Use R!

Behavioral Research Data Analysis with R

Springer

Use R!

Series Editors:
Robert Gentleman Kurt Hornik Giovanni Parmigiani

For further volumes:
http://www.springer.com/series/6991

Use R!

Albert: Bayesian Computation with R
Bivand/Pebesma/Gómez-Rubio: Applied Spatial Data Analysis with R
Cook/Swayne: Interactive and Dynamic Graphics for Data Analysis:
With R and GGobi
Hahne/Huber/Gentleman/Falcon: Bioconductor Case Studies
Paradis: Analysis of Phylogenetics and Evolution with R
Pfaff: Analysis of Integrated and Cointegrated Time Series with R
Sarkar: Lattice: Multivariate Data Visualization with R
Spector: Data Manipulation with R

Yuelin Li • Jonathan Baron

Behavioral Research Data Analysis with R

Springer

Yuelin Li
Memorial Sloan-Kettering Cancer Center
Department of Psychiatry and Behavioral Sciences
641 Lexington Ave. 7th Floor
New York, New York 10022-4503
USA
liy12@mskcc.org

Jonathan Baron
Department of Psychology
University of Pennsylvania
3720 Walnut Street
Philadelphia, Pennsylvania 19104-6241
USA
baron@psych.upenn.edu

Series Editors:
Robert Gentleman
Program in Computational Biology
Division of Public Health Sciences
Fred Hutchinson Cancer Research Center
1100 Fairview Ave. N, M2-B876
Seattle, Washington 98109-1024
USA

Kurt Hornik
Department für Statistik und Mathematik
Wirtschaftsuniversität Wien Augasse 2-6
A-1090 Wien
Austria

Giovanni Parmigiani
The Sidney Kimmel Comprehensive Cancer
Center at Johns Hopkins University
550 North Broadway
Baltimore, MD 21205-2011
USA

ISBN 978-1-4614-1237-3 e-ISBN 978-1-4614-1238-0
DOI 10.1007/978-1-4614-1238-0
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011940221

© Springer Science+Business Media, LLC 2012
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This book is written for behavioral scientists who want to consider adding R to their existing set of statistical tools, or want to switch to R as their main computation tool. We aim primarily to help practioners of behavioral research make the transition to R. The focus is to provide practical advice on some of the widely used statistical methods in behavioral research, using a set of notes and annotated examples. We also aim to help beginners learn more about statistics and behavioral research. These are statistical techniques used by psychologists who do research on human subjects, but of course they are also relevant to researchers in others fields that do similar kinds of research.

We assume that the reader has read the relevant parts of R manuals on the CRAN website at `http://www.r-project.org`, such as "An Introduction to R", "R Data Import/Export", and "R Installation and Administration". We assume that the reader has gotten to the point of installing R and trying a couple of examples. We also assume that the reader has relevant experiences in using other statistical packages to carry out data analytic tasks covered in this book. The source code and data for some of the examples in the book can be downloaded from the book's website at: `http://idecide.mskcc.org/yl_home/rbook/`. We do not dwell on the statistical theories unless some details are essential in the appropriate use of the statistical methods. When they are called for, theoretical details are accompanied by visual explanations whenever feasible. Mathematical equations are used throughout the book in the hopes that reader will find them helpful in general, and specifically in reaching beyond the scope of this book. For example, matrix notations are used in the chapters covering linear regression and linear mixed-effects modeling because they are the standard notations found in statistics journals. A basic appreciation of mathematical notations may help the readers implement these new techniques before a packaged solution is available. Nevertheless, the main emphasis of this book is on the practical data analytic skills so that they can be quickly incorporated into the reader's own research.

The statistical techniques in this book represent many of statistical techniques in our own research. The pedagogical plan is to present straightforward solutions and add more sophisticated techniques if they help improve clarity and/or efficiency.

