Mark van der Loo and Edwin de Jonge

STATISTICAL DATA CLEANING

with Applications in R

WILEY

Statistical Data Cleaning with Applications in R

Statistical Data Cleaning with Applications in R

Mark van der Loo
Statistics Netherlands
The Netherlands

Edwin de Jonge
Statistics Netherlands
The Netherlands

WILEY

This edition first published 2018

Registered Offices
John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, USA
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK

Editorial Office
9600 Garsington Road, Oxford, OX4 2DQ, UK

For details of our global editorial offices, customer services, and more information about Wiley products visit us at www.wiley.com.

Wiley also publishes its books in a variety of electronic formats and by print-on-demand. Some content that appears in standard print versions of this book may not be available in other formats.

Library of Congress Cataloging-in-Publication Data

Names: Loo, Mark van der, 1976- author.| Jonge, Edwin de, 1972- author.
Title: Statistical data cleaning with applications in R / by Mark van der Loo, Edwin de Jonge
Description: Hoboken, NJ : John Wiley & Sons, 2018. | Includes bibliographical references and index. |
Identifiers: LCCN 2017049091 (print) | LCCN 2017059014 (ebook) | ISBN 9781118897140 (pdf) | ISBN 9781118897133 (epub) | ISBN 9781118897157 (cloth)
Subjects: LCSH: Statistics–Data processing. | R (Computer program language)
Classification: LCC QA276.45.R3 (ebook) | LCC QA276.45.R3 J663 2018 (print) | DDC 519.50285/5133–dc23
LC record available at https://lccn.loc.gov/2017049091

Cover design by Wiley
Cover image: © enot-poloskun/Gettyimages; © 3alexd/Gettyimages

Set in 10/12pt WarnockPro by SPi Global, Chennai, India

10 9 8 7 6 5 4 3 2 1

Contents

Foreword *xi*
About the Companion Website *xiii*

1 Data Cleaning *1*
1.1 The Statistical Value Chain *1*
1.1.1 Raw Data *2*
1.1.2 Input Data *2*
1.1.3 Valid Data *3*
1.1.4 Statistics *3*
1.1.5 Output *3*
1.2 Notation and Conventions Used in this Book *3*

2 A Brief Introduction to R *5*
2.1 R on the Command Line *5*
2.1.1 Getting Help and Learning R *6*
2.2 Vectors *7*
2.2.1 Computing with Vectors *9*
2.2.2 Arrays and Matrices *10*
2.3 Data Frames *11*
2.3.1 The Formula-Data Interface *12*
2.3.2 Selecting Rows and Columns; Boolean Operators *12*
2.3.3 Selection with Indices *13*
2.3.4 Data Frame Manipulation: The `dplyr` Package *14*
2.4 Special Values *15*
2.4.1 Missing Values *17*
2.5 Getting Data into and out of R *18*
2.5.1 File Paths in R *19*
2.5.2 Formats Provided by Packages *20*
2.5.3 Reading Data from a Database *20*
2.5.4 Working with Data External to R *21*
2.6 Functions *21*

2.6.1 Using Functions *22*
2.6.2 Writing Functions *22*
2.7 Packages Used in this Book *23*

3 Technical Representation of Data *27*
3.1 Numeric Data *28*
3.1.1 Integers *28*
3.1.2 Integers in R *30*
3.1.3 Real Numbers *31*
3.1.4 Double Precision Numbers *31*
3.1.5 The Concept of Machine Precision *33*
3.1.6 Consequences of Working with Floating Point Numbers *34*
3.1.7 Dealing with the Consequences *35*
3.1.8 Numeric Data in R *37*
3.2 Text Data *38*
3.2.1 Terminology and Encodings *38*
3.2.2 Unicode *39*
3.2.3 Some Popular Encodings *40*
3.2.4 Textual Data in R: Objects of Class Character *43*
3.2.5 Encoding in R *44*
3.2.6 Reading and Writing of Data with Non-Local Encoding *46*
3.2.7 Detecting Encoding *48*
3.2.8 Collation and Sorting *49*
3.3 Times and Dates *50*
3.3.1 AIT, UTC, and POSIX Seconds Since the Epcoch *50*
3.3.2 Time and Date Notation *52*
3.3.3 Time and Date Storage in R *54*
3.3.4 Time and Date Conversion in R *55*
3.3.5 Leap Days, Time Zones, and Daylight Saving Times *57*
3.4 Notes on Locale Settings *58*

4 Data Structure *61*
4.1 Introduction *61*
4.2 Tabular Data *61*
4.2.1 data.frame *61*
4.2.2 Databases *62*
4.2.3 dplyr *64*
4.3 Matrix Data *65*
4.4 Time Series *66*
4.5 Graph Data *68*
4.6 Web Data *69*
4.6.1 Web Scraping *70*
4.6.2 Web API *70*
4.7 Other Data *72*
4.8 Tidying Tabular Data *72*
4.8.1 Variable Per Column *74*
4.8.2 Single Observation Stored in Multiple Tables *75*

5 Cleaning Text Data *77*
5.1 Character Normalization *78*
5.1.1 Encoding Conversion and Unicode Normalization *78*
5.1.2 Character Conversion and Transliteration *80*
5.2 Pattern Matching with Regular Expressions *81*
5.2.1 Basic Regular Expressions *82*
5.2.2 Practical Regular Expressions *85*
5.2.3 Generating Regular Expressions in R *92*
5.3 Common String Processing Tasks in R *93*
5.4 Approximate Text Matching *98*
5.4.1 String Metrics *100*
5.4.2 String Metrics and Approximate Text Matching in R *109*

