Chapter 2
Introduction to R

Abstract This chapter provides a brief introduction to the R environment. The material covers all topics that are necessary to understand the remaining chapters. In addition to basic arithmetic and logical operations, functions and values, data structures and functions that are fundamental to performing operations on sets and set memberships are introduced. Short sections on how to install R, text editors, and finding help complete the chapter.

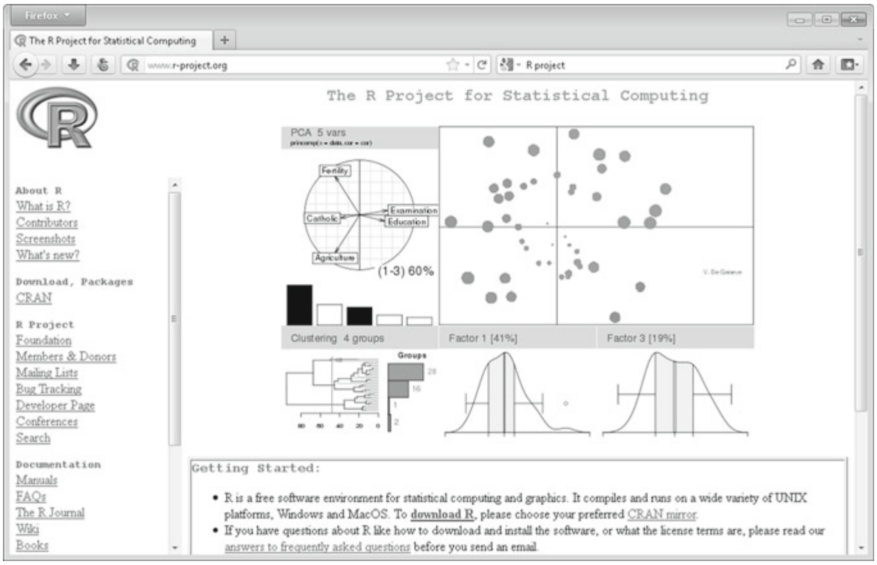
2.1 Installation and Usage

If R is not yet installed, it should first be downloaded from the central R website at `http://www.r-project.org`. This website, shown in Fig. 2.1, provides information about R, its history, documentation, and other resources. In order to download the latest version of R, click on the *CRAN mirror* link in the *Getting Started* box.[1] Select a link under the country nearest to you and choose the appropriate version for your operating system. After the file has been downloaded, install the software with its default settings and open it.

First time users of R may be disappointed by what shows up: an unspectacular window similar to Fig. 2.2. However, R hides its light under the bushel, something you will discover over the course of this book. Most important at this stage is the R console inside the RGui window. The console provides information about the installed version and a few additional things. Unlike other statistical software such as SPSS, R possesses no integrated graphical user interface for communicating your commands to the software by means of radio buttons and mouse clicks.[2] Instead, the commands which tell the software what to do exactly have to be formulated using the

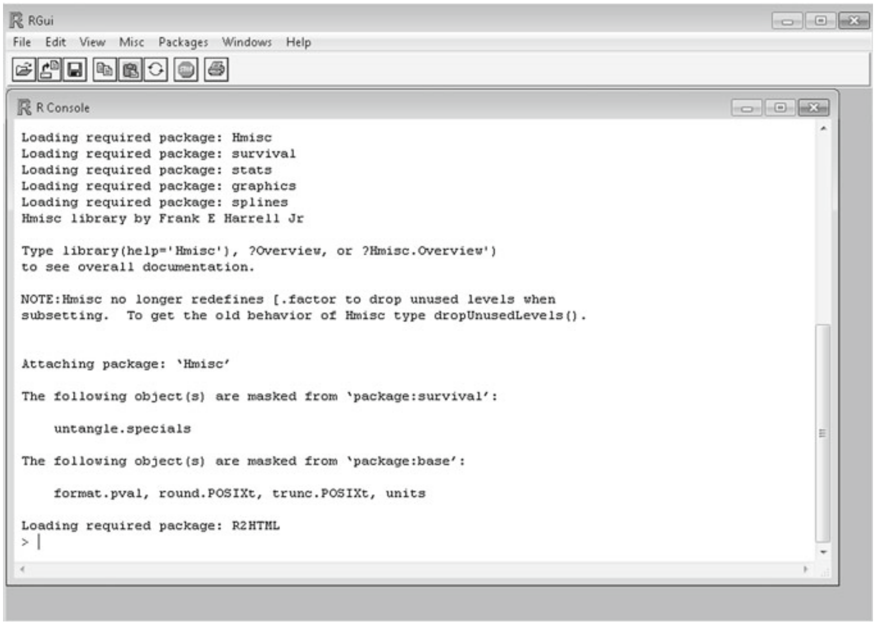
[1] CRAN stands for Comprehensive R Archive Network, describing all servers around the world on which code and documentation for R is stored.

[2] Some packages, such as the Rcmdr, add a graphical user interface.



- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To download R, please choose your preferred CRAN mirror.
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

Fig. 2.1 The R website



```
Loading required package: Hmisc
Loading required package: survival
Loading required package: stats
Loading required package: graphics
Loading required package: splines
Hmisc library by Frank E Harrell Jr

Type library(help='Hmisc'), ?Overview, or ?Hmisc.Overview')
to see overall documentation.

NOTE:Hmisc no longer redefines [.factor to drop unused levels when
subsetting.  To get the old behavior of Hmisc type dropUnusedLevels().


Attaching package: 'Hmisc'

The following object(s) are masked from 'package:survival':

    untangle.specials

The following object(s) are masked from 'package:base':

    format.pval, round.POSIXt, trunc.POSIXt, units

Loading required package: R2HTML
> |
```

Fig. 2.2 Starting a session in R

language of R. It expects these commands in the console where the blinking cursor appears next to the "is-greater-than" sign >.

2.2 Installing and Loading Packages

A package is a directory system containing R code, documentation, and sometimes also data. Packages extend the software by making functions which are not part of the basic distribution available to the user. Some essential packages are already included by default, but the vast majority of packages has to be installed separately.[3] For example, the QCA package contains functions for performing QCA, but it also contains datasets and help files which document everything. Packages which do not come as part of the basic R distribution can be installed through the RGui menu (*Packages→Install package(s)...*) or the `install.packages()` function. Type the following code into the console and press the [Enter] key. This will install the QCA package.

```
> install.packages("QCA", dependencies = TRUE)
```

The `dependencies` argument is set to `TRUE`, which causes all other packages on which the functionality of the QCA package depends to also be installed. The QCA package incorporates functionality provided in the lpSolve package, so lpSolve will be installed alongside QCA. The mere installation of a package, however, is not enough to make its functionality available. Packages also have to be loaded into R at the beginning of a session, either through the RGui menu (*Packages→Load package...*) or the `library()` command.

```
> library("QCA")
```

Installed packages should be regularly updated either through the menu (*Packages→Update packages...*) or the `update.packages()` function.

2.3 Basic Operations, Functions and Values

In essence, R can be conceived of as a programmable calculator whose core functionality can be extended almost limitlessly. Start by entering the following code in the console.

```
> 3 * 5
```

```
[1] 15
```

It does not matter whether spaces are left between the single elements. The two expressions `3*5` and `3 *5` yield the same result, but once expressions become more complex, leaving sensible, and avoiding unnecessary, blank spaces

[3] At the time of writing, about 3,900 packages have been available on CRAN.

in a structured manner is highly advisable. The [1] in front of the result is called the *index*. We will come back to what indexes are useful for at a later stage. This minimal example already illustrates the basic work flow in R: first formulate a command and then send it to the interpreter for processing.

R is an object-oriented language, which means that everything is treated as an object, including functions. A function f has the following structure in R:

```
> f(argument1 = value1, argument2 = value2, ...)
```

For example, the sine function is implemented in R as sin().[4]

```
> sin(pi/2)

[1] 1
```

There are many other such basic functions. For example, instead of using the * operator for calculating the product of 3 and 5, the prod() function could have been invoked.

```
> prod(3, 5)

[1] 15
```

Besides standard names for specific functions, R also has a standard notation for certain values, such as pi for π, NA for missing data, and NULL for empty sets. If an operation is performed which is not defined, R returns NaN (not a number). Note that the division of zero by zero returns NaN, whereas the division of a nonzero number by zero returns Inf, which designates infinity.

```
> 0/0

[1] NaN

> 1/0

[1] Inf
```

In addition to these arithmetic objects, R also offers logical operators, functions, and values. The two logical values are TRUE (true) and FALSE (false). Logical functions include == (is equal) and != (is not equal).

```
> 6 == 7

[1] FALSE

> 6 != 7

[1] TRUE
```

[4] The sine of an angle is the ratio of the length of the side opposite of this angle in a right-angled triangle to the length of the longest side.

Table 2.1 Basic operators, functions, and values

Operator / Function / Value		Description
Arithmetic	`+`, `-`	Addition, subtraction
	`*`, `/`	Multiplication, division
	`^`	Power
Logical	`&`, `&&`	And, and (not vector-valued)
	`|`, `||`	Or, or (not vector-valued)
	`xor`	Either . . . or
Arithmetic	`sum()`, `prod()`	Sum, product
	`min()`, `max()`	Minimum, maximum
	`round()`, `floor()`, `ceiling()`	Round (down / up to integer)
	`sqrt()`	Square root
	`abs()`	Modulus
	`log()`	Natural logarithm
	`exp()`	Exponential function
Logical	`==`, `!=`	Equals, equals not
	`>`, `>=`	Larger, larger or equal
	`<`, `<=`	Smaller, smaller or equal
	`!`	Not (negation)
Arithmetic	`pi`	π
	`Inf`, `-Inf`	Positive and negative infinity
	`NA`	Missing value
	`NULL`	Empty set
	`NaN`	Not a number
Logical	`TRUE`, `FALSE`	True, false

