

CLAUS THORN EKSTRØM

The Primer



CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

The R Primer

This page intentionally left blank

The R Primer

CLAUS THORN EKSTRØM



CRC Press

Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business

A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2012 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20110713

International Standard Book Number-13: 978-1-4398-6208-7 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

Contents

Preface	xi
1 Importing data	1
1.1 Read data from a text file	1
1.2 Read data from a simple XML file	3
1.3 Read data from an XML file	4
1.4 Read data from an SQL database using ODBC	8
Reading spreadsheets	9
1.5 Read data from a CSV file	9
1.6 Read data from an Excel spreadsheet	10
1.7 Read data from an Excel spreadsheet under Windows . .	12
1.8 Read data from a LibreOffice or OpenOffice Calc spread- sheet	13
1.9 Read data from the clipboard	14
Importing data from other statistical software programs . .	15
1.10 Import a SAS dataset	15
1.11 Import an SPSS dataset	16
1.12 Import a Stata dataset	16
1.13 Import a Systat dataset	17
Exporting data	17
1.14 Export data to a text file	17
1.15 Export a data frame to a CSV file	18
1.16 Export a data frame to a spreadsheet	19
1.17 Export a data frame to an Excel spreadsheet under Win- dows	20
1.18 Export a data frame to a SAS dataset	20
1.19 Export a data frame to an SPSS dataset	21
1.20 Export a data frame to a Stata dataset	22
1.21 Export a data frame to XML	22
2 Manipulating data	25
2.1 Using mathematical functions and operations	27
2.2 Working with common functions	29
2.3 Working with dates	31

2.4	Working with character vectors	33
2.5	Find the value of x corresponding to the maximum or minimum of y	34
2.6	Check if elements in one object are present in another object	35
2.7	Transpose a matrix (or data frame)	35
2.8	Impute values using last observation carried forward	37
2.9	Convert comma as decimal mark to period	39
	Working with data frames	40
2.10	Select a subset of a dataset	40
2.11	Select the complete cases of a dataset	41
2.12	Delete a variable from a data frame	42
2.13	Join datasets	43
2.14	Merge datasets	44
2.15	Stack the columns of a data frame together	45
2.16	Reshape a data frame from wide to long format or vice versa	47
2.17	Create a table of counts	50
2.18	Convert a table of counts to a data frame	52
2.19	Convert a data frame to a vector	53
	Factors	54
2.20	Convert a factor to numeric	54
2.21	Add a new level to an existing factor	55
2.22	Combine the levels of a factor	56
2.23	Remove unused levels of a factor	56
2.24	Cut a numeric vector into a factor	57
	Transforming variables	58
2.25	Sort data	58
2.26	Transform a variable	59
2.27	Apply a function multiple times to parts of a data frame or array	60
2.28	Use a Box-Cox transformation to make non-normally distributed data approximately normal	62
2.29	Calculate the area under a curve	64
3	Statistical analyses	67
	Descriptive statistics	70
3.1	Create descriptive tables	70
	Linear models	72
3.2	Fit a linear regression model	72
3.3	Fit a multiple linear regression model	74
3.4	Fit a polynomial regression model	75
3.5	Fit a one-way analysis of variance	76

3.6	Fit a two-way analysis of variance	79
3.7	Fit a linear normal model	82
	Generalized linear models	85
3.8	Fit a logistic regression model	85
3.9	Fit a multinomial logistic regression model	89
3.10	Fit a Poisson regression model	92
3.11	Fit an ordinal logistic regression model	96
	Methods for analysis of repeated measurements	100
3.12	Fit a linear mixed-effects model	100
3.13	Fit a linear mixed-effects model with serial correlation	104
3.14	Fit a generalized linear mixed model	110
3.15	Fit a generalized estimating equation model	113
3.16	Decompose a time series into a trend, seasonal, and residual components	117
3.17	Analyze time series using an ARMA model	120
	Specific methods	123
3.18	Compare populations using t test	123
3.19	Fit a nonlinear model	125
3.20	Fit a Tobit regression model	129
	Model validation	131
3.21	Test for normality of a single sample	131
3.22	Test for variance homogeneity across groups	132
3.23	Validate a linear or generalized linear model	134
	Contingency tables	137
3.24	Analysis of two-dimensional contingency tables	137
3.25	Analyze contingency tables using log-linear models	139
	Agreement	142
3.26	Create a Bland-Altman plot of agreement to compare two quantitative methods	142
3.27	Determine agreement among several methods of a quantitative measurement	144
3.28	Calculate Cohen's kappa	148
	Multivariate methods	150
3.29	Fit a multivariate regression model	150
3.30	Cluster observations	152
3.31	Use principal component analysis to reduce data dimensionality	155
3.32	Fit a principal component regression model	158
3.33	Classify observations using linear discriminant analysis	160
3.34	Use partial least squares regression for prediction	163
	Resampling statistics and bootstrapping	166
3.35	Non-parametric bootstrap analysis	166