6 Data Validation *119*
6.1 Introduction *119*
6.2 A First Look at the `validate` Package *120*
6.2.1 Quick Checks with `check_that` *120*
6.2.2 The Basic Workflow: `validator` and `confront` *122*
6.2.3 A Little Background on `validate` and DSLs *124*
6.3 Defining Data Validation *125*
6.3.1 Formal Definition of Data Validation *126*
6.3.2 Operations on Validation Functions *128*
6.3.3 Validation and Missing Values *130*
6.3.4 Structure of Validation Functions *131*
6.3.5 Demarcating Validation Rules in `validate` *132*
6.4 A Formal Typology of Data Validation Functions *133*
6.4.1 A Closer Look at Measurement *134*
6.4.2 Classification of Validation Rules *135*
6.5 Validating Data with the `validate` Package *137*
6.5.1 Validation Rules in the Console and the `validator` Object *137*
6.5.2 Validating in the Pipeline *139*
6.5.3 Raising Errors or Warnings *140*
6.5.4 Tolerance for Testing Linear Equalities *140*
6.5.5 Setting and Resetting Options *141*
6.5.6 Importing and Exporting Validation Rules from and to File *142*
6.5.7 Checking Variable Types and Metadata *145*
6.5.8 Checking Value Ranges and Code Lists *146*
6.5.9 Checking In-Record Consistency Rules *146*
6.5.10 Checking Cross-Record Validation Rules *148*
6.5.11 Checking Functional Dependencies *149*
6.5.12 Cross-Dataset Validation *150*
6.5.13 Macros, Variable Groups, Keys *152*
6.5.14 Analyzing Output: `validation` Objects *152*
6.5.15 Output Dimensionality and Output Selection *155*

7 Localizing Errors in Data Records *157*
7.1 Error Localization *157*

7.2 Error Localization with R *160*
7.2.1 The Errorlocate Package *160*
7.3 Error Localization as MIP-Problem *163*
7.3.1 Error Localization and Mixed-Integer Programming *163*
7.3.2 Linear Restrictions *164*
7.3.3 Categorical Restrictions *165*
7.3.4 Mixed-Type Restrictions *167*
7.4 Numerical Stability Issues *170*
7.4.1 A Short Overview of MIP Solving *170*
7.4.2 Scaling Numerical Records *172*
7.4.3 Setting Numerical Threshold Values *173*
7.5 Practical Issues *174*
7.5.1 Setting Reliability Weights *174*
7.5.2 Simplifying Conditional Validation Rules *176*
7.6 Conclusion *180*

8 Rule Set Maintenance and Simplification *183*
8.1 Quality of Validation Rules *183*
8.1.1 Completeness *183*
8.1.2 Superfluous Rules and Infeasibility *184*
8.2 Rules in the Language of Logic *184*
8.2.1 Using Logic to Rewrite Rules *185*
8.3 Rule Set Issues *186*
8.3.1 Infeasible Rule Set *186*
8.3.2 Fixed Value *187*
8.3.3 Redundant Rule *188*
8.3.4 Nonrelaxing Clause *189*
8.3.5 Nonconstraining Clause *189*
8.4 Detection and Simplification Procedure *190*
8.4.1 Mixed-Integer Programming *190*
8.4.2 Detecting Feasibility *191*
8.4.3 Finding Rules Causing Infeasibility *191*
8.4.4 Detecting Conflicting Rules *191*
8.4.5 Detect Partial Infeasibility *192*
8.4.6 Detect Fixed Values *192*
8.4.7 Detect Nonrelaxing Clauses *192*
8.4.8 Detect Nonconstraining Clauses *193*
8.4.9 Detect Redundant Rules *193*
8.5 Conclusion *194*

9 Methods Based on Models for Domain Knowledge *195*
9.1 Correction with Data Modifying Rules *195*
9.1.1 Modifying Functions *196*
9.1.2 A Class of Modifying Functions on Numerical Data *201*
9.2 Rule-Based Correction with `dcmodify` *205*
9.2.1 Reading Rules from File *206*
9.2.2 Modifying Rule Syntax *207*

9.2.3 Missing Values *208*
9.2.4 Sequential and Sequence-Independent Execution *208*
9.2.5 Options Settings Management *209*
9.3 Deductive Correction *209*
9.3.1 Correcting Typing Errors in Numeric Data *209*
9.3.2 Deductive Imputation Using Linear Restrictions *213*

10 Imputation and Adjustment *219*
10.1 Missing Data *219*
10.1.1 Missing Data Mechanisms *219*
10.1.2 Visualizing and Testing for Patterns in Missing Data Using R *220*
10.2 Model-Based Imputation *224*
10.3 Model-Based Imputation in R *226*
10.3.1 Specifying Imputation Methods with `simputation` *226*
10.3.2 Linear Regression-Based Imputation *227*
10.3.3 *M*-Estimation *230*
10.3.4 Lasso, Ridge, and Elasticnet Regression *231*
10.3.5 Classification and Regression Trees *232*
10.3.6 Random Forest *235*
10.4 Donor Imputation with R *236*
10.4.1 Random and Sequential Hot Deck Imputation *237*
10.4.2 *k* Nearest Neighbors and Predictive Mean Matching *238*
10.5 Other Methods in the `simputation` Package *239*
10.6 Imputation Based on the EM Algorithm *240*
10.6.1 The EM Algorithm *241*
10.6.2 EM Imputation Assuming the Multivariate Normal Distribution *243*
10.7 Sampling Variance under Imputation *244*
10.8 Multiple Imputations *246*
10.8.1 Multiple Imputation Based on the EM Algorithm *248*
10.8.2 The `Amelia` Package *249*
10.8.3 Multivariate Imputation with Chained Equations (Mice) *252*
10.8.4 Imputation with the `mice` Package *254*
10.9 Analytic Approaches to Estimate Variance of Imputation *256*
10.9.1 Imputation as Part of the Estimator *256*
10.10 Choosing an Imputation Method *257*
10.11 Constraint Value Adjustment *259*
10.11.1 Formal Description *259*
10.11.2 Application to Imputed Data *262*
10.11.3 Adjusting Imputed Values with the `rspa` Package *263*