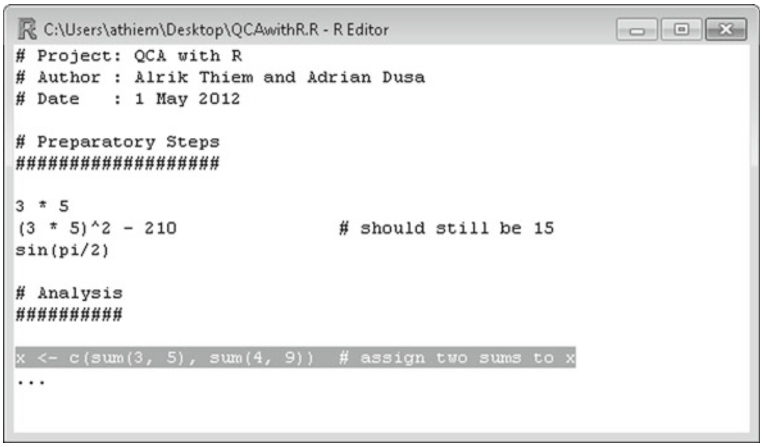
Two important logical operators are `&` (and), and `|` (or).

```
> FALSE & TRUE

[1] FALSE

> 2 == 2 | 2 == 3

[1] TRUE
```

A summary of basic operators, functions, and values available in R, both arithmetic and logical, is given in Table 2.1. We leave it to the reader to experiment with those that have not been introduced in this section.

2.4 Using an Editor

It is inefficient to type commands into the console. Once code grows to more than simple one-line calculations, a good editor becomes an indispensable tool. Editors also aid in identifying programming errors, they ease commenting and sometimes



```
# Project: QCA with R
# Author : Alrik Thiem and Adrian Dusa
# Date   : 1 May 2012

# Preparatory Steps
###################

3 * 5
(3 * 5)^2 - 210              # should still be 15
sin(pi/2)

# Analysis
##########

x <- c(sum(3, 5), sum(4, 9))  # assign two sums to x
...
```

Fig. 2.3 The R script editor

even provide built-in function templates. On the MacOS and Windows operating systems, R comes with a basic editor. It can be started in the RGui window via the menu entry *File→New script*. This rudimentary editor suffices for everything that is presented in this book, but if you intend to carry on with R and the QCA package, we strongly recommend the use of a more sophisticated editor. A good overview of what is available can be found at http://www.sciviews.org/_rgui/projects/Editors.html.

In order to send commands from the R editor to the console, type the command without preceding it by the "is-greater-than" sign >, mark it (with or without comments) and press [Ctrl + R] as shown in Fig. 2.3. The marked code will be send to the console and executed. At the end of a session, scripts can be saved as *.R* files through the editor menu *File→Save as...*.

Besides a good editor, larger projects should also make use of comments, which provide information relating to the code. Comments not only make it easier to keep track of code or re-use it in other projects, but they also allow peers to better understand and replicate results. As replication is a cornerstone of scientific research, the practice of commenting is not to be underestimated. In R, comments can be inserted with the "hash" sign #, after which everything else on the same line will be ignored.

```
> (3 * 5)^2 - 210     # should still be 15
[1] 15
```

2.5 Objects and Assignments

It was mentioned above that R is an object-oriented language. Object-orientation not only means that functions are treated as objects, but also that results from operations using these functions can themselves be saved as objects again. This process is referred to as *assignment*. The most common form of assignment uses the *assignment arrow*, which consists of the "is smaller than" sign < and the "minus" sign - joined together, without spaces in between.

```
> x <- c(sum(3, 5), sum(4, 9)) # assigning two sums to x
```

The two built-in functions c() and sum() have been used here to arrive at a result that is then assigned to the new object x.[5] The c() function concatenates its arguments, the two sums of $3 + 5 = 8$ and $4 + 9 = 13$.[6] In order to avoid mistakes, a new object should thus not be named with the lower case letter c. As R is case-sensitive, upper case C would have also been fine. More generally, however, a name of an object should be an optimal trade-off between a description of its content and efficient programming. For example, one way of naming is to combine lower and upper case letters, as in dataAuthor for a dataset. This way of naming is referred to as *camel notation*. Another system uses a dot as the separator, as in data.author. When x is now entered, the result of the expression assigned to it is called up and printed into the console.

```
> x

[1]  8 13
```

As expected, x consists of two elements—the two sums of 8 and 13—which are preceded by the index [1]. R only shows the index of the first element in that line of the console. To see what happens beyond the first line, enter the following code, including the enclosing parentheses. If necessary, resize the console window.

```
> (x <- rep(5, times = 40))

 [1] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
[26] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
```

A new object x has been created by overwriting the original one that consisted of two sums. This new object results from the rep() function, which repeats its first argument as often as specified in its second argument times. Now the indexing extends over several lines, with only the index of the first element in that line being displayed. Putting parentheses around an entire expression is a useful shortcut to calling up the object by retyping its name.

[5] The sum() function was listed in Table 2.1 above. It adds together all its arguments.

[6] Typing 3 + 5 is an alternative to sum(3, 5).