3.36	Use cross-validation to estimate the performance of a model or algorithm	169
3.37	Calculate power or sample size for simple designs	172
	Robust statistics	176
3.38	Correct p -values for multiple testing	176
	Non-parametric methods	178
3.39	Use Wilcoxon's signed rank test to test a sample median	178
3.40	Use Mann-Whitney's test to compare two groups	180
3.41	Compare groups using Kruskal-Wallis' test	181
3.42	Compare groups using Friedman's test for a two-way block design	183
	Survival analysis	185
3.43	Fit a Kaplan-Meier survival curve to event history data	185
3.44	Fit a Cox regression model (proportional hazards model)	188
3.45	Fit a Cox regression model (proportional hazards model) with time-varying covariates	192
4	Graphics	197
4.1	Including Greek letters and equations in graphs	199
4.2	Set colors in R graphics	201
4.3	Set color palettes in R graphics	202
	High-level plots	204
4.4	Create a scatter plot	204
4.5	Create a histogram	206
4.6	Make a boxplot	208
4.7	Create a bar plot	209
4.8	Create a bar plot with error bars	211
4.9	Create a plot with estimates and confidence intervals	213
4.10	Create a pyramid plot	214
4.11	Plot multiple series	217
4.12	Make a 2D surface plot	218
4.13	Make a 3D surface plot	220
4.14	Plot a 3D scatter plot	222
4.15	Create a heat map plot	224
4.16	Plot a correlation matrix	226
4.17	Make a quantile-quantile plot	227
4.18	Graphical model validation for linear models	229
	More advanced graphics	233
4.19	Create a broken axis to indicate discontinuity	233
4.20	Create a plot with two y -axes	234
4.21	Rotate axis labels	236
4.22	Multiple plots	237
4.23	Add a legend to a plot	239

4.24	Add a table to a plot	240
4.25	Label points in a scatter plot	241
4.26	Identify points in a scatter plot	243
4.27	Visualize points, shapes, and surfaces in 3D and interact with them in real-time	244
	Working with graphics	247
4.28	Exporting graphics	247
4.29	Produce graphics output in L ^A T _E X-ready format	248
4.30	Embed fonts in postscript or pdf graphics	250
5	R	253
	Getting information	253
5.1	Getting help	253
5.2	Finding R source code for a function	255
	R packages	257
5.3	Installing R packages	257
5.4	Update installed R packages	259
5.5	List the installed packages	260
5.6	List the content of a package	260
5.7	List or view vignettes	262
5.8	Install a package from BioConductor	263
5.9	Permanently change the default directory where R in- stalls packages	264
5.10	Automatically load a package when R starts	265
	The R workspace	266
5.11	Managing the workspace	266
5.12	Changing the current working directory	267
5.13	Saving and loading workspaces	268
5.14	Saving and loading histories	268
5.15	Interact with the file system	270
5.16	Locate and choose files interactively	271
5.17	Interact with the operating system	272
	Bibliography	275
	Index	277

This page intentionally left blank

Preface

This book is not about statistical theory, neither is it meant to teach R programming. This book is intended for readers who know the basics of R, but find themselves with problems or situations that are commonly encountered by newcomers to R or for readers who want to see compact examples of different types of typical statistical analyses. In other words, if you understand basic statistics and already know a bit about R then this book is for you.

R has rapidly become the *lingua franca* of statistical computing; it is a free statistical programming software and it can be downloaded from <http://cran.r-project.org>. Many newcomers to R are often intimidated by the command-line interface, or the sheer number of functions and packages, or just trying to figure out how to import data and perform a simple statistical analysis.

The book consists of a number of examples that illustrate a specific situation, topic or problem from data import over data management and classical statistical analyses to graphics. Each example is self-contained and provides R code that can be run exactly as shown and the results from the book can be reproduced. The only change — barring simulated data, machine set-up and small tweaks to make figures suitable for printing — is that some of the output lines have been removed for brevity.

This is not a “missing manual” or a thorough exploration of the functions used. Instead of trying to cover every possible option or special case that might be of interest, we focus on the common situations that most beginning users are likely to encounter. Thus we concentrate on the basics of getting things done and giving examples that can be used as a starting point for the reader rather than exploring the multitude of options available with every command and the ever-increasing number of packages. For most problems — and this is particularly true for a programming language like R — there is more than one way to solve a problem. Here, I have provided a single solution to most problems and have tried to use base R if at all possible. If there are other functions and/or packages available that cover or extend the same functionality, then some of them are listed at the end of each example.