11 Example: A Small Data-Cleaning System *265*
11.1 Setup *266*
11.1.1 Deterministic Methods *266*
11.1.2 Error Localization *269*
11.1.3 Imputation *269*
11.1.4 Adjusting Imputed Data *271*
11.2 Monitoring Changes in Data *273*

11.2.1 Data Diff (Daff) *274*
11.2.2 Summarizing Cell Changes *275*
11.2.3 Summarizing Changes in Conformance to Validation Rules *277*
11.2.4 Track Changes in Data Automatically with `lumberjack` *278*
11.3 Integration and Automation *282*
11.3.1 Using `RScript` *283*
11.3.2 The `docopt` Package *283*
11.3.3 Automated Data Cleaning *285*

References *287*

Index *297*

Foreword

Data cleaning is often the most time-consuming part of data analysis. Although it has been recognized as a separate topic for a long time in Official Statistics (where it is called 'data editing') and also has been studied in relation to databases, literature aimed at the larger statistical community is limited. This is why, when the publisher invited us to expand our tutorial "An introduction to data cleaning with R", which we developed for the *useR!*2013 conference, into a book, we grabbed the opportunity with both hands. On the one hand, we felt that some of the methods that have been developed in the Official Statistics community over the last five decades deserved a wider audience. Perhaps, this book can help with that. On the other hand, we hope that this book will help in bringing some (often pre-existing) techniques into the Official Statistics community, as we move from survey-based data sources to administrative and "big" data sources.

For us, it would also be a nice way to systematize our knowledge and the software we have written on this topic. Looking back, we ended up not only writing this book, but also redeveloping and generalizing much of the data cleaning R packages we had written before. One of the reasons for this is that we discovered nice ways to generalize and expand our software and methods, and another is that we wished to connect to the recently emerged "tidyverse" style of interfaces to the R functionality.

What You Will Find in this Book

This book contains a selection of topics that we found to be useful while developing data cleaning (data editing) systems. The range is very broad, ranging from topics related to computer science, numerical methods, technical standards, statistics and data modeling, and programming.

This book covers topics in "technical data cleaning", including conversion and interpretation of numerical, text, and date types. The technical standards related to these data types are also covered in some detail. On the data content side of things, topics include data validation (data checking), error localization, various methods for error correction, and missing value imputation.

Wherever possible, the theory discussed in this book is illustrated with an executable R code. We have also included exercises throughout the book which we hope will guide the reader in further understanding both the software and the methods.

The mix of topics reflects both the breadth of the subject and of course the interests and expertise of the authors. The list of missing topics is of course much larger than that

what is treated, but perhaps the most important ones are cleaning of time series objects and outlier detection.

For Who Is this Book?

Readers of this book are expected to have basic knowledge of mathematics and statistics and also some programming experience. We assume concepts such as expectation values, variance, and basic calculus and linear algebra as previous knowledge. It is beneficial to have at least some knowledge of R, since this is the language used in this book, but for convenience and reference, a short chapter explaining the basics is included.

Acknowledgments

This book would have not been possible without the work of many others. We would like to thank our colleagues at Statistics Netherlands for fruitful discussions on data validation, imputation, and error localization. Some of the chapters in this book are based on papers and reports written with co-authors. We thank Jeroen Pannekoek, Sander Scholtus, and Jacco Daalmans for their pleasant and fruitful collaboration. We are greatly indebted by the R core team, package developers, and the very supportive R community for their relentless efforts.

Finally, we would like to thank our families for their love and support.

June 2017 *Mark and Edwin*

About the Companion Website

Do not forget to visit the companion website for this book:

www.data-cleaning.org

There you will find valuable materials designed to enhance your learning, including:

- supplementary materials

1

Data Cleaning

1.1 The Statistical Value Chain

The purpose of data cleaning is to bring data up to a level of quality such that it can reliably be used for the production of statistical models or statements. The necessary level of quality needed to create some statistical output is determined by a simple cost-benefit question: when is statistical output fit for use, and how much effort will it cost to bring the data up to that level?

One useful way to get a hold on this question is to think of data analyses in terms of a value chain. A value chain, roughly, consists of a sequence of activities that increase the value of a product step by step. The idea of a *statistical value chain* has become a common term in the official statistics community over the past two decades or so, although a single common definition seems to be lacking.[1] Roughly then, a statistical value chain is constructed by defining a number of meaningful intermediate data products, for which a chosen set of quality attributes are well described (Renssen and Van Delden 2008). There are many ways to go about this, but for these authors, the picture shown in Figure 1.1 has proven to be fairly generic and useful to organize thoughts around a statistical production process.

A nice trait of the schema in Figure 1.1 is that it naturally introduces activities that are typically categorized as 'data cleaning' into the statistical production process. From the left, we start with raw data. This must be worked up to satisfy (enough) technical standards so it can serve as input for consistency checks, data correction, and imputation procedures. Once this has been achieved, the data may be considered valid (enough) for the production of statistical parameters. These must then still be formatted to be ready for deployment as output.

One should realize that although the schema nicely organizes data analysis activities, in practice, the process is hardly linear. It is more common to clean data, create some aggregates, notice that something is wrong, and go back. The purpose of the value chain is more to keep an overview of where activities take place (e.g., by putting them in separate scripts) than to prescribe a linear order of the actual workflow. In practice, a workflow cycles multiple times through the subsequent stages of the value chain, until the quality of its output is good enough. In the following sections we will discuss each stage in a little more detail.

