
> head(col5, n = 5)
 [1] 2.6390573 1.0986123 2.5649494 1.6094379 1.0986123
```

Individual elements can be accessed from these vectors by the same subscripting mechanism, but simplified to just one index between the square brackets:

```
> row1[1]
  RealizationOfRec
1               NP
> col5[1]
[1] 2.639057
```

Because the row vector has names, we can also address its elements by name, properly enclosed between double quotes:

```
> row1["RealizationOfRec"]
  RealizationOfRec
1               NP
```

You now know how to extract single elements, rows, and columns from data frames, and how to access individual elements from vectors. However, we often need to access more than one row or more than one column simultaneously. R makes this possible by placing vectors before or after the comma when subscripting the data frame, instead of single elements. (For R, single elements are actually vectors with only one element.) Therefore, it is useful to know how to create your own vectors from scratch. The simplest way of creating a vector is to combine elements with the concatenation operator `c()`. In the following example, we select some arbitrary row numbers that we save in the variable `rs` (shorthand for rows):

```
> rs = c(638, 799, 390, 569, 567)
> rs
 [1] 638 799 390 569 567
```

We can now use this vector of numbers to select precisely those rows from `verbs` that have the row numbers specified in `rs`. We do so by inserting `rs` before the comma:

```
> verbs[rs, ]
    RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
638               PP  pay      animate      inanimate     0.6931472
799               PP sell      animate      inanimate     1.3862944
390               NP lend      animate        animate     0.6931472
569               PP sell      animate      inanimate     1.6094379
567               PP send    inanimate      inanimate     1.3862944
```

Note that the appropriate rows of `verbs` appear in exactly the same order as specified in `rs`.

The combination operator `c()` is not the only function for creating vectors. Of the many other possibilities, the colon operator should be mentioned here. This

operator brings into existence sequences of increasing or decreasing numbers with a stepsize of one:

```
> 1 : 5
[1] 1 2 3 4 5
> 5 : 1
[1] 5 4 3 2 1
```

In order to select from `verbs` the rows specified by `rs` and the first three columns, we specify the row condition before the comma and the column condition after the comma:

```
> verbs[rs, 1:3]
    RealizationOfRec  Verb AnimacyOfRec
638               PP  pay      animate
799               PP sell      animate
390               NP lend      animate
569               PP sell      animate
567               PP send    inanimate
```

Alternatively, we could have specified a vector of column names instead of column numbers:

```
> verbs[rs, c("RealizationOfRec", "Verb", "AnimacyOfRec")]
```

Note once more that when strings are brought together into a vector, they must be enclosed between quotes.

Thus far, we have selected rows by explicitly specifying their row numbers. Often, we do not have this information available. For instance, suppose we are interested in those observations for which the `AnimacyOfTheme` has the value `animate`. We do not know the row numbers of these observations. Fortunately, we do not need them either, because we can impose a condition on the rows of the data frame such that only those rows will be selected that meet that condition. The condition that we want to impose is that the value in the column of `AnimacyOfTheme` is `animate`. Since this is a condition on rows, it precedes the comma:

```
> verbs[verbs$AnimacyOfTheme == "animate", ]
    RealizationOfRec  Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
58                NP  give      animate        animate     1.0986123
100               NP  give      animate        animate     2.8903718
143               NP  give    inanimate        animate     2.6390573
390               NP  lend      animate        animate     0.6931472
506               NP  give      animate        animate     1.9459101
736               PP trade      animate        animate     1.6094379
```

This is equivalent to:

```
> subset(verbs, AnimacyOfTheme == "animate")
```

It is important to note that the equality in the condition is expressed with a double equal sign. This is because the single equal sign is the assignment operator. The following example illustrates a more complex condition with the logical operator

AND (&) (the logical operator for OR is |):

```
> verbs[verbs$AnimacyOfTheme == "animate" & verbs$LengthOfTheme > 2, ]
    RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
100               NP give      animate        animate      2.890372
143               NP give    inanimate        animate      2.639057
```

Row and column names of a data frame can be extracted with the functions `rownames()` and `colnames()`:

```
> head(rownames(verbs))
 [1] "1"   "2"   "3"   "4"   "5"   "6"
> colnames(verbs)
[1] "RealizationOfRec" "Verb"  "AnimacyOfRec" "AnimacyOfTheme"
[5] "LengthOfTheme"
```

The vector of column names is a string vector. Perhaps surprisingly, the vector of row names is also a string vector. To see why this is useful, we assign the subtable of `verbs` obtained by subscripting the rows with the `rs` vector to a separate object that we name `verbs.rs`:

```
> verbs.rs = verbs[rs, ]
```

We can extract the first line not only by row number,

```
> verbs.rs[1, ]
    RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
638               PP  pay      animate      inanimate     0.6931472
```

but also by row name:

```
> verbs.rs["638",]     # same output
```

The row name is a string that reminds us of the original row number in the data frame from which `verbs.rs` was extracted:

```
> verbs[638, ]         # same output again
```

Let's finally extract a column that does not consist of numbers, such as the column specifying the animacy of the recipient:

```
> verbs.rs$AnimacyOfRec
 [1] animate animate animate animate inanimate
Levels: animate inanimate
```

Two things are noteworthy. First, the words *animate* and *inanimate* are not enclosed between quotes. Second, the last line of the output mentions that there are two LEVELS: `animate` and `inanimate`. Whereas the row and column names are vectors of strings, non-numerical columns in a data frame are automatically converted by R into FACTORS. In statistics, a factor is a non-numerical predictor or response. Its values are referred to as its levels. Here, the factor `AnimacyOfRec` has as its only possible values `animate` and `inanimate`, hence it has only two levels. Most statistical techniques don't work with string vectors, but with factors. This is the reason why R automatically converts non-numerical columns into factors. If you really want to work with a string vector

instead of a factor, you have to do the back-conversion yourself with the function `as.character()`:

```
> verbs.rs$AnimacyOfRec = as.character(verbs.rs$AnimacyOfRec)
> verbs.rs$AnimacyOfRec
 [1] "animate" "animate" "animate" "animate" "inanimate"
```

Now the elements of the vector are strings, and as such properly enclosed between quotes. We can undo this conversion with `as.factor()`:

```
> verbs.rs$AnimacyOfRec = as.factor(verbs.rs$AnimacyOfRec)
```

If we repeat these steps, but with a smaller subset of the data in which `AnimacyOfRec` is only realized as animate,

```
> verbs.rs2 = verbs[c(638, 390), ]
> verbs.rs2
    RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
638               PP  pay      animate      inanimate     0.6931472
390               NP lend      animate        animate     0.6931472
```

we observe that the original two levels of `AnimacyOfRec` are remembered:

```
> verbs.rs2$AnimacyOfRec
[1] animate animate
Levels: animate inanimate
```

In order to get rid of the uninstantiated factor level, we convert `AnimacyOfRec` to a character vector, and then convert it back to a factor:

```
> as.factor(as.character(verbs.rs2$AnimacyOfRec))
[1] animate animate
Levels: animate
```

An alternative with the same result is:

```
> verbs.rs2$AnimacyOfRec[drop=TRUE]
```

1.4 Operations on data frames

1.4.1 Sorting a data frame by one or more columns

In the previous section, we created the data frame `verbs.rs`, the rows of which appeared in the arbitrary order specified by our vector of row numbers `rs`. It is often useful to sort the entries in a data frame by the values in one of the columns, for instance, by the realization of the recipient,

```
> verbs.rs[order(verbs.rs$RealizationOfRec), ]
    RealizationOfRec Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
390               NP lend      animate        animate     0.6931472
638               PP  pay      animate      inanimate     0.6931472
799               PP sell      animate      inanimate     1.3862944
569               PP sell      animate      inanimate     1.6094379
567               PP send    inanimate      inanimate     1.3862944
```