The R list of frequently asked questions is highly recommended and covers a few of the same topics mentioned here. However, it does not cover examples of statistical analyses and it rarely covers some of the most basic problems new users encounter.

Base graphics are used throughout the book. More advanced graphics can be produced with the recent `lattice` and `ggplot2` packages (see Sarkar (2008), Wickham (2009), or Murrell (2011) for further information on advanced R graphics). A more complete coverage of R and/or statistics can be found in the books by Venables and Ripley (2002), Verzani (2005), Crawley (2007), Dalgaard (2008), and Everitt and Hothorn (2010). These books have a slightly different target audience than the present text and are all highly recommended.

The R Primer has a supporting web site at

`http://www.statistics.life.ku.dk/primer/`

where additional topics are covered and where the R code used in the book can be found.

I would like to thank all R developers and package writers for the enormous work they have done and continue to put into the R program and extensions. I appreciate all the helpful responses to my enquiries and suggestions. I am grateful to my colleagues at the Faculty of Life Sciences, University of Copenhagen as well as Klaus K. Holst, Duncan Temple Lang, and Bendix Carstensen for their ideas, comments, suggestions and encouragement on various stages of the manuscript. Many thanks to Tina Ekstrøm for once again creating a wonderful cover, and last, but not least, thanks to Marlene, Ellen and Anna for bearing with me through yet another book.

Claus Thorn Ekstrøm
Frederiksberg 2011

Chapter 1

Importing data

1.1 Read data from a text file

Problem: You want to import a dataset stored in an ASCII text file.

Solution: Data stored in simple text files can be read into R using the `read.table` function. By default, the observations should be listed in columns where the individual fields are separated by one or more white space characters, and where each line in the file corresponds to one row of the data frame. The columns do not need to be straight or formatted, but multi-word observations like `high income` need to be put in quotes or combined into a single word. Assume we have a text file, `mydata.txt`, with the following contents

acid	digest	name
30.3	70.6	NA
29.8	67.5	Eeny
NA	87.0	Meeny
4.1	89.9	Miny
4.4	.	Moe
2.8	93.1	.
3.8	96.7	" "

which we read the data into R with the following command:

```
> indata <- read.table("mydata.txt", header=TRUE)
> indata
  acid digest name
1 30.3   70.6 <NA>
2 29.8   67.5 Eeny
3  NA   87.0 Meeny
4  4.1   89.9 Miny
5  4.4     .  Moe
6  2.8   93.1 .
7  3.8   96.7 " "
```

The first argument is the name of the data file, and the second argument (`header=TRUE`) is optional and should be used only if the first

line of the text file provides the variable names. If the first line does not contain the column names, the variables will be labeled consecutively `V1`, `V2`, `V3`, etc. Each line in the input file must contain the same number of columns for `read.table` to work. The `sep` option should be included to indicate which character separates the columns if the columns are separated by other characters than spaces. For example, if the columns are separated by tabs then we can use `sep="\t"`. Data read with `read.table` are stored as a data frame within R.

The default code for missing observations is the character string `NA` which we can see works both for the first and third observation above (`acid` is read as a numeric vector and `name` as a factor). Empty character fields are scanned as empty character vectors, unless the option `na.strings` contains the value `" "` in which case they become missing values. Empty numeric fields (for example if the columns are separated by tabs) are automatically considered missing.

```
> indata <- read.table("mydata.txt", header=TRUE,
+                       na.strings=c("NA", " "))
> indata
  acid digest  name
1 30.3   70.6 <NA>
2 29.8   67.5 Eeny
3  NA   87.0 Meeny
4  4.1   89.9 Miny
5  4.4     .  Moe
6  2.8   93.1  .
7  3.8   96.7 <NA>
```

Note that due to the period `'.'` for observation 5, R considers the variable `digest` as a factor and not numeric since the period is read as a character string. If periods should also be considered missing variables we need to include that in `na.strings`.

```
> indata <- read.table("mydata.txt", header=TRUE,
+                       na.strings=c("NA", " ", "."))
> indata
  acid digest  name
1 30.3   70.6 <NA>
2 29.8   67.5 Eeny
3  NA   87.0 Meeny
4  4.1   89.9 Miny
5  4.4    NA  Moe
6  2.8   93.1 <NA>
7  3.8   96.7 <NA>
```

R looks for the file `mydata.txt` in the current working directory, but the full path can be specified in the call to `read.table`, e.g.,

```
> indata <- read.table("d:/mydata.txt", header=TRUE)
```

See Rule 5.12 on how to change the current working directory.