2.6 Data Structures

A *data structure* is a particular form of arrangement which usually represents the nature of the elements it consists of. A number of different data structures exist in R, but this section focuses on those which occur most often in social science research—*vectors*, *matrices*, *data frames*, and *lists*.

2.6.1 Vectors

Vectors generally represent single rows or columns of data. A single number is also a vector, but a special type thereof referred to as a *scalar*. You have already created two vectors, one of length two and the other of length fifty, using the `c()` and the `rep()` functions in Sect. 2.5. The length of a vector is the number of elements it comprises. It can be found with the `length()` function.

```
> x <- rep(5, times = 50)
> length(x)
```

```
[1] 50
```

Vectors need not necessarily consist of numbers. For example, the three words "Qualitative", "Comparative", and "Analysis" can be concatenated to form a single vector of length three.

```
> (y <- c("Qualitative", "Comparative", "Analysis"))
```

```
[1] "Qualitative" "Comparative" "Analysis"
```

Besides the `c()` and the `rep()` functions, the `seq()` function is also often used to generate vectors which have some sequential structure.

```
> seq(from = 0, to = 10, by = 2)
```

```
[1]  0  2  4  6  8 10
```

The first two arguments `from` and `to` indicate the sequence starting and end points, while `by` specifies the increment. If the starting point and the increment are to equal one, then a shorter route can be taken.

```
> seq(10)
```

```
 [1]  1  2  3  4  5  6  7  8  9  10
```

Sequences of increment one that consist only of integers can be easily produced using a colon.

```
> 1:5
```

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

2.6.2 Matrices

A matrix is a rectangular arrangement of elements on which mathematical operations can be performed. The `matrix()` function offers the easiest way to create matrices in R.

```
> x <- seq(20)
> matrix(x, nrow = 5)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

In addition to the first argument `x` representing the data, the function can take two further arguments: `nrow` for the number of rows and `ncol` for the number of columns. It suffices to provide either the former or the latter. Matrices are filled column-wise by default. For row-wise filling the argument `byrow = TRUE` should be used.

Matrices can also be constructed from existing row or column vectors. The `cbind()` function binds columns together, whereas the `rbind()` function binds rows together.

```
> x <- rep(seq(4), 2)
> y <- rep(c(7, 8), 4)
> rbind(x, y)
```

```
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
x    1    2    3    4    1    2    3    4
y    7    8    7    8    7    8    7    8
```

Notice that the name for the second argument in the `rep()` function—`times`—has been omitted. The `rep()` function "knows" that the second argument is always the number of times the first argument should be repeated. The omission of argument names works for all R functions, but for didactic reasons, we will always write them out.

A peculiar feature of R can be nicely demonstrated with matrices—vector recycling. It allows to perform operations and create objects which are usually mathematically impossible. Enter the following block of code, the two last input lines of which seem meaningless.

```
> x <- seq(4)
> y <- rep(c(7, 8), 4)
> rbind(x, y)
```

```
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
x    1    2    3    4    1    2    3    4
y    7    8    7    8    7    8    7    8
```

```
> x + y
```

```
[1]  8 10 10 12  8 10 10 12
```

It remains true that two vectors of different lengths cannot be bound together to create a matrix. Vector `x` is of length four, `y` of length eight. Also, vector addition is undefined for summands of unequal length. In these and similar cases, however, R automatically recycles the shorter vector as long as the length of the longer vector is a multiple of the length of the shorter vector.

2.6.3 Data Frames

Data frames are the most common data structure in the social sciences. They are very similar to matrices, but unlike matrices, which can only contain data of one *data type*, data frames can accommodate different types. Let us first create a small data frame from the information about the three QCA variants that was presented in the introduction in Fig. 1.1 using the `data.frame()` function.

```
> QCAdat <- data.frame(variant = c("csQCA", "mvQCA", "fsQCA"),
+  number = c(170, 7, 62))
> QCAdat
```

```
  variant number
1   csQCA    170
2   mvQCA      7
3   fsQCA     62
```

The data frame consists of three rows and two columns. The particularity of `QCAdat` is that the first column contains elements which consist only of letters, whereas the second column's elements are numbers. The two columns therefore contain data of different data types.

2.6.4 Lists

Data frames are special cases of *lists*, another useful data structure in R. Lists are extremely flexible because they can store all of the above structures in a single object. The creation of lists is achieved with the `list()` function.

Table 2.2 Basic data types

Type	Description		Example
Logical	Logical values		`TRUE`
Numeric	Real numbers	*Integer*	`3`
		Double	`2.71`
Character	Letters and strings		`"QCA"`

```
> (QCAlist <- list(dat = QCAdat, txt = c("Happy", "QCAing")))