1 One of the earliest references seems to be by Willeboordse (2000).

Statistical Data Cleaning with Applications in R, First Edition. Mark van der Loo and Edwin de Jonge.

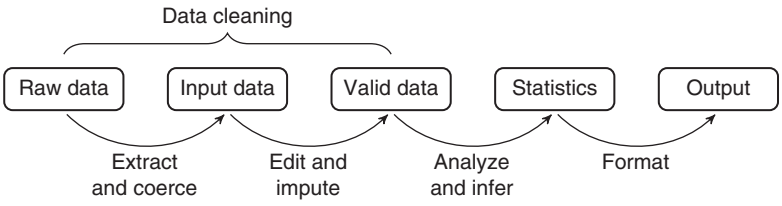

Figure 1.1 Part of a statistical value chain, showing five different levels of statistical value going from raw data to statistical product.

1.1.1 Raw Data

With *raw data*, we mean the data as it arrives at the desk of the analyst. The state of such data may of course vary enormously, depending on the data source. In any case, we take it as a given that the person responsible for analysis has little or no influence over how the data was gathered. The first step consists of making the data readily accessible and understandable. To be precise after the first processing steps, we demand that each value in the data be identified with the real-world object they represent (person, company, something else), for each value it is known what variable it represents (age, income, etc.), and the value is stored in the appropriate technical format (number and string).

Depending on the technical raw data format, the activities necessary to achieve the desired technical format typically include file conversion, string normalization (such as encoding conversion), and standardization and conversion of numerical values. Joining the data against a backbone population register of known statistical objects, possibly using procedures that can handle inexact matches of keys, is also considered a part of such a procedure. These procedures are treated in Chapters 3–5.

1.1.2 Input Data

Input data are data where each value is stored in the correct type and identified with the variable it represents and the statistical entity it pertains to. In many cases, such a dataset can be represented in tabular format, rows representing entities and columns representing variables. In the R community, this has come to be known as *tidy data* (Wickham, 2014b). Here, we leave the format open. Many data can be usefully represented in the form of a tree or graph (e.g., web pages and XML structures). As long as all the elements are readily identified and of the correct format, it can serve as Input data.

Once a dataset is at the level of input data, the treatment of missing values, implausible values, and implausible value combinations must take place. This process is commonly referred to as data editing and imputation. It differs from the previous steps in that it focuses on the consistency of data with respect to domain knowledge. Such domain knowledge can often be expressed as a set of rules, such as `age >= 0`, `mean(profit) > 0`, or `if ( age < 15 ) has_job = FALSE`. A substantial part of this book (Chapters 6–8) is devoted to defining, applying, and maintaining such rules so that data cleaning can be automated and hence be executed in a reproducible way. Moreover, in Chapter 7, we look into methodology that allows one to pick out a minimum number of

fields in a record that may be altered or imputed such that all the rules can be satisfied. In Chapter 9, we will have a formal look on how data modification using knowledge rules can be safely automated, and Chapter 10 treats missing value imputation.

1.1.3 Valid Data

Data are valid once they are trusted to faithfully represent the variables and objects they represent. Making sure that data satisfies the domain knowledge expressed in the form of a set of validation rules is one reproducible way of doing so. Often this is complemented by some form of expert review, for example, based on various visualizations or reviewing of aggregate values by domain experts.

Once data is deemed valid, the statistics can be produced by known modeling and inference techniques. Depending on the preceding data cleaning process, these techniques may need those procedures into account, for example, when estimating variance after certain imputation procedures.

1.1.4 Statistics

Statistics are simply estimates of the output variables of interest. Often, these are simple aggregates (totals and means), but in principle they can consist of more complex parameters such as regression model coefficients or a trained machine learning model.

1.1.5 Output

Output is where the analysis stops. It is created by taking the statistics and preparing them for dissemination. This may involve technical formatting, for example, to make numbers available through a (web) API or layout formatting, for example, by preparing a report or visualization. In the case of technical formatting, a technical validation step may again be necessary, for example, by checking the output format against some (json or XML) schema. In general, the output of one analyst is raw data for another.

1.2 Notation and Conventions Used in this Book

The topics discussed in this book relate to a variety of subfields in mathematics, logic, statistics, computer science, and programming. This broad range of fields makes coming up with a consistent notation for different variable types and concepts a bit of a challenge, but we have attempted to use a consistent notation throughout.

General Mathematics and Logic

We follow the conventional notation and use $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ to denote the natural, integer, and real numbers. The symbols $\vee$, $\wedge$, $\neg$ stand for logical disjunction, conjunction, and negation, respectively. In the context of logic, $\oplus$ is used to denote 'exclusive or'. Sometimes, it is useful to distinguish between a definition and an equality. In these cases, a definition will be denoted using $\equiv$.

Linear Algebra

Vectors are denoted in lowercase bold, usually $\boldsymbol{x}, \boldsymbol{y}, \ldots$. Vectors are column vectors unless noted otherwise. The symbols $\mathbf{1}$ and $\mathbf{0}$ denote vectors of which the coefficients are all 1 or all 0, respectively. Matrices are denoted in uppercase bold, usually $\boldsymbol{A}, \boldsymbol{B}, \ldots$. The identity matrix is denoted by $\mathbb{I}$. Transposition is indicated with superscript T and matrix multiplication is implied by juxtaposition, for example, $\boldsymbol{x}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{x}$. The standard Euclidean norm of a vector is denoted by $\|\boldsymbol{x}\|$, and if another L_k norm is used, this is indicated with a subscript. For example, $\|\boldsymbol{x}\|_1$ denotes the L_1 norm of $\boldsymbol{x}$. In the context of linear algebra, $\otimes$ and $\oplus$ denote the direct (tensor) product and the direct sum, respectively.

Probability and Statistics

Random variables are denoted in capitals, usually $X, Y, \ldots$. Probabilities and probability densities are both denoted by $P(X)$. Expected value, variance, and covariance are notated as $\mathrm{E}(X)$, $\mathrm{V}(X)$, and $\mathrm{cov}(X, Y)$, respectively. Estimates are accented with a hat, for example, $\hat{E}(X)$ denotes the estimated expected value for X.

Code and Variables

R code is presented in fixed width font sections. Output is prefixed with two hashes.

```
age <- sample(100,25,replace=TRUE)
mean(age)
## [1] 52.44
```

Sometimes, it is useful to distinguish between the variable in the code and the logical concept. In such cases, the code variable will be denoted `age`, and the concept will be denoted *age*.

2

A Brief Introduction to R

The following sections provide an overview of some of R's core features. Besides an installation of R, we recommend installing one of the available integrated development environments (IDEs) for R. A good IDE does not only offer a nice interface to R and its help system but also helps you to organize projects, code, and data.

To benefit the most of this tutorial, it is a good idea to try out the code examples for yourself, play around with them, and to explain the results.

2.1 R on the Command Line

After starting R, or an IDE that connects to R, you have access to an interactive console, or command-line interface. The first use of it is to replace a pocket calculator. You can type in a calculation, and R will return the answer (preceded by a `[1]`).

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

To get started, experiment with the following statements. Make sure to play around a little. All common mathematical functions are implemented in R.

```
1 + 1
3^2
sin(pi/2)
(1 + 4) * 3
exp(1)
sqrt(16)
```

To reuse results or values, you can store them with the `<-` operator.

```
x <- 10
y <- 20
```

R has now remembered the values 10 and 20 and named them `x` and `y`. In fact, `x` and `y` are now officially *R objects*. R is very flexible, and there are several other ways to define an R object. We may replace `<-` with `=`, we may replace a statement `x <- 10` with `10 -> x`, or we can be extra verbose and use `assign("x",10)`. The `=` operator is the only one that is encountered with some frequency in practice. Since `=` is also used for named argument passing in function calls (see Section 2.6.1), we recommend using the `<-` for assignment.

Statistical Data Cleaning with Applications in R, First Edition. Mark van der Loo and Edwin de Jonge.

The content of an R object can be printed simply by typing its name in the console.

```
x
## [1] 10
```

R objects can be stored for further computation, the results of which may again be stored.

```
x + y
## [1] 30
z <- x * y
q <- x^2*z
q
## [1] 20000
```

Finally, we note that values and variables can be compared using standard comparison operators.

```
x <= y
## [1] TRUE
x == y
## [1] FALSE
x > y
## [1] FALSE
```

Observe that the operator testing for equality is written as the double equals symbol '==='. Make sure not to confuse this with the single equals symbol, which functions as assignment operator.

2.1.1 Getting Help and Learning R

R has a built-in help system where every possible function is described. If you know the name of the function, its help file can be requested with the `?` operator. For example, to show the help of the function `mean`, type the following:

```
?mean
```

If you are not sure of the function's name, the help files may be searched using the double question mark operator.

```
??average
```

IDEs for R have built-in search for the help files that may be more convenient.

There are a number of good online resources to get help from fellow users. Most notably, the Q&A site `stackoverflow.com` provides many R-related questions that have already been answered by users (and questions about many other topics as well). In fact, if you type an R-related question in a search engine, chances are that the first hit is a stackoverflow page. You may also want to subscribe to the R-help mailing list (see `https://www.r-project.org/mail.html`). Here, questions are often answered by the developers of the GNU R itself. Do observe the 'netiquette' and follow the posting guide before posting a question to the list. In particular, you should search the mailing list prior to posting a question to avoid double posts.

Besides resources where answers to questions can be found, there are many blogs discussing R and applications of R. A good way to become familiar with all the possibilities

of R is to frequently visit `r-bloggers.com`, where many R-related blogs are collected and presented in a newspaper-like format. Browsing through the blogs allows you to stumble upon functions and ideas that you cannot get from just following a tutorial.

Learning R is not something you should do alone. Besides the online community from which you can benefit, many cities have R user groups that organize frequent meetings that you can join. If your organization is using R, it is a good idea to organize a local user group within the organization. All you need is a room, a projector, and a laptop to start organizing meetings. In our experience, user meetings are a very efficient (and fun!) way to share knowledge and experiences among colleagues, friends, or classmates. The point is that even in base R, there are thousands of functions and many ways to solve the same problem. Informal user meetings are a good way of bumping into solutions you otherwise might not have thought of.

2.2 Vectors

The most basic type of object in R is called a *vector*, a sequence of values of the same type. The object is so basic that you have already worked with them. When in the previous examples we computed `x + y`, R was in fact adding two `numeric` vectors of length 1 containing the numbers `10` and `20`.

There are several ways to create a vector. One simple way is to use the function `c()` (for concatenate, or combine).

```
# a vector with numbers 1, 2, and 3
c(1,3,5)
# a vector with two text elements
c("hello world","hello universe")
```

Ordered number sequences can be generated with the colon operator (`:`) or with the `seq` function.

```
# a vector with numbers 1,2,...,10
1:10
# a sequence of numbers from 1 to 6 in 100 steps.
seq(1,6,length.out=100)
```

Sequences of random numbers from various distributions can be generated as well.

```
# 100 numbers drawn from the standard normal distribution
rnorm(100)
# 50 numbers drawn from the uniform distribution on [2,7]
runif(50,min=2,max=7)
```

You may try to combine values of a different type in a vector, but R will then convert the type when necessary.

```
c(1,"hello", 3.14)
## [1] "1"     "hello" "3.14"
```

When this vector is printed, there are quotes around the 'numbers' `"1"` and `"3.14"`. That is because R decided to convert these numbers to text since one of the elements in the vector is text (you can always convert a number to text but not the other way

around). By the way, in R such a conversion of type is usually referred to as *coercion*, which is just another word for the same thing.

This automatic conversion has consequences for everyday use. For example, the function `read.csv` reads `csv` files into R's working memory. It automatically detects the value types of the columns assuming that the first row contains the column names. Now if you feed it a `csv` file, where one of the columns contains all numeric data, except in one field, say somewhere at the bottom, that whole column will be interpreted as a categorical variable by default. Of course this behavior can be controlled, but it is typical of R to perform coercion rather than throwing an error.

There are a few basic vector types with which R can work, listed in the following table:

`logical`	Boolean values `TRUE` or `FALSE`
`integer`	Whole numbers, $\mathbb{Z}$
`numeric`	Real numbers, $\mathbb{R}$
`complex`	Complex numbers, $\mathbb{C}$
`character`	Text
`raw`	Binary data.

There are also types for storing categorical and ordered data.

`factor`	Categorical data, unordered
`ordered`	Ordinal data

These types are really integer vectors combined with a table that describes which category (level) is stored as what integer.

You can ask any object of what type it is, using the `class` function.

```
x <- 1:3
y <- c("foo", "bar")
class(x)
## [1] "integer"
class(y)
## [1] "character"
```

There are two more types of metadata stored with a vector. The first is its number of elements, which can be retrieved with the `length` function.

```
length(y)
## [1] 2
```

Secondly, the elements of a vector can be given names. For example:

```
shoesize <- c(jan=43, pier=39, joris=45, korneel=42)
```

The names are printed when a vector is printed to screen, but they do not affect any computations based on the vector.

```
mean(shoesize)
## [1] 42.25
```

The names of a vector can be retrieved with the `names` function.

```
names(shoesize)
## [1] "jan"     "pier"    "joris"   "korneel"
```

2.2.1 Computing with Vectors

All arithmetic and comparison operators and mathematical functions can be used on numerical vectors as you would on single numbers. The convention is that such operators and functions work element-wise on vectors.

```
x <- c(2,3,5,7)
y <- c(1,2,4,8)
x + y
## [1]  3  5  9 15
x < y
## [1] FALSE FALSE FALSE  TRUE
exp(-x) + sin(y)
## [1]  0.9768063  0.9590845 -0.7500645  0.9902701
```

The result of adding or comparing two vectors is again a vector, which may be stored and used in further computation.

It is possible to combine two vectors of different length. To compute the result, the shortest vector is repeated over the longer one.

```
3 + x
## [1]  5  6  8 10
z <- c(1,2)
z + y
## [1]  2  4  5 10
```

Here, R adds 3 to each element of `x`. In the second line, it adds 1 to the first element of `y` and 2 to the second element of `y`. It then notices that it got to the end of the vector `z`, so it starts back at the beginning adding 1 to the third element of `y` and 2 to the second element of `y`. The formal term for this is *recycling*; it is a behavior that is deeply embedded in R. A natural question is what happens when one tries to add two vectors where the shorted vector does not 'fit' a whole number of times on the longer vector. The reader is invited to test this by executing the following statement:

```
1:3 + 5:8
```

Besides vectorized operations where vectors are combined to new vectors of similar size, the content of vectors can be summarized in various ways.

```
x <- rnorm(100)
# compute the mean
mean(x)
## [1] 0.01797222
# compute the sample variance
var(x)
## [1] 1.355544
# standard deviation
sd(x)
## [1] 1.164278
# Tukey's five-number summary
fivenum(x)
## [1] -2.3084766 -0.7386481 -0.2128319  0.8240454  2.9226912
```

Especially useful is the function `summary`, which can be used to summarize just about any type of R object, including vectors.

```
summary(x)
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -2.30848 -0.73186 -0.21283  0.01797  0.81193  2.92269
```

It is also possible to visualize the data in vectors. Common plotting functionality includes the following:

```
x <- rnorm(100)
y <- x + rnorm(100)
# scatterplot
plot(x,y)
# boxplot
boxplot(x)
# histogram
hist(x)
```

2.2.2 Arrays and Matrices

An *array* is just a vector, endowed with a bit of metadata that states its dimensions. Arrays can be created with the `array` function.

```
A <- array(1:12, dim=c(2,3,2))
```

Here, we created a $2 \times 3 \times 2$ array containing the numbers 1–12. We advise you to execute the above statement and observe the order in which R has filled this multidimensional array. Many data structures can usefully be represented as (multidimensional) arrays, with high-dimensional tables being the obvious example. Since arrays are all but equal to vectors, we can perform computations with them just like we can with vectors.

```
2 * A
c(1,2) * A
```

A `matrix` is an array that has precisely two dimensions. The purpose of a matrix in R is to represent the objects familiar from linear algebra. Matrices can be created with the matrix function.

```
A <- matrix(1:6,ncol=2)
b <- matrix(c(-1,1),nrow=2)
```

Since matrices are vectors under the hood, the multiplication `A*A` is executed elementwise. This means that linear operations on matrices (addition and multiplication by a constant) work as expected out of the box, thanks to recycling. To perform matrix operations, some special operators and functions are available in R. Below is an overview of the most important operations.

`A %*% b`	Matrix multiplication $\boldsymbol{Ab}$
`t(A)`	Matrix transposition $\boldsymbol{A}^T$
`solve(t(A)%*%A,b)`	Solve the linear system $\boldsymbol{A}^T\boldsymbol{Ax} = \boldsymbol{b}$
`solve(t(A)%*%A)`	Compute the inverse $(\boldsymbol{A}^T\boldsymbol{A})^{-1}$
`svd(A)`	Singular value decomposition of $\boldsymbol{A}$.