1.2 Read data from a simple XML file

Problem: You want to import a dataset stored as a simple structure in the XML file format.

Solution: The XML (eXtensible Markup Language) was designed to transport and store data and XML has seen widespread use in inter-changing data over the Internet.

An XML file consists of a series of elements which form a document tree. The tree starts at the root and branches to the lowest level of the tree. XML documents must contain a root node (or element) which is “the parent” of all other nodes, and all nodes can have their own sub-nodes (“child elements”).

An example XML file is shown below where the tree data from the `trees` dataset are stored in XML format. The root node `<document>` has several child nodes (the `<rows>`) and each row has its own child elements corresponding to the variables in the data frame and their values.

```
<?xml version="1.0"?>
<document>
  <row>
    <Girth>8.3</Girth>
    <Height>70</Height>
    <Volume>10.3</Volume>
  </row>
  <row>
    <Girth>8.6</Girth>
    <Height>65</Height>
    <Volume>10.3</Volume>
    .
    .
    .
    <Volume>77</Volume>
  </row>
</document>
```

The XML package provides numerous tools for parsing and generating XML in R. Since XML is such a flexible format, the XML package primarily consists of functions that must be combined to parse and extract information from a specific type of XML structure.

XML document files with a simple structure can be imported and converted to a data frame directly using the `xmlToDataFrame` function. By simple, we mean a collection of nodes that have the same sub-nodes such that each node corresponds to an observation or row in the data frame and each of its sub-nodes contains primitive values correspond-

ing to the variables. The data file shown above has such a simple structure.

```
> library(XML)
> url <- "http://www.statistics.life.ku.dk/primer/mydata.xml"
> indata <- xmlToDataFrame(url)
> head(indata)
   Girth Height Volume
1    8.3     70   10.3
2    8.6     65   10.3
3    8.8     63   10.2
4   10.5     72   16.4
5   10.7     81   18.8
6   10.8     83   19.7
```

Note: Installing the XML package can be a little trickier than other packages. The package uses libxml, the XML parser that is frequently found as part of the Gnome system, but also exists as a stand-alone library for many systems.

See also: Use Rule 1.3 to import XML files that do not have a simple structure.

1.3 Read data from an XML file

Problem: You want to import a dataset stored in the XML file format by manually coding how to extract the relevant information.

Solution: Rule 1.2 showed how to import data from an XML (eXtensible Markup Language) file with a simple structure. Here we will try to import data from a more non-trivial situation, which `xmlToDataFrame` cannot handle.

As a more complex example we will try to import the following XML file that contains artificial data on currency exchange rates. The first couple of nodes are document creation data, while the actual exchange rates begin with the `<rates>` node. The exchange rates (measured against the euro) and dates are coded as tags to the `<exch>` and `<Date>` nodes while the bank source is a leaf node with no children. Also note, that information from both banks are not available for both dates and that not all exchange rates nodes may be present.

```
<?xml version="1.0"?>
<bankdata>
<author>Claus</author>
<valid>Not at all</valid>
```

```

<rates>
<Date time="2011-03-10">
  <bank>
    <source>Some bank</source>
    <exch currency="USD" rate="1.3817"/>
    <exch currency="DKK" rate="7.4581"/>
  </bank>
  <bank>
    <source>Some other bank</source>
    <exch currency="USD" rate="1.2382"/>
    <exch currency="DKK" rate="7.3312"/>
  </bank>
</Date>
<Date time="2011-03-09">
  <bank>
    <source>Some bank</source>
    <exch currency="USD" rate="1.3884"/>
  </bank>
</Date>
</rates>
</bankdata>

```

Recall that an XML tree structure consists of a series of nodes branching out from the root node, and that each of the nodes may itself have children. Data are stored either as values or as attributes/tags of a node.

The `xmlTreeParse` is the work-horse for importing general XML documents. `xmlTreeParse` parses an XML file and stores the tree in an R structure. We subsequently traverse the tree and extract data from the relevant nodes. `xmlTreeParse` requires a file name or location as input for where to find the XML file, and it returns an R XML object with the parsed XML file. The `useInternalNodes` option can be set to `TRUE` to increase parsing speed.

First, `xmlRoot` should be called to get a pointer to the top-level node or parent of the XML tree. The `skip` option can be set to `FALSE` to prevent R from skipping over document type definitions in the XML file if those are present.

The XML tree structure works like a recursive list-like object and the individual nodes in the tree are accessed using named or numbered indices, `[[]]`. The XML tree can be traversed with the proper indices and for each node we can get the parent and list of children sub-nodes using the `xmlParent` and `xmlChildren` functions, respectively.