$dat
  variant number
1   csQCA    170
2   mvQCA      7
3   fsQCA     62

$txt
[1] "Happy"  "QCAing"
```

Rarely are lists created directly by end-users. Instead, they are usually generated when a complex function, such as QCA's `truthTable()`, returns a result that is not just a single number, but a collection of several different objects.

2.7 Data Types

At the most basic level, objects can be divided into different atomic *data types*. Three such types are listed in Table 2.2 in increasing order of hierarchy. The lowest priority is given to *logical*, the highest to *character* values. *Numeric* values fall in between. The reason for this hierarchy builds on set relations. It is possible to represent all real numbers with character strings, but not the other way around. It will later be shown that real numbers can in turn represent the logical values `TRUE` and `FALSE`, but logical values cannot represent all real numbers. In set-theoretic language, the set of logical values is a subset of the set of real numbers, which is itself a subset of the set of character values. The real numbers are further divided into *integer* and *double*, the former of which is again a subset of the latter.[7]

Below we define two vectors, the first consisting of three letters, the second of a sequence of six numbers.[8]

[7] More precisely, *numeric* is identical to *double*.

[8] The two objects `letters` and `LETTERS` are predefined constants in R.

```
> (Letters <- c("Q", "C", "A"))
[1] "Q" "C" "A"
> (Numbers <- seq(from = 1, to = 2, by = 0.2))
[1] 1.0 1.2 1.4 1.6 1.8 2.0
```

The data type of `Letters` and `Numbers` can be queried with the `mode()` function.

```
> mode(Letters)
[1] "character"
> mode(Numbers)
[1] "numeric"
```

The data type of `Letters` is *character*, that of `Numbers` is *numeric*. The two are now to be concatenated with the `c()` function to create the new object `LetNum`. The data type of `LetNum` will be *character*, not *numeric*, because that data type which is highest in the hierarchy will always be chosen so as to avoid a loss of information.

```
> (LetNum <- c(Letters, Numbers))
[1] "Q"   "C"   "A"   "1"   "1.2" "1.4" "1.6" "1.8" "2"
> mode(LetNum)
[1] "character"
```

Whether or not an object is of a specific data type can be tested with the function class `is.<data type>()`.

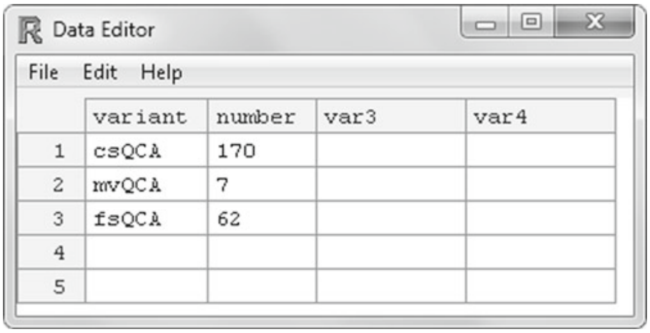
```
> is.character(Letters)
[1] TRUE
```

While `is.<data type>()` only tests for the data type, the function class `as.<data type>()` can be used in order to coerce objects to specific data types.

```
> (Numbers <- as.integer(Numbers))
[1] 1 1 1 1 1 2
```

The vector `Numbers` remains *numeric*, but it is now not *double* anymore.[9]

[9] Note that `as.integer()` works similarly to the `floor()` function presented in Table 2.1.



	variant	number	var3	var4
1	csQCA	170		
2	mvQCA	7		
3	fsQCA	62		
4				
5				

Fig. 2.4 R data editor

2.8 Accessing Data

Working with and operating on datasets in R is different from working with general spreadsheet software. We introduce several ways of accessing data in this section, each of which may be more useful than the other in certain situations. As a preparatory step, let us recreate the data frame `QCAdat` from Sect. 2.6.3, which gave the number of times each QCA variant has been applied.

```
> QCAdat <- data.frame(variant = c("csQCA", "mvQCA", "fsQCA"),
+  number = c(170, 7, 62))
> QCAdat

  variant number
1   csQCA    170
2   mvQCA      7
3   fsQCA     62
```

The data frame can be seen in the R console, but trying to change elements in `QCAdat` from within the console will not work. There exist several ways whereby `QCAdat` can be accessed. For example, the `edit()` function can be used for small changes, such as the replacement of single values or a correction of a variable label. It will call up the built-in R data editor shown in Fig. 2.4.

```
> QCAdat <- edit(QCAdat)
```

The data frame is conveniently small, but problems arise if, for example, a new variable with the recoded values of an existing variable in a larger dataset should be added. For this and similar purposes, the "dollar" sign `$` is useful. Suppose a nominally-scaled variable indicating in which research area the occurrence of each QCA variant has been highest should be generated. Applications of csQCA have appeared most often in sociology, those of mvQCA and fsQCA in political science.

```
> QCAdat$occur <- c("sociology", rep("politics", 2))
> QCAdat

  variant number     occur
1   csQCA    170 sociology
2   mvQCA      7  politics
3   fsQCA     62  politics
```

The variable *occur* is created within `QCAdat` by putting a `$` in front of it. As both mvQCA and fsQCA applications have occurred most often in political science, the `rep()` function avoids typing in the character value `"politics"` twice. The `$` sign can also be applied to perform operations on existing variables. For computing the total number of QCA applications, the `sum()` function can be run over all values in the respective column.

```
> sum(QCAdat$number)

[1] 239
```

Sometimes access to only a subset of the data may be needed. Possibly more than a few values of a variable but fewer than all of them. In such cases, the `subset()` function is useful.

```
> subset(QCAdat, subset = number > 50)

  variant number     occur
1   csQCA    170 sociology
3   fsQCA     62  politics
```

Its argument `subset` requires a logical expression for the selection of elements or rows. A second argument of `subset()` is `select`, which specifies the desired columns.

```
> subset(QCAdat, select = variant)

  variant
1   csQCA
2   mvQCA
3   fsQCA
```

These two arguments can also be applied together in order to extract any combination of elements or rows and columns from the data.

```
> subset(QCAdat, subset = number > 50, select = variant)

  variant
1   csQCA
3   fsQCA
```

Another way of accessing data is through *indexing*, also referred to as *subscripting*. The structure of an index is `<object>[<row(s)>, <column(s)>]`. For example, in order to select the entry for the number of mvQCA applications, the second row has to be specified, because mvQCA applications are listed there, and the second column, because this is the variable with the number of applications.

```
> QCAdat[2, 2]
```

```
[1] 7
```

Now suppose all values from the number of applications should be extracted. In this case, a substitute for `QCAdat$number` is to leave away the row index and only provide the column index.

```
> QCAdat[ , 2]
```

```
[1] 170   7  62
```

It is also possible to use logical expressions, as in the `subset()` function, with indexes.

```
> QCAdat[QCAdat$number > 50, 2]
```

```
[1] 170  62
```

In the case of lists, a special way of indexing must be used. For example, in order to access the second element of the list object `QCAlist` created in Sect. 2.6.4, the value 2 should be enclosed by double square brackets.

```
> QCAlist[[2]]
```

```
[1] "Happy"  "QCAing"
```

After this more general introduction, the next two sections will now introduce useful functions for operating on the elements of sets and their set memberships.

2.9 Operations on Sets

This section introduces the basic operations available in R with regard to the elements of sets, not their membership, which is assumed to be crisp. Let us first create two equally-sized sets, **X** and **Y**, whose six elements are random samples of integers between one and ten.

```
> set.seed(1)
> (X <- sample(1:10, size = 6))
```

```
[1] 3 4 5 7 2 8
```

```
> set.seed(10)
> (Y <- sample(1:10, size = 6))
```

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

The `set.seed()` function is very useful whenever random sampling is applied because it allows the retrieval of the exact same sample. Its single argument is just a starting number for the random number generator. The `sample()` function takes a sample of the specified size from the elements of its first argument. The union of these two sets containing all *unique* elements which belong either to **X** or **Y** can be found with the `union()` function.

```
> union(X, Y)
```

```
[1] 3 4 5 7 2 8 6 1
```

In contrast, the intersection containing all unique elements which belong to **X** and **Y** can be found with the `intersect()` function.

```
> intersect(X, Y)
```

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

Another useful function is `setdiff()`, which returns all unique elements of the first set which are not unique elements of the second set.

```
> setdiff(X, Y)
```

```
[1] 7 8
```

The last function to be introduced is `setequal()`, which returns a logical statement about the equality between all *unique* elements of two sets.

```
> setequal(X, Y)
```

```
[1] FALSE
```

It is important to emphasize the word *unique* here, otherwise the result of the following example would be surprising.

```
> A <- c(1, 2, 3, 4)
> B <- c(1, 1, 2, 3, 3, 3, 4)
> setequal(A, B)
```

```
[1] TRUE
```

In this case, the test result is true because all unique elements in **A** and **B** are equal. However, if *exact* equality is to be tested, the `identical()` function should be used.

```
> B <- A
> identical(A, B)
```

```
[1] TRUE
```

2.10 Operations on Set Memberships

After basic operations on sets have been introduced, this section now demonstrates how to perform calculations on elements' set memberships. The two functions `pmin()` and `pmax()` are central for this purpose. Let us begin with a small dataset of fuzzy-set membership scores, named `datFS`. All functions and structures work exactly the same for crisp sets.

```
> set.seed(1)
> datFS <- data.frame(A = runif(5), B = runif(5), C = runif(5))
> (datFS <- round(datFS, 2))
```

```
     A    B    C
1 0.27 0.90 0.21
2 0.37 0.94 0.18
3 0.57 0.66 0.69
4 0.91 0.63 0.38
5 0.20 0.06 0.77
```

The dataset `datFS` consists of five cases with set membership scores in three conditions **A**, **B**, and **C**. The `runif()` function generates a random deviate from a uniform distribution for each case and condition. Before calling up `datFS`, the `round()` function introduced in Table 2.1 cuts these deviates back to two decimal places. Having created the dataset of set membership scores, the `pmin()` function can now be applied in order to calculate the result of the expression $\mathbf{a} \cdot \mathbf{B} \cdot \mathbf{C}$ (not **A** AND **B** AND **C**). The `pmin()` function returns parallel minima, which means that instead of simply taking the single smallest value from all values in the sets passed to `pmin()` as arguments, set membership scores are compared separately for each case (row). The computation of complements can be achieved by making use of the fact that for every set complement **set**, $\mathbf{set} = 1 - \mathbf{SET}$.

```
> (datFS$aBC <- pmin(1 - datFS$A, datFS$B, datFS$C))
```

```
[1] 0.21 0.18 0.43 0.09 0.06
```

Recall that R recycles the value(s) of the shorter vector when performing operations on two vectors of different lengths. This is why the scalar `1` need not be repeated five times in order to compute the complement of **A**. The function `pmax()` is applied analogously for parallel maxima. For example, the result of the expression $\mathbf{A}+\mathbf{b}+\mathbf{C}$ (**A** OR not **B** OR **C**) can be calculated as follows:

```
> (datFS$"A+b+C" <- pmax(datFS$A, 1 - datFS$B, datFS$C))
```

```
[1] 0.27 0.37 0.69 0.91 0.94
```

The set name for the new disjunctive combination has to be enclosed in double quotes because the Boolean "or" sign `+` would otherwise be treated as an arithmetic operator rather than part of a string. The two functions `pmin()` and `pmax()` can also be combined in nested structures so that even complex expressions require relatively little programming effort. As an example, let us calculate $\mathbf{A} \cdot \mathbf{b} + \mathbf{B} \cdot \mathbf{c}$.

```
> datFS$"Ab+Bc" <- pmax(
+  pmin(datFS$A, 1 - datFS$B),
+  pmin(datFS$B, 1 - datFS$C)
+ )
> datFS
```

```
     A    B    C  aBC A+b+C Ab+Bc
1 0.27 0.90 0.21 0.21  0.27  0.79
2 0.37 0.94 0.18 0.18  0.37  0.82
3 0.57 0.66 0.69 0.43  0.69  0.34
4 0.91 0.63 0.38 0.09  0.91  0.62
5 0.20 0.06 0.77 0.06  0.94  0.20
```

Even complete truth tables can be constructed with `pmin()` and `pmax()`. Instead of using numeric truth values of `1` and `0`, here we choose `TRUE` and `FALSE` in order to demonstrate why, as stated in Sect. 2.7, these logical values are subsets of real numbers in R. Using QCA's `createMatrix()` function, enter the following code to construct the first part of the truth table `tt`, containing all 2^k configurations from the $k = 3$ crisp sets **A**, **B**, and **C**.[10]

```
> tt <- data.frame(createMatrix(rep(2, 3), logical = TRUE))
> names(tt) <- c("A", "B", "C")
```

After having generated all $2^3 = 8$ configurations, the condition labels for sets **A**, **B**, and **C** are assigned to each column of `tt` using the `names()` function. The outcome value (truth value) of the expression $\mathbf{a} \cdot \mathbf{B} \cdot \mathbf{C}$ can then be computed as follows:

```
> tt$OUT <- pmin(1 - tt$A, tt$B, tt$C)
> tt
```

```
      A     B     C OUT
1 FALSE FALSE FALSE   0
2 FALSE FALSE  TRUE   0
3 FALSE  TRUE FALSE   0
4 FALSE  TRUE  TRUE   1
5  TRUE FALSE FALSE   0
6  TRUE FALSE  TRUE   0
7  TRUE  TRUE FALSE   0
8  TRUE  TRUE  TRUE   0
```

[10] The `createMatrix()` function is primarily used internally for constructing truth tables.

In R, the logical value TRUE is equivalent to the numerical value 1, whereas FALSE corresponds to the numerical value 0. The set of logical values is therefore a subset of the set of real numbers and R can calculate the result of 1 - FALSE. In consequence, the only true statement in the truth table tt for the expression $\mathbf{a} \cdot \mathbf{B} \cdot \mathbf{C}$ is the combination FALSE-TRUE-TRUE in row four.

2.11 Importing and Exporting Data

Social-science datasets are rarely built directly in R, but usually in spreadsheet or database software such as Excel or Access. It is thus important to know how to import external data files. R can read many different file formats and from different sources, but the easiest way is to first prepare the data in a spreadsheet software and save it as a tab-delimited text file.[11] This file type ensures a small file size and it can be easily imported into all kinds of software on all kinds of operating systems. Make sure that no cell entry, including values, row, and column labels, has a blank space. Blank spaces should generally be avoided, irrespective of whether the data is to be further analyzed in another software or not. Instead of naming a variable *GDP Growth*, *GDPgrowth* should be used. In addition, all entries or empty cells denoting missing values should already be marked with NA.[12]

In the bibliography section at http://www.compasss.org a number of datasets from published studies can be found. Download one of them and save it in the working directory of your R installation. The working directory is the folder from which R has been started. Its path can be found by entering the command getwd().

```
> getwd()
```

The working directory can be changed by providing a file path to the setwd() function.

```
> setwd("C:/Myfolder")
```

File path specifications use the following structure in R: "C:/.../..." or "C:\\...\\...". The familiar Microsoft Windows backslash structure "C:\...\..." cannot be used because the single backslash is a special character. Once the working directory has been found or changed to the preferred folder, and the dataset saved in it, the read.table() function will load it into the R workspace. The workspace is the working environment and includes all objects (vectors, matrices, data frames, functions, etc.) which have been created in the current session. For demonstration purposes, we use the dataset by Arvind and Stirton (2010) on the reception of the Code Napoleon in Germany.

[11] Text files have the file type extension *.txt*.

[12] R can handle other indicators for missing values, but an optimal preparation of the data according to R's standards facilitates their import.

```
> AS <- read.table("ArvindStirton2010set.txt", header = TRUE,
+  row.names = "State")
> AS
```

```
         D    C   F    I    L    N    A    O
Rhine 1.00 1.00 0.4 0.00 0.50 0.00 0.50 1.00
KiWes 1.00 0.75 0.6 0.00 0.60 0.00 0.90 1.00
GDBer 1.00 0.75 0.4 0.00 0.75 0.00 1.00 1.00
..... .... .... ... .... .... .... .... ....
<<rest omitted>>
```

The `read.table()` function makes `AS` a data frame by default. The original file in which the data is stored is given as the first argument. By passing `TRUE` to the optional `header` argument, the first row of `ArvindStirton2010set.txt` is identified as containing the variable labels for `AS`. The name of the variable which contains the case identifiers is passed to the `row.names` argument. If in your own data the decimal separator is a comma instead of a point, `dec = ","` should be added as an argument. If you neither want to save the data in your working directory nor change the directory, the data can also be put into any folder and the entire file path to R be provided instead.

Besides the `read.table()` function, there also exists a `read.csv()` function for reading comma separated values. In addition, the foreign package provides functionality for importing data stored in other formats such as *.sav* (SPSS) or *.dta* (Stata). We refer you to introductory textbooks on R and R's own manuals for further information on data import.

For saving data from R as *.txt* files, the `write.table()` function can be used.

```
> write.table(myfile, file = "myfile.txt", sep = "\t",
+  quote = FALSE)
```

The specification `\t` in the `sep` argument creates tab-separated values, and the logical argument `quote = FALSE` avoids double quotes being put around character values. This way, `myfile` will be structured in exactly the same way as the file `ArvindStirton2010set.txt`.

2.12 Finding Help

Sometimes software lets you cry for help. If new to R, this will happen all the more so. The learning curve for R is steep at the beginning, but once you have become familiar with the fundamentals, the "products" it delivers will often have been cheap at twice the price. However, even experienced users regularly seek help because every new project will at least be a little different from the previous one and possibly require a slightly different solution.

The first point of reference is R's internal help facilities, which can be accessed with the question mark `?`, followed by the term which is sought. This will open the respective documentation files, in HTML or plain text format. For example, to look up the documentation for the `prod()` function, type the following:

```
> ?prod
```

Help pages have a standardized structure with all necessary information. Let us go through the most important headings, including *Description*, *Usage*, *Arguments*, *Details*, *Value*, and *Examples* because you will certainly come across them again. The *Description* states what the object you asked for is about. Not surprisingly, the `prod()` function returns the product. The *Usage* section shows the complete structure of `prod()`. Each element in this structure is explained under *Arguments*. The three dots `...` denote numeric or complex or logical vectors. Without going into the details here (the meaning of *numeric* and *logic* vectors has been explained in Sect. 2.7), this means that we could have also told R to calculate the product of 3 and 5, as we did above in Sect. 2.3 with the `*` operator, by passing to the `prod()` function the two numeric values 3 and 5.

```
> prod(3, 5)

[1] 15
```

The second element in the *Arguments* list is `na.rm`. This argument is logical, which implies that only one of two values can be passed to it: `TRUE` or `FALSE`. Under the *Usage* section, you saw that the default value of this argument was set to `FALSE`. It means that missing values in the vectors provided will not be removed unless specified otherwise by the user. This is very important! Compare the two following examples.

```
> x <- c(3, 5, NA)
> prod(x)

[1] NA

> prod(x, na.rm = TRUE)

[1] 15
```

The second version removes the missing value from `x` before taking the product. One of the main reasons for getting `NA` as the result of an operation where you did not expect it is that the `na.rm` argument has not been set to `TRUE`.[13] You are also informed about this feature in the *Details* section, which generally provides more in-depth information about an object and its particularities. The section *Value* exactly describes what is returned from the function, namely a vector of length one—the product. The *Examples* section provides minimal, and sometimes not so minimal, working examples. Just copy the example from the help page into the editor.

[13] This often happens with the `mean()` function, for example.

```
> print(prod(1:7)) == print(gamma(8))
```

```
[1] 5040
[1] 5040
[1] TRUE
```

The `print()` function is a generic function for printing its arguments into the console. As it is used twice, the first two objects that appear as printed output in the console are the results of each `print()` function's argument. Both expressions yield the same number, so the logical operator == returns the value `TRUE` as the third output that results from the entire expression.

The `help.start()` function automatically opens the HTML version of the general R help page, from which you can proceed further.[14] The advantage of the HTML version over the text version is the availability of links for jumping directly to related topics.

[14] You can also use the menu in the RGui window: *Help→HTML Help*.