Exercises for Section 2.2

Exercise 2.2.1 *On the command line, do the following:*

a) Compute $1^2 + 2^2 + \cdots + 50^2$.
b) The mean of the sequence $6, 7, \ldots, 75$
c) Can you generate the sequence $(1, 2, 4, 8)$ *in a single statement (not using* `c()`*)? Hint: think of recycling.*

Exercise 2.2.2 *If you create a vector with numbers in them, R by default stores them as* `numeric`*, or real values. You can force R to store integers by adding* `L` *after a number.*

```
x <- c(1L, 2L, 7L)
```

Now, execute the following code:

```
y <- 2 * x
```

Inspect the `class` *of* `x` *and* `y` *and explain what happened.*

2.3 Data Frames

A *data frame* is R's way to represent a rectangular data structure, where every row represents an observation, and every column represents a variable. An R-data frame is basically a sequence of vectors that may carry values of a different type, but they must all be of the same length.

R has a number of built-in datasets that can be used for examples and exercises.

```
data(iris)
head(iris,3)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

Here, we use the function `head` to print the first three lines of the dataset. The `iris` dataset contains sepal and petal length and width for three kinds of species of iris (see `?iris` for references). Like vectors, data frames can be summarized or plotted.

```
summary(iris)
plot(iris)
```

The `summary` command summarizes each column, and the plot command produces a matrix plot with scatter plots of each variable against one another. Other useful metadata can be retrieved with the following functions:

```
# the number of rows
nrow(iris)
# the number of columns
ncol(iris)
# both nr of rows and columns
```

```
dim(iris)
# names of the columns
names(iris)
# a shorter summary of a data.frame
str(iris)
```

The function `str` (short for structure) gives a technical overview of the contents of a `data.frame`, whereas `summary` gives a statistical summary.

Columns can be retrieved, added, or removed using the dollar operator.

```
# compute the mean sepal width
mean(iris$Sepal.Width)
## [1] 3.057333
# add a 'ratio' column
iris$ratio <- iris$Sepal.Width/iris$Sepal.Length
head(iris, 2)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species     ratio
## 1          5.1         3.5          1.4         0.2  setosa 0.6862745
## 2          4.9         3.0          1.4         0.2  setosa 0.6122449
# remove the 'ratio' column again
iris$ratio <- NULL
```

2.3.1 The Formula-Data Interface

Many functions in R support the so-called *formula-data interface*. This interface is aimed at specifying a relation between variables, separately from where the data can be found. A *formula* is an R-expression of the form

```
[dependent variable] ~ [independent variables]
```

where independent variables, or sometimes functions thereof, are stated on the right side of the tilde (~). Try the following examples to get a feel for this concept:

```
# specify a linear model
m <- lm(Sepal.Length ~ Sepal.Width + Species, data=iris)
summary(m)

# create a boxplot for each species
boxplot(Sepal.Length ~ Species, data = iris)

# create a scatterplot
plot(Sepal.Length ~ Sepal.Width, data = iris)
```

2.3.2 Selecting Rows and Columns; Boolean Operators

Being able to select subsets of data is a fundamental skill to data processing. In R, there are many ways to do so, including methods provided by packages. Here, we limit ourselves to methods provided by base R.

Perhaps, the most convenient command to select a subset of records using base R is the `subset` function. For example, to select a subset of records from the built-in `women` dataset whose height exceeds 70, we can do the following:

```
subset(women, height > 70)
##    height weight
```

```
## 14     71    159
## 15     72    164
```

The function accepts a `data.frame` and a logical statement that expresses the condition(s) for records in the subset. Note that we do not need to reference the `women` dataset in the logical statement. The `subset` function understands that `height` must be a variable stored in `women`. To build up logical statements, R supports the following basic Boolean operations and quantifiers:

`&`	Logical AND
`|`	Logical OR
`!`	Logical NOT
`xor()`	Logical EXCLUSIVE OR.
`all()`	Only true if every value in the input is `TRUE`.
`any()`	Only true if at least one value in the input is `TRUE`.

2.3.3 Selection with Indices

The second way to make selections from a dataset is to use vectors of indices. An index vector may be a `logical` vector of the same size as the object selected from or an `integer` or `numeric` vector stating the desired positions. To make a selection from an R object (vector, data frame), one uses the square bracket operators.

```
x <- c(1,7,10,13,19)
# select the 2nd element
x[2]
## [1] 7
# select the 2nd and 5th element
x[c(2,5)]
## [1]  7 19
# set the first element to 0
x[1] <- 0
# select all elements equal to zero
x[x==0]
## [1] 0
# set all elements equal to zero equal to 1
x[x==0] <- 1
```

To understand the last two statements, recall that a logical expression such as `x==0` returns a `logical` vector, which may then be used as an index. Mastering indices is one of the most important R skills to acquire since it allows you to very flexibly and quickly find and alter data.

It is possible to separate the computation of an index from its application to data by storing a computed index prior to usage.

```
# find x < 15; I is a logical vector
I <- x < 15
# find x > 1; J is a logical vector
J <- x > 2
# select elements in x satisfying both I and J
x[I&J]
## [1]  7 10 13
```

There are a number of functions that can help you find specific values in a vector.

`min`, `max`	The smallest or largest value
`which.min`, `which.max`	The position of the smallest or largest value
`which`	Position of `TRUE`s in a `logical` vector

Here are some examples.

```
max(x)
## [1] 19
which.max(x)
## [1] 5
which(x == 7)
## [1] 2
```

Since data frames have two dimensions, you need two indices to select from them, one for the rows and one for the columns. For example, to select rows 3–7 and columns 2–4 from the `iris` dataset, do the following:

```
iris[3:7,2:4]
##   Sepal.Width Petal.Length Petal.Width
## 3         3.2          1.3         0.2
## 4         3.1          1.5         0.2
## 5         3.6          1.4         0.2
## 6         3.9          1.7         0.4
## 7         3.4          1.4         0.3
```

Indices before the comma select rows, and indices after the comma select columns. Leaving out an index means 'make no selection', that is, everything is returned. Here, we select all columns for the first row.

```
iris[1, ]
```

Similarly, we can select all rows for columns 2–4.

```
iris[ ,2:4]
```

There is one caveat when selecting columns in the above procedure. If only a single column is selected, for example, `iris[, 1]`, R will return a vector, rather than a single-column data frame. There are two ways to prevent this behavior. The first is by providing the extra argument `drop=FALSE` (meaning, dimensions will not be dropped).

```
iris[,1,drop=FALSE]
```

The second way is to not provide a comma and use only a single index when selecting columns.

```
iris[1]
```

Finally, we note that it is also possible to select columns with (vectors of) column names.

```
iris[ iris$Sepal.Length < 6, 'Species', drop=FALSE]
```

2.3.4 Data Frame Manipulation: The `dplyr` Package

The `dplyr` package of Wickham and Francois (2014) offers a set of functions that facilitate a very consistent way of working with data frames. The package will be discussed

briefly in Section 4.2.3, but it also comes with an excellent tutorial, which can be found at the package's CRAN page.

Exercises for Section 2.3

Exercise 2.3.3 *The built-in* `women` *dataset contains two columns of data representing the average height (inches) and weight (pounds) of American women aged 30–39 (for some year prior to 1975).*

a) Add a column called `heightM` *representing* `height` *in meters. One inch equals 2.54 cm.*
b) Add a column called `weightKg` *representing* `weight` *in kilograms. One kilogram equals 2.2046 lb.*
c) The Quetelet index QI, (also body-mass index) is computed as

$$QI = \frac{\textit{weight in kilogram}}{(\textit{height in meter})^2}.$$

Add a column called `QI` *with the Quetelet index for each row in the* `women` *dataset.*
d) Compute the mean and median Quetelet index for this dataset.
e) How many indices are above 23? (Hint: you can count the number of `TRUE`*s in a vector by using computing the* `sum` *over them).*

Exercise 2.3.4 *Use (computed) indices to answer the following questions:*

a) How many rows in the `iris` *dataset have* `Petal.Length` *larger than 5 and* `Sepal.Width` *smaller than 3. Hint you can* `sum` *over* `logical` *vectors to count the number of* `TRUE`*s.*
b) There are two specimens of `iris` *that have* `Sepal.Length` *equal to 5.7 and* `Sepal.Width` *equal to 2.8. Of what species are they? Use a single R statement to select the species from the dataset.*
c) Of what species is the iris with the largest `Sepal.Length`*?*

Exercise 2.3.5 *The* winsorized mean *is a robust way to estimate the mean that works by replacing every value above (or below) a certain threshold with the threshold. Compute the winsorized mean of the column* `Sepal.Length`*, with threshold 7. Do this by first replacing every value* > 7 *with 7 and then computing the mean (the correct answer is 5.80533).*

2.4 Special Values

Similar to most programming languages, R has provisions to represent exceptional values such as ±`Inf`, `NaN` (not a number), `NA` (missing value), and `NULL`.

The value `Inf` represents the result of a numerical calculation such as `1/0`, which, when written as $\lim_{x\to 0} 1/x$, would yield ∞. Not a number, or `NaN`, results from computations such as `0/0` or `Inf-Inf`, where even taking a limit is undefined. We will be

more precise about these concepts in Section 3.1.3. For now, we only mention that conceptually, one can think of the number system with which R computes is the augmented real line

$$\mathbb{R} \cup \{\pm\infty\} \cup \texttt{NaN} \cup \texttt{NA}, \tag{2.1}$$

where NA represents a missing value. All of R's arithmetic operations and mathematical functions are closed on this set. That is, if you feed any operation element(s) from (2.1), the result will be in the same set.

```
6/0
## [1] Inf
6/0  - Inf
## [1] NaN
```

In general, you can a reason about `Inf` as if it is an ordinary number, with a few special rules you would expect, such as `Inf + x == Inf` for any finite number `x`. This also means that it can be detected using standard boolean operators.

```
x <- c(1,2,Inf)
x == Inf
## [1] FALSE FALSE  TRUE
```

The NaN value is conceptually a bit different. Informally, it can be understood as an 'undefined' value. Since we cannot be definite about the outcome when comparing an undefined value with another value, defined or undefined, any comparison with NaN will result in NA.

```
x <- c(1,2,NaN)
x == NaN
## [1] NA NA NA
```

In fact, any calculation involving NaN will yield something undefined; either a missing value or NaN but never a number. The way to detect NaNs is with a special detector function called `is.nan`.

```
is.nan(x)
## [1] FALSE FALSE  TRUE
```

For numeric data, there is a convenient function `is.finite` that establishes whether a value from the set of Eq. (2.1) represents a finite number.

```
is.finite(c(2,1,3,NA,7,Inf,NaN))
## [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
```

The value NULL can be thought of as the empty set. Unlike all the other special values, NULL has no type (its `class` is NULL) and `length` zero.

```
length( c(1,NULL,2))
## [1] 2
length( c(1,NA,2))
## [1] 3
```

Since NULL has no length and is not a vector (try `is.vector(NULL)`), comparing a vector with NULL needs to be defined separately. In R, such comparisons yield a logical vector of length zero.