Information can be extracted from a node using one of the `xmlName`, `xmlValue`, `xmlGetAttr` and `xmlAttrs` functions, which return the node name, node contents, a named attribute and all attributes, respectively.

```

> library(XML)
> # Location of the example XML file

```

```

> url <- "http://www.statistics.life.ku.dk/primer/bank.xml"
> # Parse the tree
> doc <- xmlTreeParse(url, useInternalNodes=TRUE)
> top <- xmlRoot(doc)           # Identify the root node
> xmlName(top)                  # Show node name of root node
[1] "bankdata"
> names(top)                    # Name of root node children
  author    valid    rates
"author"  "valid"  "rates"
> xmlValue(top[[1]])            # Access first element
[1] "Claus"
> xmlValue(top[["author"]])     # First element with named index
[1] "Claus"
> names(top[[3]])               # Children of node 3
  Date    Date
"Date"  "Date"
> xmlAttrs(top[[3]][[1]])       # Extract tags from a Date node
  time
"2011-03-10"
> top[["rates"]][[1]][[1]]     # Tree from first bank and date
<bank>
  <source>Some bank</source>
  <exch currency="USD" rate="1.3817"/>
  <exch currency="DKK" rate="7.4581"/>
</bank>
> xmlValue(top[[3]][[1]][[1]][[1]]) # Bank name is node value
[1] "Some bank"
> xmlAttrs(top[[3]][[1]][[1]][[1]]) # but has no tags
NULL
> xmlAttrs(top[[3]][[1]][[1]][[2]]) # The <exch> node has tags
currency    rate
  "USD"    "1.3817"
> xmlValue(top[[3]][[1]][[1]][[2]]) # but no value
[1] ""

```

A function can be applied recursively to children of a node using the `xmlApply` and `xmlSApply` functions, which works similarly as `apply` and `sapply` except for XML tree structures. Extracting the individual exchange rates and combining them with the proper bank name and date can be quite cumbersome using indices, loops and `xmlApply`. Instead, we can use the XML Path Language, XPath, to query and extract information from specific nodes in the XML tree structure. Table 1.1 shows examples of useful XPath query strings. These can be used with the `xpathApply` or `xpathSApply` functions, which accept a node from where to start the search as first argument, an XPath query string as second argument, and the function to apply as third argument.

```

> # Search tree for all source nodes and return their value
> xpathSApply(doc, "//source", xmlValue)
[1] "Some bank"      "Some other bank" "Some bank"

> # Search full tree for all exch nodes where currency is "DKK"

```

Table 1.1: Examples of XPath search expression

Expression	Description
/node	top-level node only
//node	node at any level
//node[@name]	node with an attribute named "name"
//node[@name="a"]	node with named attr. with value "a"
//node/@x	value of attribute x in node with such attr.

```
> xpathApply(doc, "//exch[@currency='DKK']", xmlAttrs)
[[1]]
currency    rate
  "DKK"    "7.4581"

[[2]]
currency    rate
  "DKK"    "7.3312"
```

Below we search through the complete XML tree for each node, <exch>, which may be found at any level. From the <exch> node we extract its attributes and get the bank name through its parent and the exchange rate date from the time attribute from its grandparent. The `do.call` function is used `rbind` to combine the resulting lists.

```
> res <- xpathApply(doc, "//exch",
+   function(ex) {
+     c(xmlAttrs(ex),
+       bank=xmlValue(xmlParent(ex)[["source"]]),
+       date=xmlGetAttr(xmlParent(xmlParent(ex)), "time"))
+   })
> result <- do.call(rbind, res)
> result
  currency rate      bank      date
[1,] "USD"   "1.3817" "Some bank" "2011-03-10"
[2,] "DKK"   "7.4581" "Some bank" "2011-03-10"
[3,] "USD"   "1.2382" "Some other bank" "2011-03-10"
[4,] "DKK"   "7.3312" "Some other bank" "2011-03-10"
[5,] "USD"   "1.3884" "Some bank"   "2011-03-09"
```

The variables in the resulting object can then be converted to their proper formats and combined in a data frame.

1.4 Read data from an SQL database using ODBC

Problem: Import data from an application that supports Open DataBase Connectivity.

Solution: Open DataBase Connectivity (ODBC) makes it possible for any application to access data from a SQL database regardless of which database management system is used to handle the data.

The `RODBC` package provides an interface to databases that support an ODBC interface, which includes most popular commercial and free databases such as MySQL, PostgreSQL, Microsoft SQL Server, Microsoft Access, and Oracle. Having one package with a common interface allows the same R code to access different database systems.