As can be seen in the first example in Chap. 1, the same analysis can be carried out by a straightforward and a more sophisticated method. Chapters 1–4 cover basic topics such as data import/export, statistical methods for comparing means and proportions, and graphics. These topics may be part of an introductory text for students in behavioral sciences. Data analysis can often be adequately addressed with no more than these straightforward methods. Chapter 4 contains plots in published articles in the journal *Judgment and Decision Making* (http://journal.sjdm.org/). Chapters 5–7 cover topics with intermediary difficulty, such as repeated-measures ANOVA, ordinary least square regression, logistic regression, and statistical power and sample size considerations. These topics are typically taught at a more advanced undergraduate level or first year graduate level.

Practitioners of behavioral statistics are often asked to estimate the statistical power of a study design. R provides a set of flexible functions for sample size estimation. More complex study designs may involve estimating statistical power by simulations. We find it easier to do simulations with R than with other statistical packages we know. Examples are provided in Chaps. 7 and 11.

The remainder of this book cover more advanced topics. Chapter 8 covers Item Response Theory (IRT), a statistical method used in the development and validation of psychological and educational assessment tools. We begin Chap. 8 with simple examples and end with sophisticated applications that require a Bayesian approach. Such topics can easily take up a full volume. Only practical analytic tasks are covered so that the reader can quickly adapt our examples for his or her own research. The latent regression Rasch model in Sect. 8.4.2 highlights the power and flexibility of R in working with other statistical languages such as WinBUGS/OpenBUGS. Chapter 9 covers missing data imputation. Chapters 10–11 cover hierarchical linear models applied in repeated-measured data and clustered data. These topics are written for researchers already familiar with the theories. Again, these chapters emphasize the practical data analysis skills and not the theories.

R evolves continuously. New techniques and user-contributed packages are constantly evolving. We strive to provide the latest techniques. However, readers should consult other sources for a fuller understanding of relevant topics. The R journal publishes the latest techniques and new packages. Another good source for new techniques is The Journal of Statistical Software (http://www.jstatsoft.org/). The R-help mailing list is another indispensable resource. User contributions make R a truly collaborative statistical computation framework. Many great texts and tutorials for beginners and intermediate users are already widely available. Beginner-level tutorials and how-to guides can be found online at the CRAN "Contributed Documentation" page.

This book originated from our online tutorial "Notes on the use of R for psychology experiments and questionnaires." Many individuals facilitated the transition. We would like to thank them for making this book possible. John Kimmel, former editor for this book at Springer, first encouraged us to write this book and provided continuous guidance and encouragement. Special thanks go to Kathryn Schell and Marc Strauss and other editorial staff at Springer on the preparation of the book. Several annonymous reviewers provided suggestions on how to improve the book.

We are especially indebted to the individuals who helped supply the data used in the examples, including the authors of the R packages we use, and those who make the raw data freely accessible online.

New York Yuelin Li
Philadelphia Jonathan Baron

Contents

Chapter 1
Introduction

1.1 An Example R Session

Here is a simple R session.

```
> help(sleep)
> x1 <- sleep$extra[sleep$group == 1]
> x2 <- sleep$extra[sleep$group == 2]
> t.test(x1, x2)
> sleep[c(1:3, 11:13), ]
> with(sleep, t.test(extra[group == 1],
+        extra[group == 2]))
> q()
```

The `help()` command prints documentation for the requested topic. The `sleep` dataset is a built-in dataset in R. It comes from William Sealey Gosset's article under the pseudonym Student (1908). It contains the effects of two drugs, measured as the extra hours of sleep as compared to controls. The vectors `x1` and `x2` are assigned the values of the extra hours of sleep in drugs 1 and 2, respectively. (a less than sign followed by a minus sign, `<-`, represents assignment) Two equal signs, `==`, represent the logical equal operator. The `t.test(x1, x2)` carries out an independent sample *t*-test of the sleep time between the two groups. The same analysis can be done using `with(sleep, t.test(extra[group == 1], extra[group == 2]))`. `sleep[c(1:3, 11:13), ]` prints observations 1 through 3 and 11 through 13. To exit the R program, type `q()`. Typing `q` without the parentheses prints out the contents of the function to quit R. Most functions are visible to the user in this way. The advantage of using built-in datasets is that they have already been imported. The next example describes how to import data from a text file.

