

# Chapter 2

## An Introduction to R for Quality Control

**Abstract** This chapter introduces R as statistical software and programming language for quality control. The chapter is organized as a kind of tutorial with lots of examples ready to be run by the reader. Moreover, the code is available at the book’s companion website. Even though the RStudio interface is also introduced in the chapter, any other user interface can be used, including the R default GUI and code editor.

### 2.1 Introduction

In this chapter, the essentials of the R statistical software and programming language [27] are explained. This provides the reader with the basic knowledge to start using R for quality control. You should try the code by yourself while reading this chapter, and therefore you need R and RStudio (optionally but recommended) installed on your computer before continuing reading the chapter. Follow the step-by-step instructions explained in Sect. 1.6 of Chapter 1, or just go to the R website<sup>1</sup> and to the RStudio website,<sup>2</sup> download the installation files, and install them to your computer. If you are reading the electronic version of this chapter, you can copy and paste the code in the examples.<sup>3</sup> The code is also available at the book’s companion website.<sup>4</sup> In any case, we recommend typing everything, at least at the beginning, in order to get used to the R mood.

In Chapter 1, we introduced *the power of R for quality control*, what it is, its history, etc. This chapter goes into the details of the software to get advantage of that power. We highlight here some of the R features explained in Sect. 1.4 of Chapter 1:

---

<sup>1</sup><http://www.r-project.org>.

<sup>2</sup><http://www.rstudio.com>.

<sup>3</sup>Please note that sometimes what you paste could not be exactly what you see in the book and some modifications could be needed.

<sup>4</sup><http://www.qualitycontrolwithr.com>.

- R is the evolution of the S language, developed at Bell Laboratories (then AT&T and Lucent Technologies) in the 1970s [6]. Note that it is the same company where Walter Shewhart developed modern statistical quality control 50 years before [34];
- R is maintained by a foundation, a Core Team, and a huge community of users and stakeholders, including commercial companies that make their own developments;
- R is Free and Open Source Software (FOSS). Free as in free beer, and free as in free speech [14];
- R is also a programming language, and a system for statistical computing and graphics;
- R is platform independent: it runs in Windows, Mac, and Linux;
- The way of interacting with R is by means of expressions, which are evaluated in the R Console, or can be stored in R scripts to be run as programs;
- R has Reproducible Research and Literate Programming capabilities, which has proven quite useful for quality control reports in Sect. 1.6, Chapter 1;
- R base functionality provides a complete set of tools for statistical computing and plotting, developed by time-honored experts;
- R base functionality is expanded by an increasing number of contributed packages for a wide range of applications, including some for quality control;
- The software can be customized creating new functions for particular needs.

The toughest part for new R users is to get used to the interactivity with the system. Having to write the expressions prompts errors which, especially at the beginning, are not easy to interpret. Nevertheless, those errors are usually caused by similar patterns. Find below a list of **common errors** while writing R expressions. If you get an error when running an R expression, it very likely can be classified into one of those categories. Please take into account that those types of errors are not made only by beginners, but it is part of the normal use of R. Practice will reduce the number of times errors are produced and, more importantly, the time one lasts realizing where is the problem and fix the expression. This list contains concepts that you still do not know about. Note the list as a reference and come back here whenever you get an error while reading the chapter and practicing with the code. Once you have completed the chapter, read the list again to fix concepts.

- **Missing closing character.** You need to *close* all the parentheses, square brackets, curly brackets, or quotation marks you had opened in an expression. Otherwise the expression is incomplete, and the console prompt keeps waiting for you to finish it with the + symbol. If you are running a script, R will try to continue the expression with the next one, and the error message could be uninformative. So always check that you do not have a missing closing symbol;
- **String characters without quotation marks.** String characters must be provided in quotation marks ("). Everything in an expression that is not in quotation marks is evaluated in the workspace, and therefore it should exist, either in the global environment or in other environments. Usually, the error message indicates that the object does not exist, or something related to the class of the object;

- **Missing parenthesis in functions.** Every function must include parenthesis, even if it does not need any argument;
- **Missing arguments in a function call.** Functions accept arguments, which sometimes can be omitted either because they are optional or because they have a default value. Otherwise they are mandatory and a message indicates so;
- **Wrong arguments in functions.** Sometimes it is due to the missing quotation marks mentioned above. Check the class of the object you are using as argument, and the class the function expects as argument;
- **Incompatible lengths.** Data objects have a length that may be critical when using them. For example, the columns of a data frame must be of the same length;
- **Wrong data.** If a vector of data is supposed to be, for example, numeric, but one or more of its components is another thing, for example a character string, then computations over the vector might result on unexpected results. This is not always so evident, as it may be *a number* but the computer might interpret a character, for example due to spurious blank spaces or the like;
- **Other wrong syntax errors.** Check the following:
  - The arguments in a function are separated by commas (,);
  - The conditions in loops and conditions are in parenthesis;
  - You do not have wrong blank spaces, for example in the assignment operator;
  - Use a period (.) as decimal separator in numbers;
  - Expressions are in different lines, or separated by a semicolon.

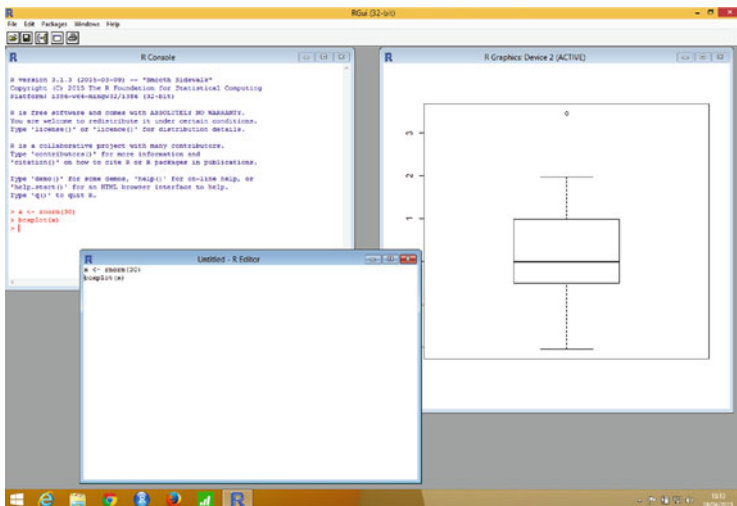
In the remaining of the chapter you will find an overview of R interfaces in Sect. 2.2; a description of the main R elements in Sect. 2.4; an introduction to RStudio in Sect. 2.5; Sects. 2.6 and 2.7 describe how to work with data within R and with external data sources. This is the starting point for the application of the quality control tools explained throughout the book; a QualityControl task view is proposed in Sect. 2.8; finally, some ideas and thoughts about R and Standardization are given in Sect. 2.9. Note that the specific functions and packages for quality control are not included in this chapter, as they are explained in detail in the corresponding chapter. For example, functions for modelling processes are in Chapter 5, and so on. Appendix C is a complete cheat sheet for quality control with R.

## 2.2 R Interfaces

The R base installation comes with a Command Line Interface (CLI). This CLI allows interacting with R using the R Console as outlined above, by means of expressions and scripts. This is one of the hardest parts for beginners, especially for those who do not have experience in programming. Luckily, being R open source software and a programming language at the same time allows developing more advanced interfaces to work with R. For the Windows and Mac versions of R, an extremely simple Graphical User Interface (GUI) is also included with the base installation. It can be started as any other application in the system, Figure 2.1 shows the GUI for Windows.

There are a number of projects regarding R interfaces. A list of them can be found in the R website itself following the “Related Projects” link and then R GUIs, or just visit [http://www.sciviews.org/\\_rgui](http://www.sciviews.org/_rgui). We can find two types of R interfaces:

- **Interfaces with menus and dialog boxes (MDB GUIs).** Interfaces of this type provide the user with menus and dialog boxes to perform statistical analysis in a similar way other commercial software does. However, only a limited number of options are included in those menus. They are based on a common framework, where package developers build functionality for their functions. The most popular frameworks are **R Commander** (package `Rcmdr`) [13] and **Deducer** (package `Deducer`) [11], and they can be loaded inside **R** as any other package;
- **Interfaces for development (Integrated Development Environment, IDE).** Interfaces of this type provide an environment to make the analyst life easier, but they do not provide an interface where one can enter data or click options and then run an analysis. Nevertheless, they allow to exploit all the capabilities of the R system. Most popular environments include **RStudio**<sup>5</sup> [30], **Emacs + ESS**<sup>6</sup> (Emacs Speaks Statistics), and **Eclipse + StatET**.<sup>7</sup>



**Fig. 2.1** R GUI for windows. The R GUI allows basic interaction with **R** through the R console; scripts can be created using the R Editor; and the R Graphics device opens when invoking a plot. The menu bar contains access to some basic operations such as installing packages, or save and load files

<sup>5</sup><http://www.rstudio.com>.

<sup>6</sup><http://www.ess.r-project.org>.

<sup>7</sup><http://www.walware.de/goto/statet>.

The approach followed in this book is using an interface of the second type. This allows to use all the capabilities of R, and the examples provided throughout the book can be used either in the built-in R GUI, both in the R console and as scripts in the R Editor, or in other available GUIs. In what follows, we explain one of the interfaces that has become very popular among a wide range of R users, including those using R in Industry: RStudio. This choice does not mean that one interface is better than the others. In fact, we invite the reader to try out more than one and decide by themselves which one fits better their needs. In fact, we have been using both RStudio and Eclipse + StatET to write this book using Reproducible Research and Literate Programming techniques. The good thing is that we can choose between several alternatives. Moreover, as we remarked above, all the examples in the book are ready to use in any R interface, or interactively in the console.

## 2.3 R Expressions

The way to interact with R is through R expressions, sometimes named as commands. As explained above, R is interactive, in the sense that it responds to given inputs. Such inputs are R expressions, which can be of several types, mainly:

- An arithmetic expression;
- A logical expression;
- A call to a **function**;
- An **assignment**.

Expressions are evaluated by R when run in the console or through a script. If the expression is incomplete, the R Console prompt keeps waiting until the expression is complete. If the expression is complete, R evaluates the expression, producing a result. This result may show some *output* to the user, which can be textual or graphical. Some expressions do not produce any *visible* output, being the result, for example, storing data in variables, or writing data to disk.

One of the characteristics of R is that it works with in-memory data. Nevertheless, we will need to work with expressions containing files in several ways. Some of them are:

- Read data files to use in data analysis;
- Write data files to use later on;
- Save plots to be included in reports using other software tools;
- Create R scripts to write sets of expressions containing a complete analysis;
- Create report files with code, results, data, and text suitable to be compiled and delivered.

In summary, the purpose of using files in R can be either working with data, or working with code. When files are involved in R expressions, we can provide the file location using two approaches:

- Through the absolute path, i.e., the location in the computer from the root file system;
- Through the relative path, i.e., the location in the computer from the working directory of the R session (see below).

File paths must be provided as character strings, and therefore quotation marks must be used. When using Windows, it is important to note that the backward slash character (“\”) is reserved for *escaping*<sup>8</sup> in R, and Windows paths must be provided using either a forward slash (“/”) or a double backward slash (“\\”) to separate folders and file names. For relative paths, the usual symbols for current and parent directories (“.” and “..” respectively) can be used.

## 2.4 R Infrastructure

The R infrastructure is composed of the following elements:

- The console
- The editor
- The graphical output
- The history
- The workspace
- The working directory

In the R GUI, the console, the editor, and the graphical output are the three windows that can be seen in Fig. 2.1. However, the history, the workspace, and the working directory are *hidden* and we need to use coding to access them. As remarked above, interfaces like RStudio allow more options in order to work with those system-related elements. Moreover, advanced functionality is available to: easily access to objects and functions; syntax highlighting; contextual menus; access to help; explore data; etc. Nevertheless, the interface is actually a *wrapper* for the R system, and the level of interaction for the statistical analysis is the same: console and scripts.

## 2.5 Introduction to RStudio

RStudio is a Java-based application, and therefore having Java installed is a prerequisite. Make sure you have the latest version of Java<sup>9</sup> to avoid possible issues.

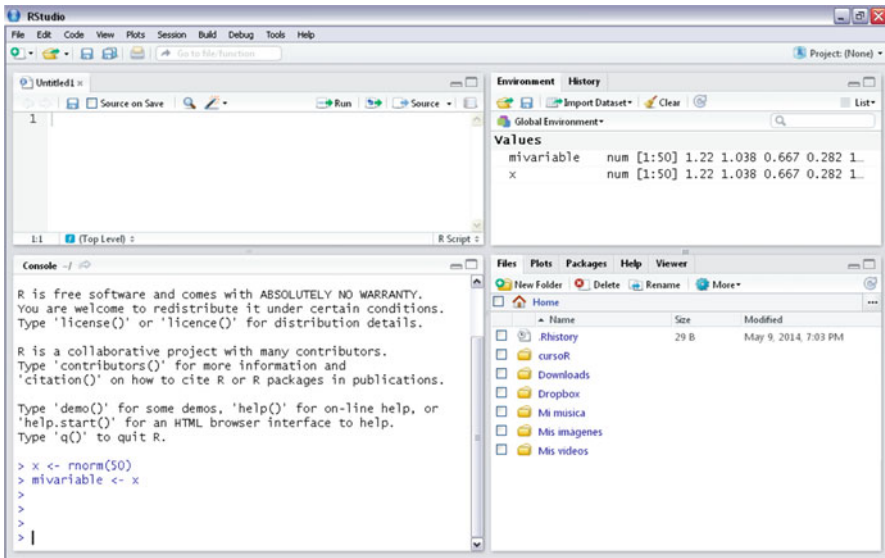
---

<sup>8</sup>Escaping means to provide a character string with special characters. For example, \n is for the special character *new line*.

<sup>9</sup><http://www.java.com>.

The RStudio interface is shown in Fig. 2.2.<sup>10</sup> It has a layout of four panes whose dimensions can be adjusted, and each pane can contain different types of elements by means of tabs. Most of those elements are basic components of the R system listed above. The default layout is as follows<sup>11</sup>:

1. Lower-left pane. This pane is for the **R Console**. It can also show system-related elements such as the output of the system console when calling system commands, for example to compile a report;
2. Upper-left pane. This pane is for the **R Source**. R Scripts are managed in this pane. Other types of files can also be opened in this pane, for example text files containing data, code in other programming languages, or report files. Data sets are also shown in this pane;
3. Upper-right pane. This pane is mainly for the **R History** and the **R Environment**. Other tabs appear when using certain features of RStudio, such as packages development, or R Presentations;
4. Lower-right pane. This pane is the most populated. It has the following tabs:
  - **Files**. It is a system file explorer with basic functions. It can be linked to the R working directory;



**Fig. 2.2** RStudio layout. The RStudio interface is divided into four panes: the console pane, the source pane, the workspace and history pane, and the files, plots, packages, and help pane. The layout can be modified through the global options in the Tools menu

<sup>10</sup>The version used while writing this book was 0.99.xxx.

<sup>11</sup>It can be changed through the *Tools > Global options* menu.

- **Plots.** It is the RStudio graphics device. The plots generated in R are shown here;
- **Packages.** Shows the packages available in the system, and we can install, uninstall, or update them easily;
- **Help.** This tab provides access to all the R Documentation, including the documentation of the installed contributed packages;
- **Viewer.** This tab is used to develop web applications with RStudio, which will not be covered in this book.

## The R(Studio) Console

The R console in RStudio is located by default in the lower-left pane, see Fig. 2.3. Its behavior is the same as in the standard R GUI: there is a prompt identified by the “>” symbol that is waiting for an expression. The user writes an expression after the prompt and presses the `Intro` or `return` key. R evaluates the expression and produces a result. An important issue puzzling for newcomers that arises quite often is that if an expression is incomplete, the prompt changes to the “+” symbol, waiting for the rest of the expression. Most of the times the user *thought* that the expression was complete and does not know what to do. Usually, it is due to a missing closing parenthesis or the like, and the way to cancel the expression is to press the `ESC` key. Some details about the RStudio console:

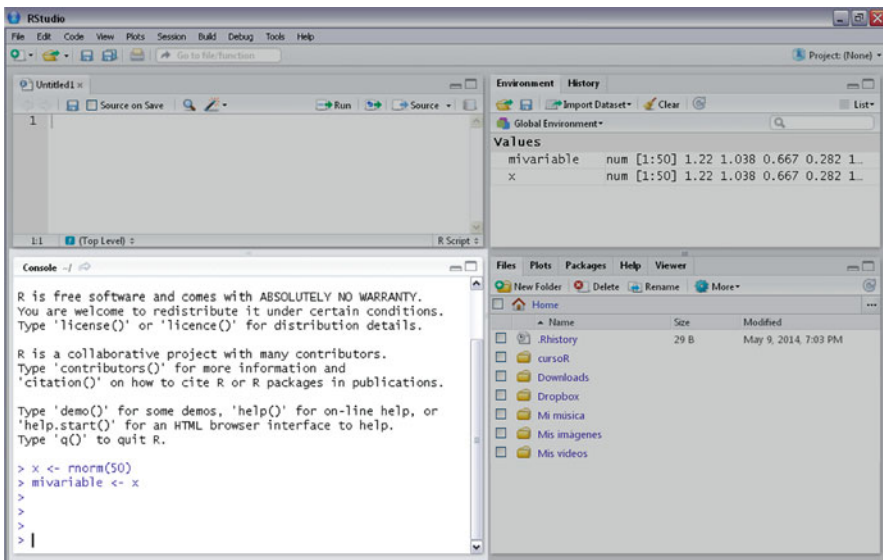


Fig. 2.3 RStudio console. The RStudio console provides interaction with R



- We can go to the console prompt using the keyboard shortcut CTRL+2 from anywhere;
- The title bar of the RStudio console contains a valuable information: the current working directory (see below);
- The arrow next to the working directory path is to show the working directory in the files pane;
- When writing an expression, we can press TAB or CTRL+SPACEBAR to see a contextual menu and select: available objects in the workspace and functions; arguments of a function (within a function); or files and folders of the working directory (within quotation marks);
- The ESC key cancels the current expression;
- CTRL+L clears the console;
- The up and down arrow keys navigate through the history.

In what follows, R code is shown in gray background. Input expressions can be written directly in the RStudio Console or script editor (or copy-pasted if you are reading the electronic version of this book). The output produced by R is shown in the book after two hash symbols (“##”) at the beginning of the line. For example, the simplest expression we can input is a number. Type the number 1 at the console and press Intro:

```
1
## [1] 1
```

We can see that the result of this input expression is a line of text with the number 1 between squared brackets followed by the number 1. The number in square brackets is an identifier that will be explained later. The result of the expression is the same number that we wrote. One step beyond would be to ask for a calculation, for example:

```
1 + 1
## [1] 2
```

Now the output is the result of the arithmetic expression. What happens if the expression is incomplete?

```
1 +
```

As you may have realized, the > symbol changes to +, denoting that the expression is not complete. The system remains in that state until either the expression is completed or the ESC key is pressed, cancelling the expression. Arithmetic expressions return the result of the operation. Another type of R expressions are logical expressions, which return the TRUE or FALSE value:

```
5 > 6
## [1] FALSE
```

An R expression can be a call to a function. In fact, it is the most used type of expression. The call to a function is always the same: the name of the function followed (imperatively) by opening and closing parenthesis and, within the parenthesis, the **arguments** of the function separated by commas. The function arguments can be provided in several ways:

- Explicitly by the name of the argument in the form “name = value”. R allows *partial matching* of names<sup>12</sup>
- Implicitly in the same order they were defined in the function.
- Using the default value defined for the function.

We can see the arguments of a function and their default values in the documentation of the function, or by pressing the TAB key after the opening parenthesis. The function `str` using a function name as an argument also returns the arguments of the function. This is a simple example of the use of a function:

```
log(pi)
## [1] 1.14473
```

where `pi` is itself an expression that gets the value of the internal object containing the value of  $\pi = 3,14159\dots$

```
pi
## [1] 3.141593
```

The `log` function gets the logarithm of a number. We can see the possible arguments of the function through the function `str`<sup>13</sup>:

```
str(log)
## function (x, base = exp(1))
```

Therefore, `log` is a function that accepts two arguments: `x`, that does not have any default value, and `base`, whose default value is the expression:

```
exp(1)
## [1] 2.718282
```

---

<sup>12</sup>This means that only the first letters of the argument name can be provided. We do not recommend that, though.

<sup>13</sup>This function returns the structure of any R object.

i.e., the  $e$  constant. Thus, the value that we get with the `log` function is the natural logarithm, i.e., with base  $e$ , of the number we pass as first argument, or with a different base if we pass the `base` argument. For example, the decimal logarithm would be:

```
log(1000, base = 10)
## [1] 3
```

What happens if we pass no arguments to the function?

```
log()
## Error in eval(expr, envir, enclos): argument "x" is
missing, with no default
```

What happens is that the expression returns an error because there is no default value for the first argument (`x`) and the function needs it. Read carefully the error messages, they usually have clues to solve the problem.

Some functions do not need any arguments to work. For example, the `seq` function generates sequences of numbers.

```
str(seq)
## function (...)
```

The dot-dot-dot (...) argument means that the function accepts an undefined number of arguments. So, will it work without arguments?

```
seq()
## [1] 1
```

It works, we get a sequence of numbers from 1 to 1 by steps of 1, i.e., the number 1. We can find out more about the arguments a function accepts using any of the following expressions<sup>14</sup>:

```
help("seq")
?seq
```

The documentation for the function is then shown in the Help tab, lower-right pane. As the function does not need any argument to work, could we use it without parenthesis?

---

<sup>14</sup>Pressing the F1 key when the cursor is in a function name also works.

```
seq
## function (...)
## UseMethod("seq")
## <bytecode: 0x7faede18c658>
## <environment: namespace:base>
```

The answer is no, because in R, every symbol is an object. The expression `seq` without parenthesis is the symbol of the function `seq`, and what R returns is the *content* of the function, i.e., its code.<sup>15</sup>

Let us finish this subsection with an explanation of the mysterious `[1]` at the beginning of the R console output. It is meaningless when the output is only one value, but when the output is a set of values that occupy more than one line, it is useful. The number at the beginning of each line is the index of the first element in the row. You will learn more about lengths and indices later on, just see the following example in which a vector with 20 random variates from a normal distribution are generated. The `[19]` at the beginning of the last row indicates that the first value in that line is the 19th value in the vector.

```
rnorm(20)
## [1] 0.05460517 1.70767743 -1.09437298
## [4] -0.28928182 2.20741296 0.51874901
## [7] -1.40491794 2.01486448 -1.18815834
## [10] 0.19038081 -1.16973591 -0.03808156
## [13] 2.35420426 1.39342626 -0.56033236
## [16] -0.67145938 0.49243855 -1.17939052
## [19] -1.05871745 1.13790261
```

Note that you might get a different number of elements per row, as the output width has been set to 50 characters for the generation of the book's code, you can set your own preferred output with the `options` function as follows:

```
options(width = 50)
```

## *The Source Editor*

In the source editor we create text files with R expressions. Expressions can be more than one line length. In fact, when expressions are too long, it is better to split it in lines in order to make the code more readable. On the other hand, more than one expression can be placed in the same line if we separate them with a semicolon (;).

---

<sup>15</sup>In this case not all the code is shown because `seq` is a built-in, compiled function.

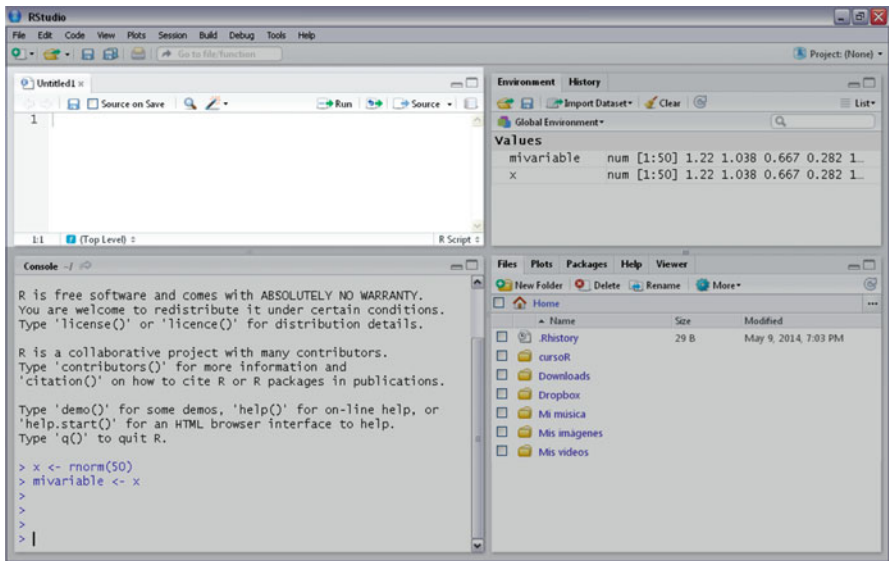
Comments can be included in the code using the # character. R scripts files have .R extension. The reason for using scripts is to reuse the code that we write once and ordinarily use it afterwards, maybe with slight changes, in quality control data analysis.

The RStudio source editor, see Fig. 2.4, has the following improvements with respect to the R editor in the R GUI:

- Code highlighting: functions, objects, numbers, and texts appear with different colors;
- Automatic indentation of code;
- Automatic closing of opening embracement characters such as parenthesis, square brackets, and quotation marks;
- CTRL+SPACEBAR and TAB keys: Provide some help in the same way that we described for the console;
- Link to the documentation using the F1 key.

Complex scripts can be run from the console or from other scripts. For example, if we have a script called `dayly.R` that performs ordinary tasks that we want to use in other scripts, e.g., loading packages, import data, etc., we can run such script with the following expression:

```
source("dayly.R")
```



**Fig. 2.4** RStudio source editor. The RStudio source editor can manage R scripts, reports, and code in other programming languages, such as C++ and Python

## The R Graphical Output

One of the R's strengths is its graphical system. Publication-quality plots can be easily produced, including mathematical symbols and formulae. The defaults of the plotting functions have been set for the minor design choices, but the user retains full control and elaborated plots can be made by tuning the graphical parameters up, functions' arguments, and global options.

The R graphical system is based in so-called *devices*. Plots are sent to devices. If no device is open, a new one is open when calling a high-level graphics function. The `grDevices` package in the R base includes a number of devices, including the appropriate one for the user's OS, and file-format devices such as pdf, jpeg, or png among others.

Devices can be managed by several functions both interactively and through scripts, thereby controlling the graphical output of our code. A global option gives the default device, which is initially set as the most appropriate for each platform. Some R packages provide further graphics devices. This is the case of RStudio, which includes its own graphics device in the lower-right pane, see Fig. 2.5.

The RStudio graphics device includes a menu bar with several options that makes life easier with devices management:

- Navigation through the graphics history;
- Zooming;
- Export files to several formats using a dialog box;
- Removing and clearing of graphics history.

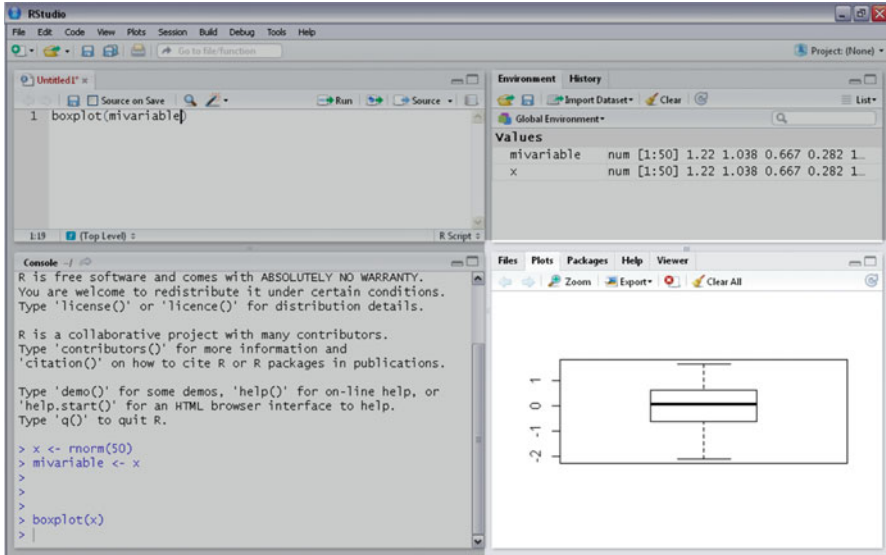


Fig. 2.5 RStudio plots tab. Plots generated in RStudio are shown in the plots tab, lower-right pane

The export menu includes three options for graphics: save as image, save as pdf, and copy to clipboard. The former two open a dialog box with export options such as the file extension (in the case of image), file path, and image size, see Fig. 2.6.

The `graphics` package contains functions for “base” graphics. Those traditional graphics are enough for most cases. The more recent development of R graphics relies on the `grid` package [24]. Both of them are included in the R base, as well as the `lattice` package, [32] aimed at elegant plots with a focus on multivariate representations. Another very popular package for graphics is the `ggplot2` package [36]. In the chapters devoted to specific quality control modelling and analysis we will see in detail how to make different types of plots and charts.

### The R Commands History

When working in the console, the commands history can be accessed using the up and down arrow keys, like in the R GUI console. In RStudio, we can also visualize all the history in the history tab, upper-right pane, see Fig. 2.7. Expressions in the history can be sent either to the console or to the source editor. Further actions such as save, open, or clean the history are available in the menu bar of the history tab.

The history can also be accessed via R code, see the documentation of the functions `loadhistory`, `savehistory`, `history`.

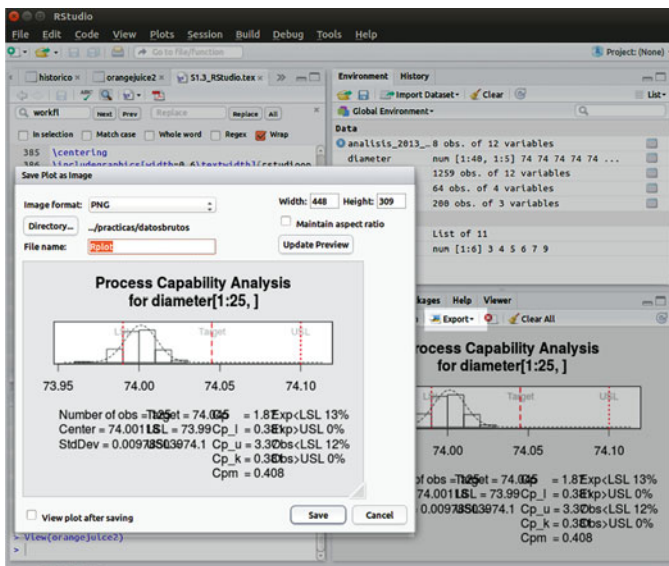
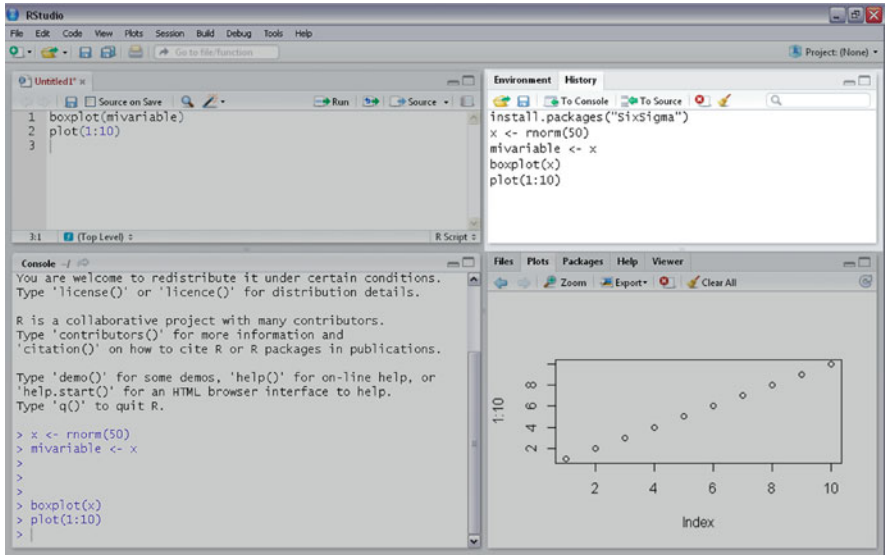


Fig. 2.6 RStudio export graphic dialog box. A preview of the image is shown along with the export options: image format and dimensions being the most relevant



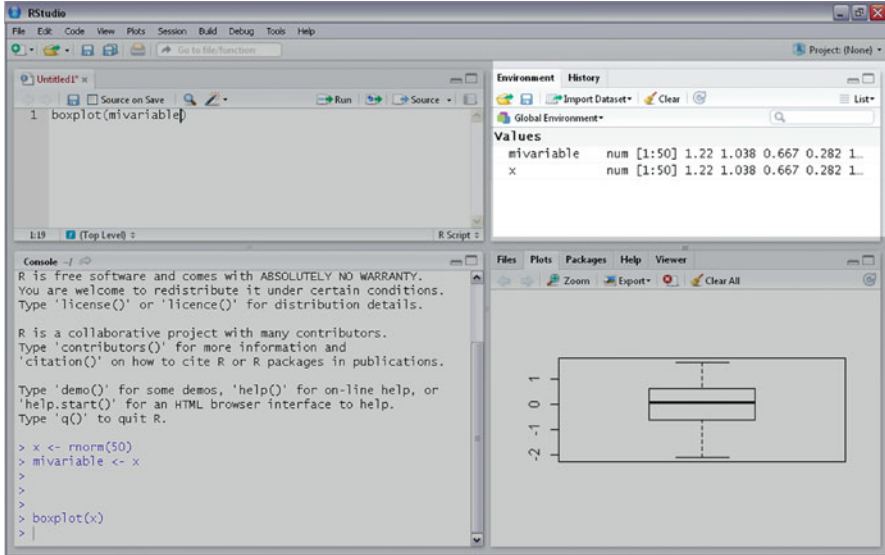
**Fig. 2.7** RStudio history. The R History can be easily consulted, searched, and used during an R session through the History tab in the upper-right pane

## *The R Workspace*

The objects that are available in R are stored in the workspace. The workspace is organized in different environments. The Global Environment is the place in which the objects we create through assignments are stored. Each loaded package has its own environment. Environments are also created for calls to functions, and the user can even create environments. For the scope of this book, just keep track of your Global Environment tab, upper-right pane, see Fig. 2.8 where you will find useful information about the objects that are available to use in your session. You can save, open, search, and clear objects in the workspace through the menu bar of the Environment tab. To make actions only over selected objects in the workspace, change the view using the upper-right icon on the menu bar from “List” (default) to “Grid,” select the objects you want to save or clear, and click the appropriate button. Remember to change again to the List view in order to be able to explore the environment. An icon for importing datasets stored in text files is also available (we will go over this later on).

The R workspace can also be accessed via R code, see the documentation of the functions `ls`, `str`, `load`, `save`, `save.image`, and `rm`.

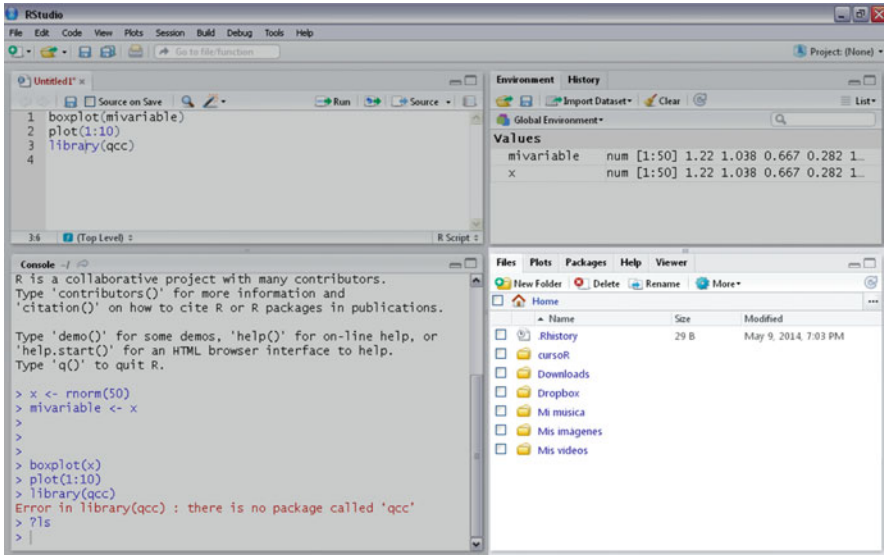




**Fig. 2.8** RStudio workspace. The R Workspace contains a list of available objects in the global environment, which can be used in R expressions during the R Session

### *The Working Directory*

A Working Directory is always associated with an R session. All tasks related to files would take this directory in the file system as the path where read or write files, for example, to read data in files, save scripts, or export plots. When an R expression contains file names, they must be provided in quotation marks ("). Relative paths from the working directory can be used. A tricky feature of RStudio is that we can pick folders and files when writing in the source editor or the console just pressing CTRL+SPACEBAR while the cursor is between quotation marks. The path to the file or folder is auto-completed as selected in the contextual menu. Furthermore, the Files tab in the lower-right pane (see Fig. 2.9) is a file explorer which can be linked to the working directory. To do so, click on the arrow icon in the title bar of the console, right after the working directory path. The reverse operation is also possible: Click the “Go to directory” button, that is, the button with the ellipsis on the upper-right side of the Files Pane; look for the directory you want to be the working directory; click on the Select button; click on the *More...* menu on the title bar of the Files tab and select the *Set as working directory* option; now the title bar of the console shows the path to your working directory, which is also visualized in the Files pane. The working directory can also be set through the Session menu, either to the Files pane location, the active source file location, or to a chosen directory. Basic operations such as creating a new folder and renaming or deleting items can be done. However, it is usually better to show the folder in a new window through the *More...* menu



**Fig. 2.9** RStudio files pane. Interaction with the file system is possible through the Files pane, including the setting and visualization of the R working directory

and work with files from there. Please note that files and folders deleted from the Files pane are permanently deleted, they do not go to the trash system folder. Finally, the default working directory can be set in the RStudio global options in the Tools menu.

The R working directory and the file system can also be accessed via R code, see the documentation of the functions `getwd` (returns the working directory), `setwd` (sets the working directory), `list.files`, `list.dirs`, and `dir`. Actually, it is common practice to include at the beginning of the scripts an expression to set the working directory, for example:

```
setwd("C:/Rprojects/myProject")
```

Recall that the backslash character (“\”) has a special meaning in R and Windows paths must be provided using either a forward slash (“/”) or a double backward slash (“\\”) to separate folders and file names. This is particularly important when copying and pasting paths from the address bar of the Windows file explorer.

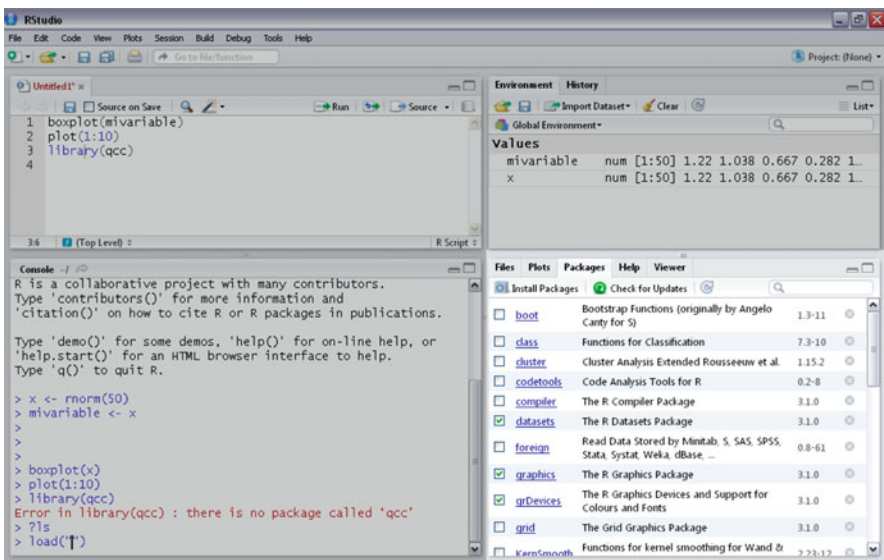
```
## Correct:
setwd("C:/myscripts")
setwd("C:\\myscripts")

## Incorrect:
setwd("C:\myscripts")
```

## Packages

R functionality is organized by means of packages. The R distribution itself includes about 30 packages. Some of them are loaded when starting R. In addition, a number of contributed packages are available, see Sect. 1.4 in Chapter 1. In order to use the functions of a package, it must be loaded in advance. Obviously, the package must be installed in the system in order to be loaded. The installation of a package is done once, while the package must be loaded in the R workspace every time we want to use it. Both operations can be done through the Packages pane of RStudio, see Fig. 2.10.

To install a package, click on the Install icon in the Packages tab menu bar. A dialog box opens where we can select whether to install the package from CRAN or from a local file. To install a package from CRAN, type the name of the package (or just the first letters to get a list) and click on the Install button. If you select to install it from a local file, a dialog box appears to search the file. This is useful for packages that are not published in official repositories, but are available from other sources. Similarly to the R software, add-on packages are regularly updated by their authors. Installed packages can be updated by clicking the Update button in the command bar of the Packages tab. From the list of installed packages we can also go to the documentation of the package by clicking on its name, remove the package from the system clicking on the icon on the right, or load the package in the workspace by selecting the check-box on the left. Nevertheless, even though the installation



**Fig. 2.10** RStudio packages. The Packages tab in the lower-right pane shows a list of installed packages with links to the documentation and command buttons to manage the packages

of packages is comfortable through the RStudio interface, it is more convenient to load the packages in the scripts as they are needed in the code using the `library` function. For example, to load the `qcc` package:

```
library(qcc)
```

Packages management can also be performed with the `install.packages` and `remove.packages` functions. Other functions related to packages are `require`, `detach`, `search`, `installed.packages`, and `available.packages`, check their documentation for details. An example of use could be to get the number of packages available at CRAN on a given date, try out by yourself and check how R grows:

```
Sys.Date()  
## [1] "2015-07-09"  
  
nrow(available.packages())  
## [1] 6797
```

## ***R and RStudio Help***

The R documentation can be accessed through the Help tab in the lower-right pane of RStudio, see Fig. 2.11. You can go there (there is even a keyboard shortcut: CTRL+3) and browse the help documentation: Manuals, packages reference, search engine and keywords, and miscellaneous material. There are other several ways of getting help in RStudio:

- The keyboard shortcut CTRL+SPACEBAR inside a function shows the basic documentation of that function;
- Pressing the F1 key when the cursor is over the name of a function or any other R object with documentation, for example a dataset;
- Using the search text box in the Help tab toolbar. A list of topics starting with the text of the search appears. If a topic is selected on this list, the documentation for the topic is shown. Otherwise, a search over all the documentation is done and the topics in which the search string appears are listed.

Typically, the documentation of a function contains the following sections:

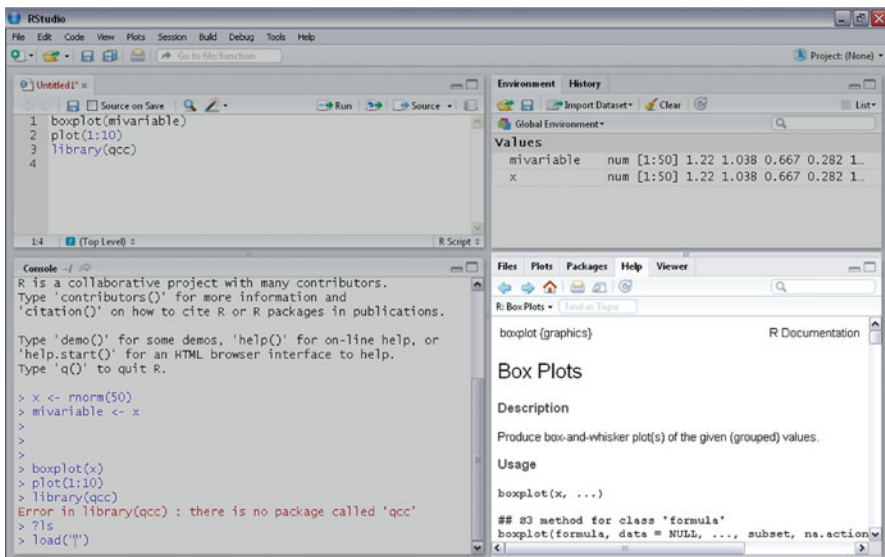
- Description: a paragraph with a description of the function;
- Usage: The function name and the arguments it expects as they are defined in the code;

- Arguments: a detailed description of each argument. This is very important in order to provide the function with objects of the correct class (see the most common errors in Sect. 2.1);
- Value: a description of the returned value. Such value can be stored in R objects;
- References;
- See Also: links to related functions or topics;
- Examples: reproducible examples which can be copy-pasted or executed by calling the `example` function with the function name as argument.

There is a special type of documentation called *vignettes*, which can include examples with output and extended explanations. If a package contains vignettes, then they are available from the package documentation index page.

R help can also be interactively accessed through R expressions. Some interesting functions to get help in such a way are (check their documentation for more details): `apropos`, `help`, `help.search`, `example`, `demo`, `vignette`, `browseVignettes`.

As for RStudio, there is a “Help” menu in the menu bar. There you can access to the R help home in the help tab, information about the RStudio version, check for updates, diagnostics, and a keyboard shortcut quick reference. Regarding help for RStudio itself, there are links to RStudio Support and to RStudio docs in the RStudio website. Note that both R and RStudio are continuously evolving, and there may have been changes with respect to what we are showing in the book, highly likely improvements for the user benefit. Check the latest documentation.



**Fig. 2.11** RStudio help. The R documentation is easily available from the RStudio interface

## 2.6 Working with Data in R

### *Data Structures*

In R, data regarding a given variable can be of a specific type, for example numeric or character. Those variables, in turn, can be included in other data structures. The simplest form is a **vector** for the data of a single variable. A **matrix** is a vector with more than one dimension. **Lists** can contain objects of different type and length. Objects whose class is `data.frame` are composed by variables which can be of different type, but with the same length. This is the most common way of organizing information for data analysis, and also for quality control. Each column of a data frame is for a variable, and each row represents an observation (item, individual, etc.) for which all the variables have been measured.

### *Classes and Data Types*

Regarding data objects in R, we can talk about them in two different senses: their class and their data type. For example, an object whose class is `data.frame` may have columns that are of type numeric, logical, or character, for example. The main basic data types in R are:

- **logical**: TRUE/FALSE;
- **integer**: Integer number;
- **double**: Real number. It can also appear as numeric or real;
- **character**: String character;

Thus, vectors whose elements are of any of those data types can be created. There are some other basic types for objects that we do not use in the book, see the documentation of the `typeof` function to learn more about them. On the other hand, those basic data are organized in data structures of different classes. The most important classes available in R to organize the information are:

- **vector**: One dimensional variable, all the values of the same type;
- **matrix**: Vector organized in rows and columns;
- **list**: List of objects that can be of different types and lengths;
- **data.frame**: Dataset organized in columns of the same length but may have different type, and rows;
- **factor**: One dimensional categorical variable. In addition to values, a factor contains information about levels and labels;
- **POSIXct, Date**: Special classes for temporal data.

The classes listed above are enough for the scope of this book, but there are many more classes in R, and new classes can be created through the programming capabilities of R. For example, objects of class `ts` are useful for working with time

series. And remember that R functions are R objects which class is “function”; Most of those extended classes are actually containers for objects of other classes, including the basic classes enumerated above. Regarding quality control, control charts can be stored in objects of class `gcc` in order to use the information of the control chart afterwards for further analysis, for example the points out of control. Such objects are usually a list of other objects of different classes and types. If in doubt, you can find out the class of an object through the `class` function. .

## Vectors

### Creating Vectors

The most basic classes in R are vectors. They are also very important because more complex data structures usually are composed by vectors. For example, the columns of a data frame with the data of a process are actually vectors with the values of different variables. Therefore, the explanations in this subsection are mostly valid for working with objects whose class is `data.frame` or `list`.

There are several ways of creating vectors. The most basic one is entering the values interactively in the console using the `scan` function. If you type on the console:

```
x1 <- scan()
```

then the console prompt changes to “1 :” waiting for the first element of the vector. Type, for example, 10 and press RETURN. Now the prompt changes again to “2 :” and waits for the second element of the vector, and so on. Enter, for example, two more values: 20 and 30. When you have finished, press INTRO without any value and the *scanning* of values finishes. Your output should look like this:

```
## 1: 10
## 2: 20
## 3: 30
## 4:
Read 3 items
```

Now you have a vector whose name is “x” on your workspace. This is what the assignment operator (“<-”) did: to assign the result of the `scan` function to the “x” symbol.<sup>16</sup> If you are using RStudio, check the Environment tab in the upper-right pane, and see the information you have at a glance. Under the “Values” group, you have the object `x` and some information about it: the data type (`num`), its length

---

<sup>16</sup>The `scan` function also accepts arguments to scan data from files and text, check the function documentation.

(from index 1 to index 3, i.e., 3 elements), and the first values of the vector (in this case all of them as there are few of them). You can always access this information from code either in the console or within a script. The following expression gets the list of objects in your workspace using the `ls` function<sup>17</sup>:

```
ls()
```

```
## [1] "x1"
```

And now you can ask for the structure of the `x` object with the `str` function:

```
str(x1)
```

```
## num [1:3] 10 20 30
```

If you input the variable symbol as a expression, you get its contents as output:

```
x1
```

```
## [1] 10 20 30
```

When using scripts, creating vectors interactively is not practical. Instead, vectors are created using the `c` function, which combines its arguments into a vector. For example, the following expression is equivalent to the above process:

```
x1 <- c(10, 20, 30)
```

We can also create vectors using operators and functions to generate sequences. For example, the `seq` function generates sequences of numbers, and the following expression is also valid to create our vector:

```
x1 <- seq(from = 10, to = 30, by = 10)
```

Sequences of integers can also be created using the colon operator (“:”) between the first and last numbers of the intended sequence. For example, the following expression creates a vector with the integer numbers from 1 to 10:

```
x2 <- 1:10; x2
```

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

Notice how in the above code we have typed two expressions in the same line, but a semicolon was used to separate them. Another useful function to generate vectors is the `rep` function, that repeats values. When working with vectors, it is common practice to combine the different ways of creating vectors:

---

<sup>17</sup>Note that the output might have more elements if further objects were created beforehand.



```
x3 <- c(rep("pinetree", 3), rep("oaktree", 2)); x3
## [1] "pinetree" "pinetree" "pinetree" "oaktree"
## [5] "oaktree"

x4 <- c(seq(from = 0, to = 1, by = 0.2), 5:9); x4
## [1] 0.0 0.2 0.4 0.6 0.8 1.0 5.0 6.0 7.0 8.0 9.0
```

The sequence of indices along a vector can also be generated with the `seq_along` function:

```
x5 <- seq_along(x4); x5
## [1] 1 2 3 4 5 6 7 8 9 10 11
```

Check that you have all the five new vectors in your workspace. We have created numeric and character vectors. Logical vectors can also be created:

```
logicalVector <- 1:6 > 3
```

Let us study this expression. We are assigning to the “logicalVector” symbol what we have on the right-hand side of the assignment expression. There, we first have the vector “1:6”, which is compared to the number “3”. This comparison is done for all the elements in the vector, and the result is another vector with the results of those comparisons, and this logical vector is assigned to the “logicalVector” object:

```
logicalVector
## [1] FALSE FALSE FALSE TRUE TRUE TRUE
```

The TRUE and FALSE values are coerced to 1 and 0, respectively, when trying to operate with them. This is useful, for example, to get the number of elements that are true in a logical vector<sup>18</sup>:

```
sum(logicalVector)
## [1] 3
```

## Vectors and Factors

Vectors and factors are different classes in R. But, actually, a factor is a kind of vector which contains information about the possible values we can find in it (levels), and the identifying labels for those possible values. For example, we might

---

<sup>18</sup>The `sum` function will be explained later.

have a variable for the machine that operates a given process, being those machines identified by letters. This identification is the label.

```
myFactor <- factor(rep(1:5, 2), labels = letters[1:5])
myFactor

## [1] a b c d e a b c d e
## Levels: a b c d e
```

In the above expression, we used the internal object `letters`, which is actually a vector with the letters of the alphabet. There is also a `LETTERS` object, guess the difference and try them in the console. We can also generate factors for a given number of replications of each level using the `gl` function:

```
factorLevels <- gl(n = 5, k = 3, labels = letters[1:5])
factorLevels

## [1] a a a b b b c c c d d d e e e
## Levels: a b c d e
```

## Lengths and Names

Vectors (and factors) lengths can be get using the `length` function. Moreover, we can assign names to each element of a vector. For example, the following expression gets the length of our first vector:

```
length(x1)

## [1] 3
```

If we want to label each element of this vector, for example because the numbers are for different weeks, we can do so using the `names` function:

```
names(x1) <- c("week1", "week2", "week3"); x1

## week1 week2 week3
##     10     20     30
```

## Accessing Vector Items

Data objects in R are indexed, and we can access each element of a vector (or factor, or any other R object as we will see later) either through its index or through its name, if such name exists. Vector indices are indicated through the square brackets symbols (“[ ]”). We can access elements of a vector for either extracting or

replacing their content. For example, the following expression extracts the third element of the “x1” vector:

```
x1[3]
## week3
##      30
```

while the following one replaces the content of the third element by the number 50:

```
x1[3] <- 50
x1
## week1 week2 week3
##      10      20      50
```

remaining the rest of the items unchanged. We can include integer vectors as index to select more than one element. For example the following expression gets the first and third elements of the x1 vector:

```
x1[c(1, 3)]
## week1 week3
##      10      50
```

We can also exclude elements from the selection instead of specifying the included elements. Thus, the previous expression is equivalent to this one:

```
x1[c(-2)]
## week1 week3
##      10      50
```

New elements can be added to a vector either creating a new vector with the original one and the new element(s) or assigning the new element to the index greater than the last one, for example:

```
c(x1, 60)
## week1 week2 week3
##      10      20      50      60
x1[4] <- 60
x1
## week1 week2 week3
##      10      20      50      60
```

To delete a vector item, we re-assign the vector resulting of the exclusion of such element:

```
x1 <- x1[c(-4)]
x1
## week1 week2 week3
##      10      20      50
```

If the elements of a vector has names, then the selection can also be done through such names as follows:

```
x1["week1"]
## week1
##      10
```

When working with data in R, it is very common to select elements of an object through logical vectors. Hence, instead of using numerical vectors as indices, we can use logical vectors of the same length than the vector, and the result will be a vector with the elements of the original vector whose indices are TRUE in the logical vector. For example, for the above selection we could make the following selection using logical indices:

```
x1[c(TRUE, FALSE, TRUE)]
## week1 week3
##      10      50
```

The combination of logical expressions and index selection is what makes this strategy powerful for data analysis. For example, to get the values of the vector that are greater than 15, we would use the following expression:

```
x1[x1 > 15]
## week2 week3
##      20      50
```

First, the expression `x1 > 15` is evaluated, returning the logical vector `{FALSE, TRUE, TRUE}`. Then, the selection is done returning only the second and third elements of the vector, which are the ones that fulfill the condition. See Appendix C for further logical operators.

## Ordering Vectors

Two functions are related with the ordering of vectors. Let us create a random vector to illustrate them. The following expressions are to get a random sample of size 10 from the digits 0 to 9. The `set.seed` function sets the *seed* in order to make the example reproducible, see `?RNG` to get help about random numbers generation with R.

```
set.seed(1234)
x6 <- sample(0:9, 10, replace = TRUE); x6
## [1] 1 6 6 6 8 6 0 2 6 5
```

The `sort` function returns the values of the vector ordered:

```
sort(x6)
## [1] 0 1 2 5 6 6 6 6 6 8
```

The `order` function returns the indices of the ordered values of the original vector, i.e., the first element is the index of the minimum value in the original vector, and so on:

```
order(x6)
## [1] 7 1 8 10 2 3 4 6 9 5
```

This function is very useful for sorting datasets as we will see later. Both the `order` function and the `sort` function accept a “decreasing” argument to get the reverse result. In addition, the `rev` function reverses the order of any vector, for example, the following expressions are equivalent:

```
sort(x6, decreasing = TRUE)
## [1] 8 6 6 6 6 6 5 2 1 0
rev(sort(x6))
## [1] 8 6 6 6 6 6 5 2 1 0
```

## Operating with Vectors

There are two types of operations we can perform over a vector, namely:

- Operations over all elements of a vector as a whole. A function is applied using all the elements in the vector to produce a given result, which can be a computation, some other values, a plot, etc. For example, to compute the average of all the elements in vector `x1` we can apply the `mean` function passing the vector as first argument:

```
mean(x1)
## [1] 26.66667
```

- Operations over each element of the vector, resulting on a vector of the same length with a computation over each value of the vector. For example, arithmetic operations and some mathematical functions work like that:

```
x1 + 5
## week1 week2 week3
##      15      25      55
```

At this point, let us introduce one interesting feature of R: recycling. The first expression in the above chunk of code is a sum of a vector whose length is 3 and another vector whose length is 1. To do that operation, the vector of length 1, i.e., the number 5, is coerced to a vector of length 3 *recycling* the number 5 twice. If we add a vector of length 2, recycling is also done, but we get a warning because the length of the first vector is not a multiple of the second one:

```
x1 + c(5, 6)
## Warning in x1 + c(5, 6): longer object length is not
## a multiple of shorter object length
## week1 week2 week3
##      15      26      55
```

In this case, the 5 has been recycled once to complete a 3-length vector. Mathematical functions which require a single value as argument return vectors with the result of the function over each value of the original vector. For example, the `sqrt` function returns the square root of a number:

```
sqrt(x1)
##      week1      week2      week3
## 3.162278 4.472136 7.071068
```

## Matrices

### Creating and Accessing Matrices

A matrix is actually a vector organized in rows and columns. All the elements must be of the same type. The most common way of creating matrices is through the `matrix` function, whose main arguments are: (1) the vector with all elements of the matrix; (2) the number of rows; and (3) the number of columns. The data are added by columns, unless the `byrow` argument is set to TRUE:

```
myMatrix <- matrix(c(10, 20, 30, 40, 12, 26, 34, 39),
  nrow = 4, ncol = 2); myMatrix

##      [,1] [,2]
## [1,]  10  12
## [2,]  20  26
## [3,]  30  34
## [4,]  40  39
```

We can extract and replace parts of a matrix in the same way as in vectors. The only difference is that now we have two indices rather than one inside the squared brackets, separated by a comma. The first one is for the row index, and the second one is for the column index. We can extract a whole row (column) by leaving the second (first) index empty:

```
myMatrix[3, 2]

## [1] 34

myMatrix[1, ]

## [1] 10 12

myMatrix[, 1]

## [1] 10 20 30 40
```

Notice that in the Environment tab of the upper-right pane of RStudio, the matrix is under the “Data” group, instead of the “Values” one. As matrices have two dimensions, i.e., rows and columns, they can be visualized in the RStudio data viewer by clicking on the icon on the right of the list. The structure of the matrix can be also get using the `str` function. See how now the lengths of the two dimensions are shown, i.e., four rows and two columns:

```
str(myMatrix)

##  num [1:4, 1:2] 10 20 30 40 12 26 34 39
```

## Basic Matrix Operations

We can assign names to rows and/or columns of matrices:

```
colnames(myMatrix) <- c("variable1", "variable2")
rownames(myMatrix) <- c("case1", "case2",
  "case3", "case4")

myMatrix
```

```
##      variable1 variable2
## case1         10         12
## case2         20         26
## case3         30         34
## case4         40         39
```

Marginal sums and means can be computed using the `rowSums`, `colSums`, `rowMeans`, and `colMeans` functions, for example:

```
rowSums(myMatrix)

## case1 case2 case3 case4
##      22      46      64      79

colMeans(myMatrix)

## variable1 variable2
##          25.00      27.75
```

See Appendix C for more examples of matrix operations. Arrays of higher dimensions are possible in R through the `array` function.

## *Lists*

### **Creating Lists**

Lists are data structures that can contain any other R objects of different types and lengths. Such objects can be created within the own definition of the list, or taken from the workspace. The elements of a list can also be named, typically this is done when creating the list. In the following example, we create a list whose name is “myList,” and has three components.

```
myList <- list(matrix = myMatrix, vector1 = x1, x2)
myList

## $matrix
##      variable1 variable2
## case1         10         12
## case2         20         26
## case3         30         34
## case4         40         39
##
## $vector1
## week1 week2 week3
##      10      20      50
```



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

See the printing of the list. The first two elements are shown with its name preceded by a \$ symbol. This is because we named them in the list definition. The third element had no name and it is identified by its index between double square brackets `[[3]]`.

## Accessing Lists

Similarly to vectors, the components of a list are indexed, and we can extract each element of the list either by its index or by its name. In the latter case, we can use the \$ operator. See the following examples:

```
myList$vector1
## week1 week2 week3
## 10 20 50

myList[[1]]
## variable1 variable2
## case1 10 12
## case2 20 26
## case3 30 34
## case4 40 39

myList["vector1"]
## $vector1
## week1 week2 week3
## 10 20 50

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

myList$matrix[, 2]
## case1 case2 case3 case4
## 12 26 34 39
```

The difference between simple and double squared brackets is that when using double squared brackets, we get the original object that is within the list, of its own class, e.g., matrix. On the contrary, if we do the extraction using the single squared

brackets like in vectors, we get an object of class list. This makes possible to select more than one element in the list, for example:

```
myList[c(2,3)]

## $vector1
## week1 week2 week3
##      10      20      50
##
## [[2]]
## [1]  1  2  3  4  5  6  7  8  9 10
```

Notice that we can extract elements from the inner components of a list, for example a column of the matrix that is the first element of the list. We can also replace parts of an object as we had done with vectors and matrices:

```
myList$matrix[, 2, drop = FALSE]

##          variable2
## case1             12
## case2             26
## case3             34
## case4             39

myList$matrix[1, 2] <- 120
myList$matrix

##          variable1 variable2
## case1             10        120
## case2             20         26
## case3             30         34
## case4             40         39
```

You can see the structure of a list in the workspace by looking at the Environment tab in the upper-right pane of RStudio, or using the `str` function:

```
str(myList)

## List of 3
## $ matrix : num [1:4, 1:2] 10 20 30 40 120 26 34..
## ..- attr(*, "dimnames")=List of 2
## .. ..$ : chr [1:4] "case1" "case2" "case3" "c"..
## .. ..$ : chr [1:2] "variable1" "variable2"
## $ vector1: Named num [1:3] 10 20 50
## ..- attr(*, "names")= chr [1:3] "week1" "week"..
## $      : int [1:10] 1 2 3 4 5 6 7 8 9 10
```

Notice that the structure of a list shows the structure of each element of the list. In the Environment tab, RStudio upper-right pane, the number of elements of the list is shown, and by clicking on the left-side icon next to the name of the list, the list is expanded to show the structure of each element of the list.

## *Data Frames*

The usual way of working with data is by organizing them in rows and columns. It is common that we have our data in such a way, either from spreadsheets, text files, or databases. Columns represent variables, which are measured or observed in a set of items, represented by rows. The class of R objects with such structure is the `data.frame` class. We refer to them as data frames hereon. Recall that matrices are also organized in rows and columns. The difference is that a matrix can only contain data of the same type, for example numbers or character strings. However, the columns of a data frame can be of different types, e.g., a numerical column for the measurement of a quality characteristic, another one logical stating whether the item is nonconforming, another one a factor for the machine where the item was produced, and so on.

### **Creating Data Frames**

Normally, we will import data to data frames from files. Nevertheless, sometimes we need to create data frames from other R objects or by generating vectors. We create data frames with the function `data.frame`

```
myData <- data.frame(type = c("A", "A", "B",  
                             "C", "C", "C"),  
                    weight = c(10.1, 20.3, 15.2,  
                               13.4, 23.2, 8.1))
```

```
myData  
  
##   type weight  
## 1    A  10.1  
## 2    A  20.3  
## 3    B  15.2  
## 4    C  13.4  
## 5    C  23.2  
## 6    C   8.1
```

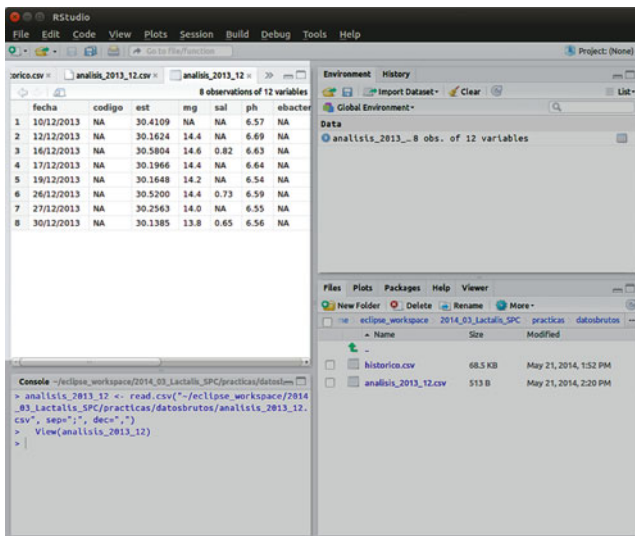
## Accessing Data Frames

Data frames are actually a sort of combination of lists, matrices, and vectors. Look at the Environment tab in the upper-right pane of RStudio. Equally to matrices, data frames are under the “Data” group and can be visualized in the RStudio data viewer: click on the right icon to show the data as a new tab in the RStudio source pane, see Fig. 2.12. In that sense, a data frame is a matrix with rows and columns. On the other hand, the information shown about the data frame is the number of observations (rows) and the number of variables (columns). Notice that the expand/collapse icon for list objects explained above is also next to the data frame name. If you click on it, the structure of each column is shown. Let us see the structure of the data frame using the `str` function:

```
str(myData)

## 'data.frame': 6 obs. of 2 variables:
## $ type : Factor w/ 3 levels "A","B","C": 1 1 2..
## $ weight: num 10.1 20.3 15.2 13.4 23.2 8.1
```

Therefore, a data frame is a list of columns, and each column is a vector. Similarly to lists, we can access data frame columns by names using the `$` operator, or by index:



**Fig. 2.12** RStudio data viewer. Matrices and data frames can be visualized in the data viewer. A new tab is open when clicking the icon right to the object in the Environment tab

```
myData$type
## [1] A A B C C C
## Levels: A B C

myData[1]
##   type
## 1   A
## 2   A
## 3   B
## 4   C
## 5   C
## 6   C
```

Notice that the access by name is equivalent to the access using double squared brackets. The difference is whether the result is a data frame or a vector. As a two dimensional data object, we can also access data frame elements in the matrix fashion:

```
myData[3, ]
##   type weight
## 3    B   15.2

myData[myData$weight < 15, ]
##   type weight
## 1    A   10.1
## 4    C   13.4
## 6    C    8.1
```

Sometimes, we need to get the number of rows or the number of columns of a data frame to be used in expressions. We can get them with the following expressions:

```
nrow(myData)
## [1] 6

ncol(myData)
## [1] 2
```

Data frames rows and columns have always names. Even if they are not available when creating the data frame, R assign them: for columns, using the letter V followed by a number (V1, V2, ...); for rows, the default names are row indices. Rows and column names can be consulted and changed afterwards in the same way

we explained above for factors and vectors, see the following examples<sup>19</sup> (we first create a copy of the data frame):

```
myEditedData <- myData
colnames(myEditedData)

## [1] "type" "weight"

colnames(myEditedData)[2] <- "itemWeight"
rownames(myEditedData)

## [1] "1" "2" "3" "4" "5" "6"

rownames(myEditedData) <- paste("case",
                                rownames(myEditedData),
                                sep = "_")

myEditedData

##      type itemWeight
## case_1      A      10.1
## case_2      A      20.3
## case_3      B      15.2
## case_4      C      13.4
## case_5      C      23.2
## case_6      C       8.1
```

## Ordering, Filtering, and Aggregating Data Frames

We already know that data frame columns are vectors. Therefore we can use the functions explained for vectors in data frames. For example, to sort the data frame created above by the weight column, we use the extracting strategy by means of the squared brackets, passing as row indices the result of the `order` function over the column (or columns) of interest:

```
myData[order(myData$weight), ]

##   type weight
## 6    C    8.1
## 1    A   10.1
## 4    C   13.4
## 3    B   15.2
## 2    A   20.3
## 5    C   23.2
```

---

<sup>19</sup>We use the `paste` function to get a sequence of character strings, see Appendix C to see more functions to work with strings.

For filtering (subsetting in R jargon) data frames, in addition to the use of indexing, we can use the `subset` function, whose use is more intuitive: the first argument is the data frame to be subset, and the second one a logical expression with the condition. Further options can be used, see the documentation of the function:

```
subset(myData, weight > 15)

##   type weight
## 2    A   20.3
## 3    B   15.2
## 5    C   23.2
```

On the other hand, the `aggregate` function allows us to get subtotals of numerical variables by categorical variables in a data frame. A special type of expression is used as the first argument of the function: a **formula**. A formula is an expression with two sides, separated by the symbol `~`. It is mainly used to specify models (see Chapter 5) in the form of `y ~ model`, where `y` is the response variable and `model` can include several independent variables and their relationship. For aggregating data, the idea is that the `y` in the formula left-hand side is the variable that we want to aggregate, and the `model` are the criteria by which we want to aggregate the data. For example, if we want to get the sum of `weight` by `type` in our data frame:

```
aggregate(weight ~ type, data = myData, sum)

##   type weight
## 1    A   30.4
## 2    B   15.2
## 3    C   44.7
```

where the third argument can be any function over a vector, typically aggregation functions, see Appendix C.

## Editing Data Frames

We use assignment expressions to edit, add, or remove elements of a data frame. Changing values in a data frame is done in the same way as in vectors or matrices. For example, to change the third observation of the second column:

```
myData

##   type weight
## 1    A   10.1
## 2    A   20.3
## 3    B   15.2
## 4    C   13.4
## 5    C   23.2
## 6    C    8.1
```

```
myData[3, 2] <- 22.2
myData
##   type weight
## 1    A  10.1
## 2    A  20.3
## 3    B  22.2
## 4    C  13.4
## 5    C  23.2
## 6    C   8.1
```

We can add new columns to a data frame as follows. For example, imagine we want to inspect the items in the data frame at a random order to check the measurements. Then we add the `randomorder` column as follows<sup>20</sup>:

```
set.seed(1)
myData$randomorder <- sample(1:6)
myData
##   type weight randomorder
## 1    A  10.1           2
## 2    A  20.3           6
## 3    B  22.2           3
## 4    C  13.4           4
## 5    C  23.2           1
## 6    C   8.1           5
```

Note that if we do the assignment over an existing column, it is overwritten. Sometimes this is what we want to do, but some others we are unexpectedly losing data. To remove a column, we assign the special value `NULL` to it:

```
myData$randomorder <- NULL
myData
##   type weight
## 1    A  10.1
## 2    A  20.3
## 3    B  22.2
## 4    C  13.4
## 5    C  23.2
## 6    C   8.1
```

Computed columns are easy to add to our data frames. The operation is similar to what we do in spreadsheets with formulas, for example to add values in columns,

---

<sup>20</sup>We first fix the seed in order to make the example reproducible.



or any other operations over data, and then copy the formulas throughout the rows, and so on. In this case it is more straightforward. Imagine we want to compute a column with the proportion over the total each item represents. The following simple expression does that, and it is ready to further work with it:

```
myData$proportion <- myData$weight/sum(myData$weight)
myData
```

##	type	weight	proportion
## 1	A	10.1	0.10380267
## 2	A	20.3	0.20863309
## 3	B	22.2	0.22816033
## 4	C	13.4	0.13771840
## 5	C	23.2	0.23843782
## 6	C	8.1	0.08324769

## Special Data Values

### Missing Values

Missing values treatment is a quite important topic in data analysis in general, and in quality control in particular, especially in early stages of data cleaning. Missing values are represented in R by the special value NA (not available). If we try to do computations over vectors that include NAs, for example the mean, we will get NA as a result, unless the argument `na.rm` (remove NAs) is set to TRUE. Such argument is available in a number of functions and methods, but not always. It may happen that NA values should actually have a value, but it was not correctly identified when creating the data object. Then we can assign other values to NAs. For that purpose (and others) the `is.na` function is very useful. First, let us create a new column in our data frame to illustrate NAs. Suppose we measured the content of salt of each element in the data frame in addition to the weight. Unfortunately, for some reason the measurements could have not be taken for all of the items. We add this new information as we learnt above:

```
myData$salt <- c(2.30, 2.15, 2.25, 2.17, NA, 2.00)
myData
```

##	type	weight	proportion	salt
## 1	A	10.1	0.10380267	2.30
## 2	A	20.3	0.20863309	2.15
## 3	B	22.2	0.22816033	2.25
## 4	C	13.4	0.13771840	2.17
## 5	C	23.2	0.23843782	NA
## 6	C	8.1	0.08324769	2.00

Let us compute the means of the two numerical variables in the data frame:

```
mean(myData$weight)
## [1] 16.21667
mean(myData$salt)
## [1] NA
```

There was no problem to compute the mean weight, as all the observations are available. However, the mean salt could not be computed because there is a missing value. To overcome this situation, we must tell the mean function to omit the missing values:

```
mean(myData$salt, na.rm = TRUE)
## [1] 2.174
```

Another possible action over NAs is to assign a value. Let us suppose that the missing value is due to the fact that the item had no salt at all, i.e., the correct value should be zero. We can turn all the NAs values into zeros (or any other value) as follows:

```
myData$salt[is.na(myData$salt)] <- 0
myData
##   type weight proportion salt
## 1    A   10.1  0.10380267  2.30
## 2    A   20.3  0.20863309  2.15
## 3    B   22.2  0.22816033  2.25
## 4    C   13.4  0.13771840  2.17
## 5    C   23.2  0.23843782  0.00
## 6    C    8.1  0.08324769  2.00
```

## Other Special Values in R

In addition to the NA and NULL values we have seen so far, there are other special values in R. For example, the Inf value represents the infinity:

```
1/0
## [1] Inf
-1/0
## [1] -Inf
```

Sometimes we get NaNs (not a number) when an operation cannot be done:

```
sqrt(-1)
## Warning in sqrt(-1): NaNs produced
## [1] NaN
```

The `i` symbol is used to represent complex numbers:

```
1i
## [1] 0+1i
as.numeric(1i^2)
## [1] -1
```

The following built-in constants are also available:

```
pi
## [1] 3.141593

letters
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"
## [12] "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v"
## [23] "w" "x" "y" "z"

LETTERS
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K"
## [12] "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V"
## [23] "W" "X" "Y" "Z"

month.name
## [1] "January" "February" "March"
## [4] "April" "May" "June"
## [7] "July" "August" "September"
## [10] "October" "November" "December"

month.abb
## [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul"
## [8] "Aug" "Sep" "Oct" "Nov" "Dec"
```

## *Data Types Conversion*

When creating objects, sometimes data types can be specified somehow, for example via the creating function arguments. Thus, the `data.frame` function accepts the `stringsAsFactors` argument to determine whether strings should be created as factors (default) or as character (setting the argument to `FALSE`). R tries to figure out what is the best type for a data set. For example, when creating a vector, if the input data includes only numbers, it creates a numeric vector; if the input data includes only character strings, it creates a character vector; if the input data includes both numbers and character strings, it creates a character vector to preserve all the information: numbers can be converted to strings, but strings cannot be converted to numbers. We can see the type of data in the vector using the `class` function:

```
vector1 <- c(1, 2, 3)
class(vector1)

## [1] "numeric"

vector2 <- c("one", "two", "three")
class(vector2)

## [1] "character"

vector3 <- c(1, 2, "three")
class(vector3)

## [1] "character"
```

We can also check whether an object is of a given type:

```
is.numeric(vector3)

## [1] FALSE
```

In any case, data structures and types can be converted from one type to another. For example, if we want `vector3` to be a numeric vector, we coerce the object to numeric:

```
as.numeric(vector3)

## Warning: NAs introduced by coercion

## [1] 1 2 NA
```

Note that, as the third element cannot be converted to a number, `NA` is introduced by coercion. Functions `as.xxx` and `is.xxx` are available for a number of types and classes, type `apropos ("^as [. ] ")` for a list.

## Working with Dates

Dates and times are important types of data. As was shown in Chapter 1, in quality control it is important to keep track of the sequential order in which the data were produced. Dates and times are usually stored in form of character strings, and they can be expressed in varied formats. For example, suppose we know the manufacturing date for some of the items in our data frame (DD/MM/YYYY format):

```
myData$date <- c("15/01/2015", "16/01/2015",
                 "17/01/2015", "18/01/2015",
                 "13/02/2015", "14/02/2015")
```

```
myData
```

##	type	weight	proportion	salt	date
## 1	A	10.1	0.10380267	2.30	15/01/2015
## 2	A	20.3	0.20863309	2.15	16/01/2015
## 3	B	22.2	0.22816033	2.25	17/01/2015
## 4	C	13.4	0.13771840	2.17	18/01/2015
## 5	C	23.2	0.23843782	0.00	13/02/2015
## 6	C	8.1	0.08324769	2.00	14/02/2015

We have added a character vector with the dates to the data frame. If we had included the column when creating the data frame, the column would have `factor` class. If we keep this variable as is, all the operations we do with it are referred to characters. For example, try to sort the data frame by date:

```
myData[order(myData$date), ]
```

##	type	weight	proportion	salt	date
## 5	C	23.2	0.23843782	0.00	13/02/2015
## 6	C	8.1	0.08324769	2.00	14/02/2015
## 1	A	10.1	0.10380267	2.30	15/01/2015
## 2	A	20.3	0.20863309	2.15	16/01/2015
## 3	B	22.2	0.22816033	2.25	17/01/2015
## 4	C	13.4	0.13771840	2.17	18/01/2015

It did not work because the string “13/02/2015” is the first one in a by-character order. To make R understand that a variable is a date, we need to convert the character string into a date. As you have likely guess, we do that with an `as.xxx` function. But in this case we need an important additional argument: the format in which the date is stored in the character vector. In the case at hand, we have a day/month/year format, which must be specified as follows (we overwrite the date variable):

```
myData$date <- as.Date(myData$date,
                       format = "%d/%m/%Y")
str(myData)

## 'data.frame': 6 obs. of 5 variables:
## $ type      : Factor w/ 3 levels "A","B","C": 1..
## $ weight    : num  10.1 20.3 22.2 13.4 23.2 8.1
## $ proportion: num  0.104 0.209 0.228 0.138 0.23..
## $ salt      : num  2.3 2.15 2.25 2.17 0 2
## $ date      : Date, format: ...
```

Note that now the date column is of Date type, and the data is represented in ISO format, i.e., “YYYY-MM-DD”. The format argument expects a character string indicating the pattern used in the character strings that store the dates. In our example, we are specifying that the string is formed by: (1) the day of the month in decimal format (%d); (2) a forward slash; (3) the month of the year in decimal format (%m); (4) another forward slash; and (5) the year with century (%Y). Check the documentation for the `strptime` topic for more options. Now we can sort the data frame by date:

```
myData[order(myData$date), ]

##   type weight proportion salt      date
## 1  A   10.1  0.10380267  2.30 2015-01-15
## 2  A   20.3  0.20863309  2.15 2015-01-16
## 3  B   22.2  0.22816033  2.25 2015-01-17
## 4  C   13.4  0.13771840  2.17 2015-01-18
## 5  C   23.2  0.23843782  0.00 2015-02-13
## 6  C    8.1  0.08324769  2.00 2015-02-14
```

It can also be useful to create variables for the year, month, etc. for aggregation, classification, stratification, or any other purpose. For example, if we store the week we can plot control charts where the groups are the weeks. We use the `format` function in the reverse sense, i.e., we turn dates into character strings, see the following examples:

```
myData$year <- format(myData$date, "%Y")
myData$month <- format(myData$date, "%Y")
myData$monthyear <- format(myData$date, "%Y-%m")
myData$week <- format(myData$date, "%Y-W%V")
myData[, c(5:9)]

##           date year month monthyear      week
## 1 2015-01-15 2015  2015    2015-01 2015-W03
## 2 2015-01-16 2015  2015    2015-01 2015-W03
## 3 2015-01-17 2015  2015    2015-01 2015-W03
```

```
## 4 2015-01-18 2015 2015 2015-01 2015-W03
## 5 2015-02-13 2015 2015 2015-02 2015-W07
## 6 2015-02-14 2015 2015 2015-02 2015-W07

str(myData)

## 'data.frame': 6 obs. of 9 variables:
## $ type      : Factor w/ 3 levels "A","B","C": 1..
## $ weight    : num 10.1 20.3 22.2 13.4 23.2 8.1
## $ proportion: num 0.104 0.209 0.228 0.138 0.23..
## $ salt      : num 2.3 2.15 2.25 2.17 0 2
## $ date      : Date, format: ...
## $ year      : chr "2015" "2015" "2015" "2015" ..
## $ month     : chr "2015" "2015" "2015" "2015" ..
## $ monthyear : chr "2015-01" "2015-01" "2015-0" ..
## $ week      : chr "2015-W03" "2015-W03" "2015" ..
```

## 2.7 Data Import and Export with R

In the previous section we have created all data from scratch. There are many situations in which we will use such strategy. However, raw data usually comes from external sources, either because they are automatically recorded during the process, or stored in databases, or manually entered in spreadsheets. The easiest way to import data in R is using .csv files. CSV stands for Comma Separated Values, and .csv files are text files in which each line corresponds to an observation of a dataset, and the values for each column are separated by a comma. Actually, the comma can be substituted by another value, for example when the comma is used as decimal point, then semicolons are used instead of commas to separate columns. The main advantage of using .csv files is that they can be generated by most of the applications that storage data, such as spreadsheets, databases, etc. Furthermore, .csv files can be opened and edited in spreadsheets applications such as **Microsoft Office** or **LibreOffice**, for which most of the users are already trained.

In the following, we will explain how to get data into R from .csv files. At the end of the section, some directions are provided to import data from other sources. A .csv file is available for downloading from the book's companion website.

### *Importing .csv Files*

In manufacturing it is common that PLCs (Programmable Logic Controllers) record data regarding product quality features. Quite often such recording machines can

automatically generate data in .csv files. In such a case the files are ready to work with them in R. However, if we are exporting data from spreadsheets, we must take into account that the resulting file will only contain text, and what we see on the screen may be different than what we get on the file. If the data on the file does not correspond with what we want, then formats, formulas, or other application-specific options might be the cause. Remove all the formats in numbers and characters. It is also recommended to do the computations in R rather than using formulas in the spreadsheet. Make sure the data in each column are consistent, for example you do not use different data types in the same columns (text and numbers). Once you have your data ready for exporting, select the “Save as ...” menu option of your spreadsheet application and select the .csv format in the “File type” list. Search the location where you want to save the file, for example your R working directory, choose a name, and save the file. Depending on your system locale configuration, the software usually decides the decimal point symbol and the separator for values. For example, if your system is in English, the decimal point symbol will be the period, and the separator, the comma; but if your system is, for example, in Spanish, then the decimal point symbol will be the comma, and the separator, the semicolon. These two formats are the most common ones.

For the examples below, you need to download the file <http://www.qualitycontrolwithr.com/lab.csv> to your working directory. You can go to your browser and download it as any other file. Alternatively, you can use the `download.file` function<sup>21</sup>:

```
download.file(  
  url = "http://emilio.lcano.com/qcrbook/lab.csv",  
  destfile = "lab.csv")
```

Now that you have a .csv file on your working directory, you can explore it. If you click the file in the Files pane of RStudio, the text file is opened in the source pane. This format is difficult to manage from a text editor, so take a look just to see how it looks like, and close it. Before importing the data into R, open the .csv file with your spreadsheet application, for example Microsoft Excel. Double-clicking the file in a files explorer window should work, but if it does not, use the “File / Open ...” menu of your spreadsheet application and search the file. It is possible that the spreadsheet application asks for the format of your data. If so, just select the period as decimal symbol and the comma as values separator. See how the data inside the .csv file looks like your usual spreadsheets, without formats, though. Now you can close the file, from now on we will work with the data in R.

---

<sup>21</sup>We use a different URL within the `download.file` function as it fails in redirecting URLs.

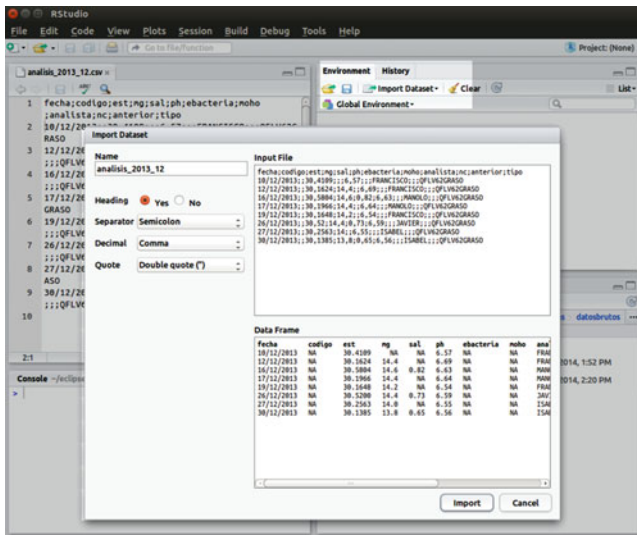


## Importing Data from Text Files

RStudio includes a functionality to import data from files. In the Environment tab, upper-right pane (see Fig. 2.8), the “Import Dataset” menu has two options: “From Text File ...” and “From Web URL ...”. The former opens a dialog box to search a text file. Select the `lab.csv` file that you downloaded and click on Open. A dialog box appears, see Fig. 2.13. In this dialog box, we can see on the right how the text input file looks like (top), and a preview of the data frame that is to be created when clicking on the Import button. On the left side we can tune the import options, which are automatically detected:

- The name of the data frame in your workspace;
- Whether the file contains headings, i.e., the first row contains the variables’ names;
- The characters that define the separator, the decimal point symbol, and the text quotes;
- The value for empty strings, NA by default;
- Whether importing strings as factors (default). Unchecking the box, string columns are imported as character vectors.

Accept the default settings and click on the Open button. Several things happen after importing data. Check your workspace in the Environment tab, upper-right pane. Now you have a data frame named `lab` in your workspace, under the “Data” group. Expand the structure using the left icon to see the variables data types.



**Fig. 2.13** RStudio import dataset dialog box. From the Environment tab we can import data in text files through the Import Dataset menu

Moreover, RStudio opens automatically the data viewer to visualize the data frame in the source pane. On the other hand, take a look to your console. After importing the data, you should have something similar to this:

```
> lab <- read.csv("<your_path>/lab.csv")
> View(lab)
```

where `<your_path>` is the path where you downloaded the `.csv` file, i.e., your working directory if you used the previous code. The second expression is the one that opened the data frame in the data viewer. It is just what RStudio does when clicking on the icon to the right of a data frame or matrix in the Environment tab. The first expression is the interesting one. Importing files from the import dataset menu is useful to explore data files, or to import a static data file once and then save the data processing as specific R data files (extension `.RData`). However, the usual way of working is that data files are regularly updated, either adding rows to the files or adding files to folders. Thus, it is more efficient to automate the data import in the scripts, and we do that with the `read.csv` function above. Note that the only argument that was included was the file path, as the rest of the default options are valid for standard `.csv` files as it is the case. The `read.csv` function is actually a wrapper of the `read.table` function, check the documentation for more details and options.

## Data Cleaning

Now we have the data available in our system, but raw data is likely to contain errors. Before applying the methods described in the following chapters, make sure that your data are ready for quality control. An exploratory data analysis should be made to detect possible errors, find outliers, and identify missing values. Some examples are given below. The first thing we must do is to verify if the data frame is what we expect to be. Check the number of rows, number of columns and their type, either in the RStudio environment tab or in the console:

```
str(lab)

## 'data.frame': 1259 obs. of 7 variables:
## $ date : Factor w/ 250 levels "", "01/02/2012"..
## $ fat : num 14 13 13 13 13.5 12.5 13 12.5 1..
## $ salt : num NA NA 1.2 NA NA NA NA NA 1.14 N..
## $ ph : num 6.64 6.65 6.66 6.6 6.6 6.63 6.6..
## $ analyst: Factor w/ 5 levels "analyst_1", "ana"..
## $ nc : logi FALSE FALSE FALSE FALSE FALSE ..
## $ type : Factor w/ 3 levels "TYPE_1", "TYPE_2"..
```

## Exploratory Data Analysis

Exploratory data analysis should include descriptive statistics, which is described in Chapter 5. This exploratory data analysis should also include techniques to detect errors like the ones presented hereon. Once the data has been cleaned, exploratory data analysis would continue with statistics, probability, and plotting techniques. Let us illustrate the data cleaning part of exploratory data analysis using the data we have imported from the .csv file. For the purpose of this illustration, we will use the `summary` function, which will be further explained in Chapter 5. This function produces result summaries of a given R object. Applied over a numeric vector, summary statistics are shown: the minimum, first quartile, median, mean, third quartile, and the maximum. This is enough for the moment.

## Missing Values

Let us get a summary of the `ph` variable of the `lab` data frame.

```
summary(lab$ph)
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  6.360   6.610   6.650   6.691   6.680   66.300
##      NA's
##         1
```

Notice that in addition to the summary statistics mentioned above, we are also informed about the number of missing values (`NA's`). It is strange that in a dataset of 1259 observations there is one and only one `NA`. We can look for that value using the extraction techniques learned in this chapter:

```
lab[is.na(lab$ph), ]
##           date fat salt ph  analyst    nc  type
## 22 10/01/2012  13   NA NA analyst_2 FALSE TYPE_3
```

Let us suppose that we do some research and find out that the measurement was taken but the operator forgave to record it. We know that the value is 6.6. Again, using objects' assignment and replacement we can fix that (note that we know from the previous output that row number 22 was the wrong one):

```
lab$ph[22] <- 6.6
summary(lab$ph)
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  6.360   6.610   6.650   6.691   6.680   66.300
```

and now we have no missing values for the `ph` variable. It is not always necessary to assign missing values. For example, for the other two numerical variables of the data

set, there are missing values, but we know from the own process that ph is always measured, but fat and salt are only measured for some items. Thus, it is normal having NAs in those columns. We just must be aware and take that into account when making computations:

```
summary(lab$salt)

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.6500  0.7600   0.8800   0.8797  0.9850   1.2000
##    NA's
##      1044

mean(lab$salt)

## [1] NA

summary(lab$fat)

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   11.50   13.40   13.80   13.73   14.00   15.75
##    NA's
##         87

mean(lab$fat, na.rm = TRUE)

## [1] 13.72922
```

In addition to the `is.na` function, the functions `any.na` and `complete.cases` functions are useful to manage missing values. The former returns `TRUE` if at least one of the values of the first argument is `NA`. The latter gets the row indices of a data frame whose columns are all not `NAs`. This can be useful to get only the rows of a data frame that are complete.

```
anyNA(lab$ph)

## [1] FALSE

anyNA(lab$salt)

## [1] TRUE

sum(complete.cases(lab))

## [1] 207
```

## Outliers

Outliers are another type of particular data that must be examined before applying statistics to quality control. An outlier of a data set is, as defined in [18], “a member of a small subset of observations that appears to be inconsistent with the remainder of a given sample.” Outliers or outlying observations can be typically attributed to one or more of the following causes:

- Measurement or recording error;
- Contamination;
- Incorrect distributional assumption;
- Rare observations.

Similarly to missing values, sometimes outliers are correct, or we just cannot remove them or assign a different value. In such cases, robust techniques should be applied, see [18]. But in some other cases, the value is either impossible or extremely unlikely to occur, and it should be corrected or removed. In a dataset with more variables, this removal means assigning the NA value.

To illustrate outliers, let us go back to the `ph` variable of our lab data frame. You have probably already realized that there is a strange number in the summary:

```
summary(lab$ph)
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  6.360  6.610   6.650   6.691  6.680  66.300
```

As you might have guessed, the median and mean are close to 6.6, the minimum is 6.63, but the maximum is ten times those values. It looks like there is something inconsistent. There is obviously something wrong with that value. We can descendently sort the dataset and check the first values to see if there are more extreme values:

```
head(lab[order(lab$ph, decreasing = TRUE), 1:5])
##      date  fat salt  ph  analyst
## 16 09/01/2012 13.0  NA 66.30 analyst_2
## 392 10/06/2012 12.5  NA  6.84 analyst_1
## 394 10/06/2012 15.0  NA  6.84 analyst_1
## 153 04/03/2012 13.2  NA  6.83 analyst_2
## 195 22/03/2012 13.6 0.71 6.83 analyst_4
## 393 10/06/2012 12.5  NA  6.83 analyst_1
```

We see that it is just row number 16 who has the maximum value. After some investigation, it was detected a wrong recording of the value, the value should have been 6.63. Note that this is a very common error when recording data. Again, we can fix the problem as follows:

```
lab$ph[16] <- 6.63
summary(lab$ph)

##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  6.360  6.610  6.650  6.644  6.680  6.840
```

In addition to the statistics summary, a powerful tool to identify outliers is the box plot. It will be described in Chapter 5, and a thorough explanation can also be found in [18]. It is basically the representation of the numbers in the summary, but all the possible outliers are also identified.

## Wrong Values

Finally, other wrong values may arise in the data. It is not always easy to detect wrong values. For categorical variables, a frequency table is a good way to find possible errors. Let us get a frequency table for the `analyst` variable using the function `table`. The result is the count of rows for each possible value of the variable used as argument:

```
table(lab$analyst)

##
## analyst_1 analyst_2 analyst_3 analyst4 analyst_4
##      319      288      355          1      296
```

Notice that there is an “analyst4” and an “analyst\_4”. The former has only one count, and the rest are 288 or above. Apparently, “analyst4” and “analyst\_4” are the same person, and we have again a recording error. Unfortunately, these types of errors are quite common when manually recording data in spreadsheets.

A more difficult to detect error is the one we have in the `date` column. There is no value for row 24, but it is not detected as missing value because it was imported as an empty string rather than a missing value:

```
anyNA(lab$date)

## [1] FALSE

which(lab$date == "")

## [1] 24
```

The `which` function returns the TRUE indices of a logical vector, very useful to use for data extraction. In this case, we should have created a column of type `Date` in advance, and then look for missing data over that column, because the empty string is coerced to `NA`:

```
lab$date2 <- as.Date(lab$date, format = "%d/%m/%Y")
anyNA(lab$date2)

## [1] TRUE
```

Let us fix it, supposing the correct date is 2012-01-10:

```
lab$date2[24] <- as.Date("2012-01-10")
anyNA(lab$date2)

## [1] FALSE
```

## *Exporting Data from R*

So far in this section, we have imported data from text files, and clean the data to get it ready for quality control. From here, we could follow two approaches:

- Save the import and data cleaning code and run it again at the beginning of the quality control data analysis;
- Save the clean data in a data file and import the data at the beginning of the quality control data analysis.

For the first approach, the expressions corresponding to the data import and replacement are to be saved in a script and then include an expression in the quality control analysis script to run the script via the `source` function as explained in Sect. 2.5. For the second approach, we can save the clean data in a file and then include an expression in the quality control analysis script to import the clean data. For that purpose, the counterpart function of the `read.csv` function is `write.csv`. The following expression saves our clean data in the file `lab_clean.csv`:

```
write.csv(lab,
          file = "lab_clean.csv",
          row.names = FALSE)
```

The first argument of the function is the object in the workspace that contains the data to be exported, preferably a matrix or data frame; The second argument is the path to the output file; and the third argument avoids to create a column with the row names, typically the row index unless row names have been set. Thus, the `lab_clean.csv` has the same structure as `lab.csv` but with the wrong data fixed. In fact, it is the same result as if we had edited the `.csv` file with a spreadsheet application such as Microsoft Excel to correct the data and then saved the file with a new name.

A strategy mixing both approaches is however the most efficient. The following step-by-step procedure can be followed as a guide when planning quality control data analysis:

1. Create folders structure for your quality control data analysis project. The following could be a general proposal which should be adapted to the project specifics, if any:
  - `data`: This folder shall contain the data files. It could contain sub-folders such as “rawdata” and “cleandata”, “yyyy\_mm\_dd” (one folder per day), etc.;
  - `code`: This folder shall contain the scripts;
  - `reports`: This folder shall contain the .Rmd files and their counterpart compiled reports as shown in Sect. 1.6, Chapter 1;
  - `plots`: This folder could contain the exported plots to be used by other programs;
  - `other...`: Any other folder useful for the purpose of the quality control data analysis.
2. Save the raw data file;
3. Create a script for data cleaning. This allows to keep track of the changes made, even including comments in the code for further reference and lessons learned;
4. Export the clean data in a data file with a new name (included in the data cleaning script);
5. Create scripts for the quality control data analysis. There might be several different scripts, for example for exploratory data analysis, control charts, capability analysis, etc.;
6. Create report files with the relevant results.

In addition to .csv files, data can be exported to many other formats. For example, one or more objects in the workspace can be saved in a .RData file using the `save` function. The following expression saves the `lab` data frame in the `lab.RData` file:

```
save(lab, file = "lab.RData" )
```

Later on, the data in a .RData file can be imported to the workspace with the `load` function:

```
load(file = "lab.RData")
```

It is up to the user which data and file formats to choose for their quality control data analysis. All of them have advantages and disadvantages. Depending on the use that will be done over the data, it could be better to use .csv files, e.g., when the data are bound to be used by other applications, or .RData files, if only R will make use of them. In addition to .csv and .RData, many other formats can be used in R for data import and export, see the following subsection.



## ***Importing Data from Other Sources***

Importing files from .RData or text files is the easiest and less prone-error way of getting data into R. Nevertheless, there are many more ways of importing data from different sources. Check the “R Data Import/Export” manual enclosed in the R documentation or at the R project website. The following is a list of the functions and packages that deal with importing data from common sources, check their documentation for details if you need to import data from the sources they manage:

- The `foreign` package [26] can read data coming from the main statistical packages, such as Minitab, S, SAS, SPSS, Stata, or Systat, among others;
- The `RODBC` package [28] deals with Open Database Connectivity (ODBC) sources. It originated on Windows but is also implemented on Linux / Unix / OS X. The supported databases include Microsoft SQL Server, Access, MySQL, PostgreSQL, Oracle, and IBM DB2;
- `RMySQL` [25], `RSQLite` [37], and `RPostgreSQL` [9] are the appropriate packages for their counterpart FOSS database management systems;
- `ROracle` [23] and `RJDBC` [35] work with Oracle and Java databases, respectively;
- The `XML` package [21] can make many operations with XML files;
- The `XLConnect` package [22] can read and write Microsoft Excel files directly<sup>22</sup>;
- Unstructured and distributed databases are also accessible, for example via the `RMongo` [7] and `h5` [2] packages.

More and more institutions are making their data available on the Internet. A general approach to deal with those data is to download the source file to disk as explained above to download the “lab.csv” file and then import the data into R.

## **2.8 R Task View for Quality Control (Unofficial)**

If there were a Task View for Quality Control at CRAN, it should include the following resources.

This Task View collects information on R packages for Statistical Quality Control. Statistical Quality Control applies statistics to process control and improvement. The main tools used in statistical quality control are control charts, capability analysis, and acceptance sampling. All statistical tools may be useful at some point in a quality control planning.

---

<sup>22</sup>There are more packages able to deal with Excel files, check <http://www.thertrader.com/2014/02/11/a-million-ways-to-connect-r-and-excel/>.

## *Modeling Quality*

The packages in this paragraph are installed with the base installation of R.

- The `base` package contains basic functions to describe the process variability. The `summary` function gets a numerical summary of a variable. The function `table` returns frequency tables. The functions `mean`, `median`, `var`, and `sd` compute the mean, median, variance, and standard deviation of a sample, respectively. For two variables, we can compute the covariance and the correlation with the functions `cov` and `cor`, respectively.
- The `stats` package includes functions to work with probability distributions. The functions for the density/mass function, cumulative distribution function, quantile function, and random variate generation are named in the form `dxxx`, `pxxx`, `qxxx`, and `rxxx` respectively, where `xxx` represents a given theoretic distribution, including `norm` (normal), `binom` (binomial), `beta`, `geom` (geometric), and so on, see `?Distributions` for a complete list. Linear models can be adjusted using the `lm` function. Analysis of Variance (ANOVA) can be done with the `anova` function. The `ts` and `arima` functions are available for time series analysis.

## *Visualizing Quality*

Standard plots can be easily made with the `graphics` package. It basically works as a painter canvas: you can start plotting a simple plot and then add more details. The `graphics`, `grid`, and `lattice` packages are included in the R base installation. The `grid` and `lattice` packages must be loaded before use, though.

- The `graphics` package allows to build standard plots using the `plot` (scatter plots), `hist` (histograms), `barplot` (bar plots), `boxplot` (box plots) functions. Low-level graphics can also be drawn using the functions: `points`, `lines`, `rect` (rectangles), `text`, and `polygon`. Those functions can also be used to annotate standard plots. Functions of `x` can be drawn with the `curve` function.
- The `grid` package implements a different way to create and modify plots in run time, including support for interaction.
- The `lattice` package [32] can plot a number of elegant plots with an emphasis on multivariate data. It is based in Trellis plots.
- `ggplot2` is another package [36] providing elegant plots through the grammar of graphics.
- Cause-and-effect diagrams can be drawn with the `cause.and.effect` (`qcc` package [33]) and the `ss.ceDiag` (`SixSigma` package [5]) functions.
- To make Pareto charts the functions `pareto.chart` (`qcc` package), `paretoChart` (`qualityTools` package [29]) and `paretochart` (`qicharts` package [1]) can be used.

## *Control Charts*

- The `qcc` package [33] can perform several types of control charts, including: `xbar` (mean), `R` (range), `S` (standard deviation), `xbar.one` (individual values), `p` (proportion), `np`, `c`, `u` (nonconformities), and `g` (number of non-events between events). The function `qcc` plots a control chart of the type specified in the `type` argument for the data specified in the `data` argument. For charts expecting data in groups, i.e., `xbar`, `R`, and `S` charts, the input data must be prepared with the function `qcc.groups`, whose arguments are the vector with the measurements and the vector with the groups identifiers. For attribute charts where the size of groups is needed, e.g., `p`, `np`, and `u`, the `sizes` argument is mandatory.
- The `qcc` package allows to implement customized control charts, see `demo("p.std.chart")`.
- The functions `ewma`, `cusum`, and `mqcc` in the `qcc` package are for exponentially weighted moving average control charts, cumulative sums control charts, and multivariate control charts, respectively.
- The `SixSigma` package can plot moving range control charts with the `ss.cc` function.
- The `qicharts` package provides the `qic` to plot control charts and run charts. It has also the `trc` function for multivariate data run charts.
- The `IQCC` package [3] implements `qcc` control charts with a focus on Phase I and Phase II analysis.
- The `qcr` package [12] provides quality control charts and numerical results.
- The `MSQC` package [31] is a toolkit for multivariate process monitoring.
- Control Charts Operating Characteristic (OC) curves. The `qcc` package `oc.curves` function draws operating characteristic curves which provide information about the probability of not detecting a shift in the process.

## *Capability Analysis*

- The `qcc` package `process.capability` function performs a capability analysis over a `qcc` object previously created.
- The `qualityTools` package `cp` function returns capability indices and charts.
- The `SixSigma` package contains functions to individually get the indices (`ss.ca.cp`, `ss.ca.cpk`, `ss.ca.z`). A complete capability analysis including plots can be done with the `ss.ca.study` function.
- The `mpcv` package [8] performs multivariate process capability analysis using the multivariate process capability vector.
- The `tolerance` package [41] contains functions for calculating tolerance intervals, useful to set specifications.

## *Acceptance Sampling*

- The `AcceptanceSampling` package [20] provides functionality for creating and evaluating single, double, and multiple acceptance sampling plans. A single sampling plan can be obtained with the `find.plan` function
- The `acc.samp` function in the `tolerance` package provides an upper bound on the number of acceptable rejects or nonconformities in a process.
- The `Dodge` package [15] contains functions for acceptance sampling ideas originated by Dodge [10].

## *Design of Experiments*

- Please visit the `ExperimentalDesign` Task View to see all resources regarding this topic.

## *Quality Control Reports*

- The `Sweave` function can produce `.pdf` files from `.Rnw` files, which can contain  $\text{\LaTeX}$  and R code.
- The `knitr` package [38–40] can produce `.pdf`, `.html`, and `.docx` files from `.Rmd` files, which can contain markdown text and R code.

## *CRAN Packages*

- `AcceptanceSampling`
- `base`
- `Dodge`
- `edcc`
- `ggplot2`
- `graphics`
- `grid`
- `IQCC`
- `knitr`
- `lattice`
- `mpcv`
- `MSQC`
- `qcc`
- `qcr`

- `qicharts`
- `qualityTools`
- `SixSigma`
- `spc`
- `spcadjust`
- `stats`
- `tolerance`

### ***Books***

- Cano, E.L., Moguerza, J.M., Redchuk, A.: *Six Sigma with R. Statistical Engineering for Process Improvement, Use R!*, vol. 36. Springer, New York (2012).
- Cano, E.L., Moguerza, J.M., Prieto, M.: *Quality Control with R. An ISO Standards Approach, Use R!*. Springer, New York (2015).
- Dodge, H., Romig, H.: *Sampling Inspection Tables, Single and Double Sampling*. John Wiley and Sons (1959)
- Montgomery, D.C. *Statistical Quality Control*, Wiley (2012)

### ***Links***

- <http://www.qualitycontrolwithr.com>
- <http://www.sixsigmawithr.com>
- <http://www.r-qualitytools.org>

## **2.9 ISO Standards and R**

This book follows an ISO Standards approach for quality control using R. The process of creating international standards is explained in Chapter 4. The aim of this approach is to present the standards relevant to the quality control topics, such as statistics, control charts, capability analysis, and acceptance sampling. In this section we reference some ISO Standards related to software and data, as well as the Certification issue.

### ***ISO Standards and Data***

In Sect. 2.6 it was shown how R represents dates in ISO format. In particular, according to the `strptime` topic documentation:

The default formats follow the rules of the ISO 8601 international standard which expresses a day as “2001-02-28” and a time as “14:01:02” using leading zeroes as here. (The ISO form uses no space to separate dates and times: R does by default.)

Check ISO 8601 international standard [17] for more details on date and time data representation. As explained in Sect. 2.6, using the `format` function and the `%` operator, any format can be obtained. The `ISOweek` package [4] could be useful if you are in trouble to get weeks in ISO format when using Windows.

As referenced in Sect. 2.7, part 4 of ISO 16269, Statistical interpretation of data—Part 4: Detection and treatment of outliers [18], “provides detailed descriptions of sound statistical testing procedures and graphical data analysis methods for detecting outliers in data obtained from measurement processes. It recommends sound robust estimation and testing procedures to accommodate the presence of outliers.”

Some examples in this chapter generated random numbers. ISO 28640 [19], “Random variate generation methods,” specifies methods for this technique.

Regarding data management and interchange, there are a number of international standards developed by the ISO/IEC JTC 1/SC 32, check the available standards in the subcommittee web page<sup>23</sup> for further details.

Data is becoming a relevant topic in standardization. Recently, a Big Data Study Group has been created within the ISO/IEC JTC 1 Technical Committee (Information Technology).<sup>24</sup> Keep updated on this standardization topic if your quality control data is big.

## ***R Certification***

Even though there is not a specific reference to ISO Standards in the R project documentation regarding the software, we can find in the R website homepage a link to “R Certification.” There we can find “A Guidance Document for the Use of R in Regulated Clinical Trial Environments,” a document devoted to “Regulatory Compliance and Validation Issues.” Even though the document focuses on the United States Federal Drug Administration (FDA) regulations, many of the topics covered can be applied to or adopted for other fields. In particular, the Software Development Life Cycle (SDLC) section, which is also available in the R Certification web page as a standalone document, represents “A Description of R’s Development, Testing, Release and Maintenance Processes,” which can be used for certification processes if needed, for example, in relation to ISO/IEC 12207 [16], Systems and software engineering—Software life cycle processes.

---

<sup>23</sup>[http://www.iso.org/iso/home/store/catalogue\\_tc/catalogue\\_tc\\_browse.htm?commid=45342](http://www.iso.org/iso/home/store/catalogue_tc/catalogue_tc_browse.htm?commid=45342).

<sup>24</sup><http://www.jtc1bigdatasg.nist.gov>.

## References

1. Anhoej, J.: qicharts: quality improvement charts. <http://www.CRAN.R-project.org/package=qicharts> (2015). R package version 0.2.0
2. Annau, M.: h5: interface to the 'HDF5' library. <http://www.CRAN.R-project.org/package=h5> (2015). R package version 0.9
3. Barbosa, E.P., Barros, F.M.M., de Jesus Goncalves, E., Recchia, D.R.: IQCC: improved quality control charts. <http://www.CRAN.R-project.org/package=IQCC> (2014). R package version 0.6
4. Block, U.: Using an algorithm by Hatto von Hatzfeld: ISOweek: week of the year and weekday according to ISO 8601. <http://www.CRAN.R-project.org/package=ISOweek> (2011). R package version 0.6-2
5. Cano, E.L., Moguerza, J.M., Redchuk, A.: Six sigma with R. In: Statistical Engineering for Process Improvement, Use R!, vol. 36. Springer, New York (2012). <http://www.springer.com/statistics/book/978-1-4614-3651-5>
6. Chambers, J.M.: Software for data analysis. In: Programming with R. Statistics and Computing. Springer, Berlin (2008)
7. Chheng, T.: RMongo: MongoDB client for R. <http://www.CRAN.R-project.org/package=RMongo> (2013). R package version 0.0.25
8. Ciupke, K.: Multivariate process capability vector based on one-sided model. Qual. Reliab. Eng. Int. (2014). doi:10.1002/qre.1590. R package version 1.1
9. Conway, J., Eddelbuettel, D., Nishiyama, T., Prayaga, S.K., Tiffin, N.: RPostgreSQL: R interface to the PostgreSQL database system. <http://www.CRAN.R-project.org/package=RPostgreSQL> (2013). R package version 0.4
10. Dodge, H., Romig, H.: Sampling Inspection Tables, Single and Double Sampling. Wiley, New York (1959)
11. Fellows, I.: Deducer: a data analysis gui for R. J. Stat. Softw. **49**(8), 1–15 (2012). <http://www.jstatsoft.org/v49/i08/>
12. Flores, M., Naya, S., Fernandez, R.: qcr: quality control and reliability. <http://www.CRAN.R-project.org/package=qcr> (2014). R package version 0.1-18
13. Fox, J.: The R commander: a basic statistics graphical user interface to R. J. Stat. Softw. **14**(9), 1–42 (2005). <http://www.jstatsoft.org/v14/i09>
14. Free Software Foundation, Inc.: Free Software Foundation website. <http://www.gnu.org> (2014) [Retrieved 2014-07-10]
15. Godfrey, A.J.R., Govindaraju, K.: Dodge: functions for acceptance sampling ideas originated by H.F. Dodge. <http://www.CRAN.R-project.org/package=Dodge> (2013). R package version 0.8
16. ISO/IEC JTC 1 – Information Technology: ISO/IEC 12207:2008, Systems and Software Engineering – Software Life Cycle Processes. ISO – International Organization for Standardization (2008). [http://www.iso.org/iso/catalogue\\_detail?csnumber=43447](http://www.iso.org/iso/catalogue_detail?csnumber=43447)
17. ISO TC154 – Processes, Data Elements and Documents in Commerce, Industry and Administration: ISO 8601 Data Elements and Interchange Formats – Information Interchange – Representation of Dates and Times. ISO – International Organization for Standardization (2004)
18. ISO TC69/SCS–Secretariat: ISO 16269-4:2010 – Statistical Interpretation of Data – Part 4: Detection and Treatment of Outliers. Published Standard. [http://www.iso.org/iso/catalogue\\_detail.htm?csnumber=44396](http://www.iso.org/iso/catalogue_detail.htm?csnumber=44396) (2010)
19. ISO TC69/SCS–Secretariat: ISO 28640:2010 – Random Variate Generation Methods. Published Standard. [http://www.iso.org/iso/catalogue\\_detail.htm?csnumber=42333](http://www.iso.org/iso/catalogue_detail.htm?csnumber=42333) (2015)
20. Kiermeier, A.: Visualizing and assessing acceptance sampling plans: the R package AcceptanceSampling. J. Stat. Softw. **26**(6) (2008). <http://www.jstatsoft.org/v26/i06/>
21. Lang, D.T., The CRAN Team: XML: tools for parsing and generating XML within R and S-plus. <http://www.CRAN.R-project.org/package=XML> (2015). R package version 3.98-1.3

22. Mirai Solutions GmbH: XLConnect: excel connector for R. <http://www.CRAN.R-project.org/package=XLConnect> (2015). R package version 0.2-11
23. Mukhin, D., James, D.A., Luciani, J.: ROracle: OCI based oracle database interface for R. <http://www.CRAN.R-project.org/package=ROracle> (2014). R package version 1.1.12
24. Murrell, P.: R Graphics, 2nd edn. Chapman & Hall/CRC, Boca Raton (2011)
25. Ooms, J., James, D., DebRoy, S., Wickham, H., Horner, J.: RMySQL: database interface and MySQL driver for R. <http://www.CRAN.R-project.org/package=RMySQL> (2015). R package version 0.10.1
26. R Core Team: Foreign: read data stored by minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, . . . . <http://www.CRAN.R-project.org/package=foreign> (2015). R package version 0.8-64
27. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna (2015). <http://www.R-project.org/>
28. Ripley, B., Lapsley, M.: RODBC: ODBC database access. <http://www.CRAN.R-project.org/package=RODBC> (2015). R package version 1.3-12
29. Roth, T.: qualityTools: statistics in quality science. <http://www.r-qualitytools.org> (2012). R package version 1.54 <http://www.r-qualitytools.org>
30. RStudio Team: RStudio: Integrated Development Environment for R. RStudio Inc., Boston, MA (2012). <http://www.rstudio.com/>
31. Santos-Fernández, E.: Multivariate Statistical Quality Control Using R, vol. 14. Springer, Berlin (2013). <http://www.springer.com/statistics/computational+statistics/book/978-1-4614-5452-6>
32. Sarkar, D.: Lattice: Multivariate Data Visualization with R. Springer, New York (2008). <http://www.lmdvr.r-forge.r-project.org>. ISBN 978-0-387-75968-5
33. Scrucca, L.: qcc: an r package for quality control charting and statistical process control. R News **4**(1), 11–17 (2004). <http://www.CRAN.R-project.org/doc/Rnews/>
34. Shewhart, W.: Economic Control of Quality in Manufactured Products. Van Nostrum, New York (1931)
35. Urbanek, S.: RJDBC: provides access to databases through the JDBC interface. <http://www.CRAN.R-project.org/package=RJDBC> (2014). R package version 0.2-5
36. Wickham, H.: ggplot2: elegant graphics for data analysis. In: Use R! Springer, Berlin (2009)
37. Wickham, H., James, D.A., Falcon, S.: RSQLite: SQLite interface for R. <http://www.CRAN.R-project.org/package=RSQLite> (2014). R package version 1.0.0
38. Xie, Y.: Dynamic Documents with R and Knitr. Chapman and Hall/CRC, Boca Raton, FL (2013). <http://www.yihui.name/knitr/>. ISBN 978-1482203530
39. Xie, Y.: knitr: a comprehensive tool for reproducible research in R. In: Stodden, V., Leisch, F., Peng, R.D. (eds.) Implementing Reproducible Computational Research. Chapman and Hall/CRC, Boca Raton (2014). <http://www.crcpress.com/product/isbn/9781466561595>. ISBN 978-1466561595
40. Xie, Y.: knitr: a general-purpose package for dynamic report generation in R. <http://www.yihui.name/knitr/> (2015). R package version 1.10.5
41. Young, D.S.: tolerance: an R package for estimating tolerance intervals. J. Stat. Softw. **36**(5), 1–39 (2010). <http://www.jstatsoft.org/v36/i05/>