The `odbcConnect` function opens a connection to a database and returns an object which works as a handle for the connection. A character string containing the data source name (DSN) should be supplied as the first argument to `odbcConnect` to set the database server to connect to. The function has two optional arguments, `uid` and `pwd`, which set the user id and password for authentication, respectively, if that is required by the database server, and is not provided by the DSN.

The data source name is located in a separate text file or in the registry and it contains the information that the ODBC driver needs in order to connect to a specific database. This includes the name, directory, and driver of the database, and possibly the user id and password. Each database requires a separate entry in the DSN, and DSN-less connections require that all the necessary information to be supplied within R (for example by using `odbcDriverConnect` instead of `odbcConnect`). The information for setting up the DSN should accompany your database software.

Once a connection to a database server is established, then the available database tables can be seen with the `sqlTables` function, with the proper handle as first argument. The `sqlFetch` function fetches the entire table from the SQL database and returns it as an R data frame. `sqlFetch` requires two arguments, where the first is the connection handle and the second is a character string containing the desired table to extract from the database.

The workhorse is the `sqlQuery` function which is used to make SQL queries directly to the database and return the results as R data frames. The first argument to `sqlQuery` sets the connection channel to use and the second argument is the selection string which should be specified as a regular SQL query string. Finally, the `odbcClose` function closes the connection to the channel specified by the first argument.

In the example below, we use an existing DSN to access a database called “myproject” which contains a “paper” table. The entire table is extracted as well as a selection of salespeople with sales larger than a given number.

```
> library(RODBC)
> # Connect to SQL database with username and password
> channel <- odbcConnect("mydata", uid="tv", pwd="office")
> sqlTables(channel) # List tables in the database
  TABLE_CAT TABLE_SCHEM TABLE_NAME TABLE_TYPE REMARKS
1 myproject          paper          TABLE
> mydata <- sqlFetch(channel, "paper") # Fetch entire table
> mydata
  ID Sales      Person
1  3    10 David Brent
2  4    12 Michael Scott
3  5    13 Gareth Keenan
4  6    20 Tim Canterbury
5  7    13   Jim Halpert
6  8    23 Dwight Schrute
> sqlQuery(channel,
+ "SELECT * FROM paper WHERE Sales>12 ORDER BY Person")
  ID Sales      Person
1  8    23 Dwight Schrute
2  5    13 Gareth Keenan
3  7    13   Jim Halpert
4  6    20 Tim Canterbury
> odbcClose(channel)
```

See also: Rule 1.7 for an example on how to use the RODBC package to import data from Excel under Windows. Microsoft Access Databases can be accessed directly from within Windows without creating a DSN using the `odbcConnectAccess` or `odbcConnectAccess2007` functions.

Reading spreadsheets

1.5 Read data from a CSV file

Problem: You want to import a dataset stored as a comma-separated values (CSV) file.

Solution: A CSV file is a plain text file where each line corresponds

to a case and where the case variables are separated by commas. Quotation marks are used to embed field values that contain the separator character.

Use `read.csv` to read in the delimited file just like you would use `read.table`. Actually, `read.csv` is a wrapper function that sets the correct options for `read.table`.

`read.csv2` is used to read in semicolon separated files which is the default CSV-format in some locales where comma is the decimal point character and where semicolon is used as separator. To read the CSV file `mydata.csv`:

```
"Id", "Sex", "Age", "Score"
21, "Male", 14, "Little"
26, "Male", 13, "None"
27, "Male", 13, "Moderate, severe"
29, "Female", 13, "Little"
30, "Female", 15, "Little"
31, "Male", 14, "Moderate, severe"
```

we use the command

```
> indata <- read.csv("mydata.csv")
> head(indata)
  Id  Sex Age      Score
1 21  Male 14    Little
2 26  Male 13      None
3 27  Male 13 Moderate, severe
4 29 Female 13    Little
5 30 Female 15    Little
6 31  Male 14 Moderate, severe
```

Both `read.csv` and `read.csv2` assume that a header line is present in the CSV file (i.e., `header=TRUE` is the default). If no header line is present you need to specify the `header=FALSE` argument.

1.6 Read data from an Excel spreadsheet

Problem: You want to import a dataset stored as a Microsoft Excel file.

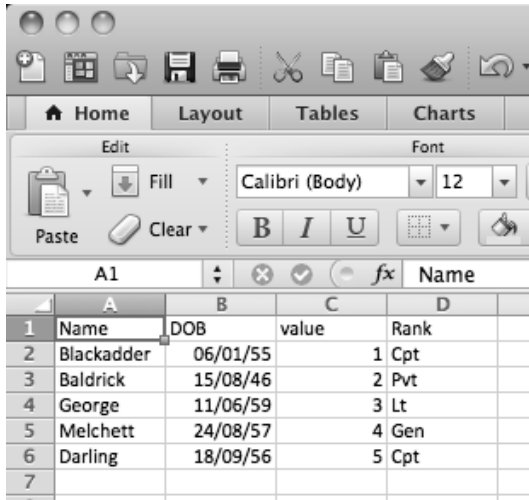
Solution: The easiest way to read data from an Excel spreadsheet into R is to export the spreadsheet to a delimited file like a comma-separated file and then import the CSV file as described in Rule 1.5.