The `sleep` data can be entered into a text file, the variable names on the first row, and the variables are separated by spaces.

DOI 10.1007/978-1-4614-1238-0_1, © Springer Science+Business Media, LLC 2012

```
 extra group ID
  0.7     1  1
 -1.6     1  2
 -0.2     1  3
 -1.2     1  4
 -0.1     1  5
  3.4     1  6
  3.7     1  7
  0.8     1  8
  0.0     1  9
  2.0     1 10
  1.9     2  1
  0.8     2  2
  1.1     2  3
  0.1     2  4
 -0.1     2  5
  4.4     2  6
  5.5     2  7
  1.6     2  8
  4.6     2  9
  3.4     2 10
```

Suppose the data entries are saved in a file named `t1.dat` in the directory `C:\\Documents and Settings\\usr1\\My Documents`, then this command imports the data and assigns it a name called `sleep.df`.

```
> sleep.df <- data.frame(read.table(file =
+"C:/Documents and Settings/usr1/My Documents/t1.dat",
+header = TRUE))
```

On a Windows platform, the double back slashes (\\) in a path name can be replaced with one forward slash (/). On Unix/Linux and Mac OS, one forward slash works fine. The `read.table()` function reads the data in `file`. It uses the first line of the raw data file (`header = TRUE`) to assign variable names to the three columns. Blank spaces in the raw data file are ignored. The `data.frame()` function converts the imported data into a data frame. The `sleep.df` data is now available for analysis (type `objects()` to see it). The example above shows some of the unique features of R. Most data analytic tasks in R are done through functions, and functions have parameters such as the options of `file` and `header` in the `read.table()` function. Functions can be nested, the output of one function can be fed directly into another. Some other basic R features are covered in the next section. These features make R flexible but more challenging to learn for beginners.

Some things are more difficult with R especially if you are used to using menus. With R, it helps to have a list of commands in front of you. There are lists in the on-line help and in the index of *An introduction to R* by the R Core Development Team, and in the reference cards listed in `http://finzi.psych.upenn.edu/`.

Some things turn out to be easier in R. Although there are no menus, the on-line help files are very easy to use, and quite complete. The elegance of the language helps too, particularly those tasks involving the manipulation of data. The purpose of this book is to reduce the difficulty of the things that are more difficult at first. Next we will go over a few basic concepts in R. The remainder of this chapter covers a few examples on how to take advantage of R's strengths.

1.2 A Few Useful Concepts and Commands

1.2.1 Concepts

In R, most commands are functions. The command is written as the name of the function, followed by parentheses, with the arguments (inputs) of the function in parentheses, separated by commas when there is more than one, e.g., `plot(swiss)` to plot a pairwise scatterplot of the `swiss` data. When there is no argument, the parentheses are still needed, e.g., `q()` to exit the program. A function is said to "return" its output when the output is printed or when we can set a variable equal to the output. For example, `sqrt(4)` returns (prints) 2 on the screen; and if we say `v1 <- sqrt(4)`, `v1` is set equal to the output of the function, or 2.

Some basic concepts in R are surprising to beginners. For example, the square of $\sqrt{7}$ is not 7.

```
> 7 == sqrt(7)^2
[1] FALSE
```

That is because floating point arithmetic is not exact.

```
> options(digits = 22)
> sqrt(7)^2
[1] 7.000000000000000888178
```

A solution is to compare `all.equal(sqrt(7)^2, 7)`.

In this book, we generally use names such as `x1` or `file1`, that is, names containing both letters and a digit, to indicate variable names that the user makes up. Really, these can be of any form. We use the number simply to clarify the distinction between a made up name and a key word with a predetermined meaning in R. R is case sensitive; for example, `X` and `x` can stand for different things. We generally use upper-case data objects like `X`, `Y`, and `M` to represent matrices or arrays; and lower-case objects to represent vectors. Although most commands are functions with the arguments in parentheses, some arguments require specification of a key word with an equal sign and a value for that key word, such as `source("myfile1.R", echo = T)`, which means read in `myfile1.R` and echo the commands on the screen. It helps to add spaces between input parameters, so that the extra spaces in `echo = T` make it easier to read than `echo=T`. But that is not necessary.

Key words can be abbreviated (e.g., `e = T`). In addition to the idea of a function, R has objects and modes. Objects are anything that you can give a name. There are many different classes of objects. The main classes of interest here are *vector*, *matrix*, *factor*, *list*, and *data frame*. The mode of an object tells what kind of things are in it. The main modes of interest here are `logical`, `numeric`, and `character`.

We sometimes indicate the class of object (vector, matrix, factor, etc.) by using `v1` for a vector, `m1` for a matrix, and so on. Most R functions, however, will either accept more than one type of object or will "coerce" a type into the form that it needs.

The most interesting object is a data frame. It is useful to think about data frames in terms of rows and columns. The rows are subjects or observations. The columns are variables, but a matrix can be a column too. The variables in a data frame can be of different classes.

The behavior of any given function, such as `plot()`, `aov()` (analysis of variance), or `summary()` depends on the object class and mode to which it is applied. A nice thing about R is that you almost do not need to know this, because the default behavior of functions is usually what you want. One way to use R is just to ignore completely the distinction among classes and modes, but *check* every step (by typing the name of the object it creates or modifies). If you proceed this way, you will also get error messages, which you must learn to interpret. Most of the time, again, you can find the problem by looking at the objects involved, one by one, typing the name of each object.

Sometimes, however, you must know the distinctions. For example, a factor is treated differently from an ordinary vector in an analysis of variance or regression. A factor is what is often called a categorical variable. Even if numbers are used to represent categories, they are not treated as ordered. If you use a vector and think you are using a factor, you can be misled.

1.2.2 Commands

As a reminder, here is a list of some of the useful commands that you should be familiar with, and some more advanced ones that are worth knowing about. Some of the more basic commands help you organize your work.

1.2.2.1 Working Directory

It helps to get into the habit of separating R sessions into different working directories specific to different projects or data analytic tasks. Here is why. On Windows, R starts in the user's default HOME directory (e.g., `getwd()` returns `C:/Documents and Settings/usr1/My Documents`). On exiting R, the user is prompted to save the current session in a `.RData` file under that directory by default. Eventually, this `.RData` file collects too many objects to be managed efficiently. You may organize R sessions into subdirectories, for example, called `project1`, `project2`, and `project3` under your home directory `C:/Documents and Settings/usr1/My Documents/`. If you are working on `project1`, you double click the R icon on your Windows desktop to launch R, then you immediately type `setwd("C:/Documents and Settings/usr1/My Documents/project1/")` to switch your working directory to `project1`. Then you can type `load(".RData")` to retrieve a previously saved session. This is probably the first thing you do each time you run R on Windows. These changes can also be set interactively using the menu. Note that R recognizes the forward slashes in the path name.

The `setwd()` command is usually not necessary if you are running R from a Unix/Linux command line. Typically, you are already in the working directory before R is called from the command line.

On a computer running the Mac OS, it depends on whether or not your R is a binary version with a graphical user interface or a version compiled from source code. R compiled from source on a Macintosh computer works like a Unix/Linux R from a command line terminal and `setwd()` is not necessary.

Another advantage of separating R sessions in different working directories is that it allows easier tracking of the command history file. All the commands typed in an R session are saved upon exit in a file called `.Rhistory` under the working directory. You can use a text editor to edit the `.Rhistory` file into a command syntax script. Then you can run R in batch mode. For example, suppose the `.Rhistory` file under `project1` contains these lines:

```
> help(sleep)
> x1 <- sleep$extra[sleep$group == 1]
> x2 <- sleep$extra[sleep$group == 2]
> t.test(x1, x2)
> sleep[c(1:3, 11:13), ]
> t.test(extra ~ group, data = sleep)
> with(sleep, t.test(extra[group == 1],
+  extra[group == 2]))
> q()
```

We can take out the first and last lines and save the edited file as `sleep.R`. Then we can run R in batch mode by calling `R CMD BATCH sleep.R`. The ouput is saved in `sleep.Rout` under the `project1` directory. The output in `sleep.Rout` file then can be shared with others.

1.2.2.2 Getting Help

`help.start()` starts the browser version of the help files. (But you can use `help()` without it.) With a fast computer and a good browser, it is often simpler to open the html documents in a browser while you work and just use the brower's capabilities. help(plot) prints the help available about plot, or help(command1) to print the help for command1. Sometimes you only need the names of the parameters, which can be printed by args(command1).

help.search("keyword1") searches keywords for help on this topic. apropos(topic1) or apropos("topic1") finds commands relevant to topic1, whatever it is. example(command1) prints an example of the use of the command. This is especially useful for graphics commands. Try, for example, example(contour), example(dotchart), example(image), and example(persp).

1.2.2.3 Installing Packages

The R base system is lean. It contains only the essential components. Additional packages can be installed when needed. For example, install.packages (c("ltm","psych")) installs the packages called ltm and psych from an archive of your choice, if your computer is connected to the internet. You do not need the c() if you just want one package. You should, at some point, make sure that you are using the CRAN mirror page that is closest to you. If you live in the U.S., you should have a .Rprofile file with options(CRAN = "http://cran.us.r-project.org") in it. There are other mirror sites in the U.S. On Windows, you have the option to interactively select a mirror site from a list in a menu if one is not already set. Other useful functions for managing packages include installed.packages() to show details of all installed packages and update.packages() to update the packages that you have installed to their latest version.

To install packages from the Bioconductor set (tools and resources for computational biology), see the online instructions (http://www.bioconductor.org/install/, last accessed, September, 2011).

When packages are not on CRAN, you can download them and use R CMD INSTALL package1.tar.gz from a Unix/Linux command line. On Windows, you would need to open a DOS command prompt, change directory to where package1.tar.gz is saved, then type the command C:\"Program Files"\R\R-2.13.0\bin\R.exe CMD BATCH package1.tar.gz.

1.2.2.4 Assignment, Logic, and Arithmetic

One of the most frequently typed commands is the assignment command, <-. It assigns what is on the right of the arrow to what is on the left. (If you use ESS, the _ key (underscore) will produce this arrow with spaces, a great convenience.) Typing the name of the object prints the object. For example, if you say:

```
> t1 <- c(1, 2, 3, 4, 5)
> t1
```

you will see `1 2 3 4 5`. The object `t1` gets a numeric vector of five numbers, put together by the `c()` function. Beginners sometimes do `c <- c(1,2,3)`. R will let you do it, and will not generate an error if you next do `x <- c(4, 5, 6)`. R knows what to do with `x` because your local copy of `c` is a numeric vector and the system copy of `c` is a function. However, it is better not to assign values to `c()` or any other system functions to minimize confusions.

Logical objects can be true or false. Some functions and operators return `TRUE` or `FALSE`. For example, `1 == 1`, is `TRUE` because 1 does equal 1. Likewise, `1 == 2` is `FALSE`, and `1 < 2` is `TRUE`. But beware, `sqrt(2)^2 == 2` is `FALSE` because they have different internal floating-point representations in R. A better test for the equality between two floating-point numbers is provided by the function `all.equal()`, `all.equal(sqrt(2)^2, 2)` is `TRUE`.

Use `all()`, `any()`, `|`, `||`, `&`, and `&&` to combine logical expressions, and use `!` to negate them. The difference between the `|` and the `||` form is that the shorter form, when applied to vectors, etc., returns a vector, while the longer form stops when the result is determined and returns a single `TRUE` or `FALSE`. Set functions operate on the elements of vectors: `union(v1,v2)`, `intersect(v1,v2)`, `setdiff(v1,v2)`, `setequal(v1,v2)`, `is.element(element1,v1)` (or, `element1 %in% v1`). Arithmetic works. For example, `-t1` yields `-1 -2 -3 -4 -5`. It works on matrices and data frames too. For example, suppose `m1` gets the matrix `m1 <- matrix(c(1,2,3,4,5,6), nrow=2, byrow=T)`.

```
1 2 3
4 5 6
```

Then `m1 * 2` is

```
2  4  6
8 10 12
```

Matrix multiplication works too. Suppose `m2` is the matrix
`m2 <- matrix(c(1,1,1,2,2,2), ncol=2)`

```
1 2
1 2
1 2
```

then `m1 %*% m2` is

```
 6 12
15 30
```

and `m2 %*% m1` is

```
9      12     15
9      12     15
9      12     15
```

You can also multiply a matrix by a vector using matrix multiplication, vectors are aligned vertically when they come after the `%*%` sign and horizontally when they come before it. This is a good way to find weighted sums, as we shall explain.

For ordinary multiplication of a matrix times a vector, the vector is vertical and is repeated as many times as needed. For example `m2 * 1:2` yields

```
1 4
2 2
1 4
```

Ordinarily, you would multiply a matrix by a vector when the length of the vector is equal to the number of rows in the matrix.

1.2.2.5 Loading and Saving

Additional functions not activated at startup have to be loaded by `library(pkg)` or `require(pkg)`, where `pkg` is the unquoted name of the package. A list of packages can be found online at `cran.r-project.org`. A useful library for psychology is `mva` (multivariate analysis). To find the contents of a library such as `mva` before you load it, say `library(help = mva)`. The `ctest` library is already loaded when you start R. Other useful functions include:

- `source("file1")` runs the commands in `file1`.
- `sink("file1")` diverts output to `file1` until you say `sink()`.
- `save(x1,file="file1")` saves object `x1` to file `file1`.
- To read in the file, use `load("file1")`.
- `q()` quits the program. `q("yes")` saves everything.
- `write(object, "file1")` writes a matrix or some other object to `file1`.
- `write.table(object1, "file1")` writes a table and has an option to make it comma delimited, so that a spreadsheet program can read it. See the help file, but to make it comma delimited, say `write.table(object1, "file1", sep=",")` or simply `write.csv(object1, "file1")`
- `round()` produces output rounded off, which is useful when you are cutting and pasting R output into a manuscript (e.g., `round(t.test(v1)$statistic, 2)` rounds off the value of `t` to two places). Other useful functions are `format` and `formatC`. For example, if we assign `t1 <- t.test(v1)` then the following command prints out a nicely formatted result, suitable for dumping into a paper:

```
> x1 <- sleep$extra[sleep$group == 1]
> x2 <- sleep$extra[sleep$group == 2]
> t1 <- t.test(x1, x2)
> print(paste("(t_{",t1[[2]],"}=",
+   formatC(t1[[1]],format="f",digits=2),", p=",
+   formatC(t1[[3]],format="f"),")",sep=""),
+   quote=FALSE)
```

This works because the output of the `t.test()` assigned to `t1` is actually a list, and the numbers in the double brackets refer to the elements of the list.

- `read.table("file1")` reads in data from a file. The first line of the file can (but need not) contain the names of the variables in each column.

1.2.2.6 Dealing with Objects

All objects created by the user are stored in an R environment called `.GlobalEnv` (can also be accessed by `globalenv()`. The `ls()` and `objects()` functions lists all the active objects in `.GlobalEnv`. Other system files, such as the built-in datasets and statistical functions, are stored in various packages. A list of the loaded packages can be found by `search()`. Note that `search()` numbers the packages.

```
> search()
 [1] ".GlobalEnv"
 [2] "package:stats"
 [3] "package:graphics"
 [4] "package:grDevices"
 [5] "package:utils"
 [6] "package:datasets"
 [7] "package:methods"
 [8] "Autoloads"
 [9] "package:base"
```

Thus, `ls(pos = 2)` or simply `ls(2)` shows all objects in the `package:stats` (or by `objects(2)`). To remove one or more data objects, do `rm(object1)` to remove only `object1` or `rm(x1, x2, v1, v2, object2, object3)` to remove multiple objects. Type `rm(list=ls())` to remove all objects in the current environment. Be careful with this because the `rm()` function assumes you know what you are doing so it does not prompt you for a confirmation. `attach(df1)` makes the variables in the data frame `df1` active and available generally. Sometimes you are working in one directory but you need to access data saved in another directory, type `attach("/another/directory/.RData")` to gain access to data objects saved in that directory. `names(obj1)` prints the names, e.g., of a matrix or data frame. `typeof()`, `mode()`, and `class()` tell you about the properties of an object.

1.3 Data Objects and Data Types

One of the most basic data objects in R is a vector. A vector can be put together by the function `c()`. We can calculate its `length`, `mean`, and other properties.

```
> x <- c(1, 2, 3, 4, 5, 6, 7)
> length(x)
[1] 7
> mean(x)
[1] 4
```

We can refer to the elements of a vector in various ways.

```
> x[6]
[1] 6
> x[-6]          # all elements except the 6th
[1] 1 2 3 4 5 7
> x[2:4]         # : represents a sequence
[1] 2 3 4
> x[c(1, 4, 7)]
[1] 1 4 7
```

A colon, `:`, is a way to abbreviate a sequence of numbers, e.g., `1:5` is equivalent to 1,2,3,4,5. A sequence of evenly spaced numbers can be generated by `seq(from = 1, to = 6, length = 20)` (20 evenly spaced numbers from 1 to 6) or `seq(from = -3, to = 3, by = 0.05)` (from -3 to $+3$ in increment of 0.05). `c(number.list1)` makes the list of numbers (separated by commas) into a vector object. For example, `c(1,2,3,4,5)` (but `1:5` is already a vector, so you do not need to say `c(1:5)`). `rep(v1,n1)` repeats the vector `v1` `n1` times. For example, `rep(c(1:5),2)` is `1,2,3,4,5,1,2,3,4,5`. `rep(v1,v2)` repeats each element of the vector `v1` a number of times indicated by the corresponding element of the vector `v2`. The vectors `v1` and `v2` must have the same length. For example, `rep(c(1,2,3),c(2,2,2))` is `1,1,2,2,3,3`. Notice that this can also be written as `rep(c(1,2,3),rep(2,3))`. (See also the function `gl()` for generating factors according to a pattern.)

1.3.1 Vectors of Character Strings

R is not intended as a language for manipulating text (unlike Perl, for example), but it is surprisingly powerful. If you know R you might not need to learn Perl. Strings are character variables that consist of letters, numbers, and symbols. A `c("one", "two", "3")` is a vector of character strings. You can use

```
> paste("one", "two", "3", sep = ":")
[1] "one:two:3"
```

to paste three character strings together into one long character string. Or to unpaste them by

```
> strsplit(paste("one", "two", "3", sep = ":"), ":")
[[1]]
[1] "one" "two" "3"
```

`grep()`, `sub()`, `gsub()`, and `regexpr()` allow you to search for, and replace, parts of strings.

The set functions such as `union()`, `intersect()`, `setdiff()`, and `%in%` are also useful for dealing with databases that consist of strings such as names and email addresses.

Calculating date and time differences needs special care because of leap years and other complications. Raw data in character strings of date and time should be converted into the POSIX date time classes using the `strptime()` function. Suppose, we have the birth dates of two children and today's date.

```
> bdate <- strptime(c("2/28/2002", "3/05/2006"),
+ format = "%m/%d/%Y")
> today <- strptime(c("2/28/2008"),
+ format = "%m/%d/%Y")
```

The option `format="%m/%d/%Y"` specifies how the date character string is formatted, by month, day, and the four-digit year (separated by forward slashes). The first child was born on 2/28/2002, precisely six years old on 2/28/2008. The second child's age in years is 1 because the child has not yet reached 2 years of age. You might be tempted to calculate age by:

```
> difftime(today, bdate, units="days")/365.25
```

But you get 5.999 and 1.985, which cannot be easily fixed by rounding. As of R-2.13.1, `difftime()` does not yet offer a `"years"` unit. Decimal age values of 5.999 and 1.985 may be acceptable for practical purposes, for example, in describing the average age of research study participants. However, they do not match the way we typically treat age as an non-negative integer.

A solution uses the components of a POSIX date.[1]

```
> age <- today$year - bdate$year
> age
[1] 6 2
> t1 <- bdate$mon + bdate$mday/31; t1
[1] 1.56 2.10
> t2 <- today$mon + today$mday/31; t2
[1] 1.56
> ti <- t2 < t1
> age[ti] <- age[ti] - 1
> age
[1] 6 1
```

The `$mon` component of a date variable takes on numeric values of 0, 1, 2, ..., 11 for January, February, March, and December, respectively. The `$mday` component

[1]Provided by Brian Ripley in R-help mailing list, https://stat.ethz.ch/pipermail/r-help/2007-September/141440.html, last accessed April 20, 2011.

represents the day of the month. So the children are 1.56 and 2.10 months from January 1, 2006; and `today` is 1.56 months from January 1, 2008. The division by 31 yields an approximated fraction of a month. So that the first child is 6 years of age and the second child is 1 year of age. The `difftime()` function provides time units in seconds, minutes, hours, days, and weeks. The resolution of weeks is usually enough for a time to event analysis. However, we are not limited by the restrictions of existing functions when we take advantage of the POSIX date components. The `strptime()` function is especially useful when date variables are entered into a spreadsheet program as character strings.

There are many other powerful features of R's character strings. You can even use these functions to write new R commands as strings, so that R can program itself. Just to see an example of how this works, try `eval(parse(text = "t.test(1:5)"))`. The `parse()` function turns the text into an R expression, and `eval()` evaluates and runs the expression. So this is equivalent to typing `t.test(1:5)` directly. But you could replace `t.test(1:5)` with any string constructed by R itself.

1.3.2 Matrices, Lists, and Data Frames

The call to `matrix(v1, nrow = 2, ncol = 3)` makes the vector `v1` into a 2x3 matrix. You do not need to specify both `nrow` and `ncol`. You can also use key words instead of using position to indicate which argument is which, and then you do not need the commas. For example, `matrix(1:10, ncol=5)` represents the matrix

$$\begin{bmatrix} 1 & 3 & 5 & 7 & 9 \\ 2 & 4 & 6 & 8 & 10 \end{bmatrix}.$$

Notice that the matrix is filled column by column. To fill the matrix by rows, do `matrix(1:10, ncol = 5, byrow = TRUE)`.

`cbind(v1,v2,v3)` puts vectors `v1, v2,` and `v3` (all of the same length) together as columns of a matrix. You can of course give this a name, such as `mat1 <- cbind(v1,v2,v2)`.

Many R functions require that you collect variables in a `data.frame()` object, for example, `datc <- data.frame(v1, v2, v3)`. Note that `v1, v2, and v3` must be of the same length. A data frame can include vectors of factors as well as numeric vectors.

```
> ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,
+ 5.33,5.14)
> trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,
+ 4.32,4.69)
> group <- gl(2,10,20, labels=c("Ctl","Trt"))
```