Alternatively, the `read.xlsx` function from the `xlsx` can be used to read Excel worksheets (both older Excel formats as well as Excel 2007

and above formats) directly into R. The `xlsx` package depends on the `xlsxjars` and `rJava` packages so they need to be installed for `xlsx` to work.

The first argument to `read.xlsx` should be the path to the Excel spreadsheet, and the second argument, `sheetIndex`, takes a number representing which sheet to import from the Excel file. By default, the first line in the Excel sheet is assumed to be a header line, but that can be changed by setting the `header=FALSE` option.

The following Excel worksheet saved as the file `cunningplan.xlsx` can be imported with the following code:



	Name	DOB	value	Rank
1	Blackadder	06/01/55	1	Cpt
2	Baldrick	15/08/46	2	Pvt
3	George	11/06/59	3	Lt
4	Melchett	24/08/57	4	Gen
5	Darling	18/09/56	5	Cpt

```
> library(xlsx)
Loading required package: xlsxjars
Loading required package: rJava
> goesforth <- read.xlsx("Documents/cunningplan.xlsx", 1)
> goesforth
```

	Name	DOB	value	Rank
1	Blackadder	1955-01-06	1	Cpt
2	Baldrick	1946-08-15	2	Pvt
3	George	1959-06-11	3	Lt
4	Melchett	1957-08-24	4	Gen
5	Darling	1956-09-18	5	Cpt

Note that we set the sheet to load with the second argument. The `rowIndex` and `colIndex` options can be set to numeric vectors to indicate which rows and columns to extract from the worksheet, respectively. They are set to `NULL` by default, which means that all rows and columns are imported.

See also: If you have Perl installed on your computer, you can use the

`read.xls` function from the `gdata` package. It automates the process of saving the Excel sheet as a CSV file and reading it in R. See Rule 1.9 for an example of importing spreadsheet data through the clipboard.

1.7 Read data from an Excel spreadsheet under Windows

Problem: You want to read data stored in an Excel spreadsheet on a machine running Windows.

Solution: You can always use Rule 1.6 to read Excel spreadsheets under Windows. However, under Windows the package `RODBC` can be used to import data directly from an Excel spreadsheet into R.

To access the Excel file `example.xls` we first open a connection using `odbcConnectExcel`, then extract any information from the spreadsheet using the `sqlFetch` function before finally closing the ODBC connection with `odbcClose`. For example, the following dataset can be read into R by using the commands

	A	B	C	D	E	F	G
1	Id	Sex	Age	Score			
2	21	Male	14	Little			
3	26	Male	13	None			
4	27	Male	13	Moderate, severe			
5	29	Female	13	Little			
6	30	Female	15	Little			
7	31	Male	14	Moderate, severe			
8							
9							
10							

```
> library(RODBC)
> channel <- odbcConnectExcel("example.xls")
> indata <- sqlFetch(channel, "Sheet1")
> odbcClose(channel)
```

```
> indata
  Id    Sex Age          Score
1 21   Male 14         Little
2 26   Male 13           None
3 27   Male 13 Moderate, severe
4 29 Female 13         Little
5 30 Female 15         Little
6 31   Male 14 Moderate, severe
```

Note that some language installations of Excel rename the sheets so they are called, for example, “Ark1”, “Ark2”, ..., instead of “Sheet1”, “Sheet2”, etc. If that is the case you should provide the correct name for the sheet in the call to `odbcConnectExcel`.

It should also be pointed out that `odbcConnectExcel` will only work with English-language versions of the Microsoft drivers, which may or may not be installed in other locales.

See also: Excel spreadsheets can also be read directly from R using the `read.xls` function from the `xlsReadWrite` package. The `gdata` package also provides a function `read.xls` which works by translating the Excel file into a temporary CSV file using a Perl script and directly reading the CSV file. See Rule 1.9 for an example of importing spreadsheet data through the clipboard. See Rule 1.4 for more examples of the `RODBC` package.

1.8 Read data from a LibreOffice or OpenOffice Calc spreadsheet

Problem: You want to read data stored as a LibreOffice or OpenOffice Calc spreadsheet.

Solution: Start Calc, save the spreadsheet as a CSV file and use Rule 1.5 to import the spreadsheet.

See also: Rule 1.9 shows an example of importing spreadsheet data through the clipboard.

1.9 Read data from the clipboard

Problem: You have selected some data and copied them to the clipboard and want to import the selection into R.

Solution: Sometimes it is desirable to select data from a document, a web page, or from a spreadsheet and import the selection directly into R. This can be done by copying the selection to the clipboard and then subsequently importing the contents of the clipboard into R. This approach can be used on platforms that have the equivalent of a clipboard, which is the case for Windows, Mac OS X and machines running the X Window System used on many Linux systems.

The contents of the clipboard can be read using the value "clipboard" for the file option to `read.table` under Windows and X, any by using pipe ("pbpaste") under Mac OS X.

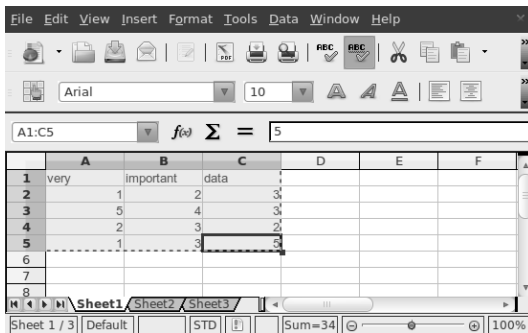


Figure 1.1: Selection of cells to be copied to the clipboard.

If we select some cells in, for example, an OpenOffice spreadsheet as shown in Figure 1.1 and copy the selection to the clipboard then we import the selection with the following code:

```
> mydata <- read.table(file="clipboard", header=TRUE) # Windows/X
> mydata <- read.table(pipe("pbpaste"), header=TRUE) # Mac OS X
> mydata
  very important data
1    1         2    3
2    5         4    3
3    2         3    2
4    1         3    5
```

Note that `read.table` reads and parses the input from the clipboard as if it had read the data from an ASCII file (see Rule 1.1). This causes

problems with empty cells and if there is text with spaces in any of the cells. If the data are separated by tabs on the clipboard — which is generally the case for spreadsheet data — we can specify the field separator to be tabs in the call to `read.table`, which will handle these two situations.

```
> mydata <- read.table(file="clipboard", sep="\t", header=TRUE)
```

On machines running the X11 Windows system the `"X11_clipboard"` value can be used for the `file` option to copy from the clipboard.

See also: The help file for the `file` function lists information about clipboards. Spreadsheet data on the clipboard are often stored in the Data Interchange Format (DIF) and the `read.DIF` function can be used to read DIF formats directly. `read.DIF` is sometimes more robust than using `read.table` when there are empty cells.

Importing data from other statistical software programs

1.10 Import a SAS dataset

Problem: You want to import a SAS dataset into R.

Solution: The `read.xport` function from the `foreign` package reads SAS datasets stored as SAS transport (XPORT) files.

SAS datasets are stored in different formats that depend on the operating system and the version of SAS. To read in SAS datasets it is necessary to save the SAS dataset as a SAS transport (XPORT) file since that can be read on any platform. The following code can be used from within SAS to store the SAS dataset `sasdata` in the XPORT format.

```
libname mydata xport "somefile.xpt";
```

```
/* Create a dataset in XPORT format and save it in  
 * the somefile.xpt file. The file is referenced  
 * internally in SAS by the name mydata.  
 * Here we take an existing SAS dataset called  
 * sasdata and put it into the mydata file.  
 */
```

```
DATA mydata.thisdata;  
  SET sasdata;
```

```
RUN;
```

Once the data are stored in the SAS XPORT format we can read the file directly using `read.xport`:

```
> library(foreign)  
> indata <- read.xport("somefile.xpt")
```

See also: The `read.xport` function from the `SASxport` package extends the functionality for reading SAS XPORT files when custom formats are present in the data.

1.11 Import an SPSS dataset

Problem: You want to import an SPSS dataset into R.

Solution: Datasets stored by the SPSS “save” and “export” commands can be read by the `read.spss` function from the `foreign` package. To read an SPSS dataset saved in the `spssfilename.sav` file we use the following command in R:

```
> library(foreign)  
> indata <- read.spss("spssfilename.sav", to.data.frame = TRUE)
```

The `to.data.frame=TRUE` option ensures that the SPSS data file is stored as a data frame in R. If that option is not included, the dataset is stored as a list.

1.12 Import a Stata dataset

Problem: You want to import a Stata dataset into R.

Solution: Datasets stored by the “SAVE” command in Stata can be read in R by the `read.dta` function from the `foreign` package. To read a Stata dataset saved as the file `statafile.dta` we use the following commands in R: