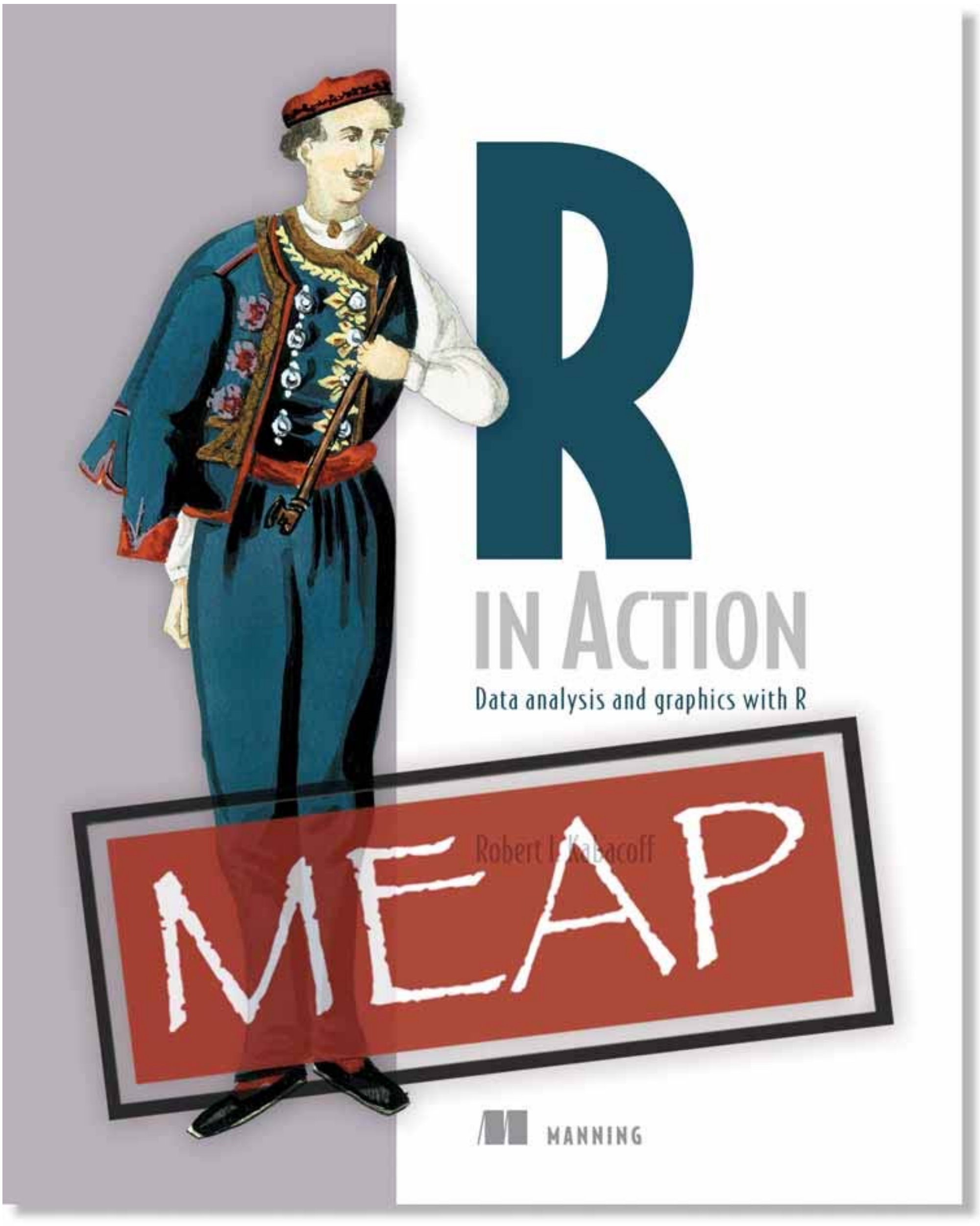

R
IN ACTION
Data analysis and graphics with R
Robert I. Kabacoff
MEAP
MANNING

MEAP Edition
Manning Early Access Program

1 Introduction to R

This Chapter covers:

- Installing R
- Understanding the R language
- Running programs
-

How we analyze data has changed dramatically in recent years. With the advent of personal computers and the internet, the sheer volume of data we have available has grown enormously. Companies have terabytes of data on the consumers they interact with, while governmental, academic, and private research institutions have extensive archival and survey data on every manner of research topic. Gleaning information (let alone wisdom) from these massive stores of data has become an industry in itself. At the same time, presenting the information in easily accessible and digestible ways has become increasingly challenging.

The science of data analysis (statistics, psychometrics, econometrics, machine learning) has kept pace with this explosion of data. Before personal computers and the Internet, new statistical methods were developed by academic researchers who published their results as theoretical papers in professional journals. It could take years for these methods to be adapted by programmers and incorporated into the statistical packages widely available to the data analysts. Today, new methodologies appear *daily*. Statistical researchers publish new and improved methods, along with code to produce them, on easily accessible websites And the code is typically written for, or easily adapted to freely available statistical packages like R.

The advent of personal computers had another effect on the way we analyze data. When data analysis was carried out on mainframe computers, computer time was precious and difficult to come by. Analysts would carefully set up a computer run with all the parameters and options thought to be needed. When the procedure ran, the resulting output could be dozens or hundreds of pages long. The analyst would sift through this output, extracting useful material and discarding the rest. With the cheap and easy access afforded by personal computers, data analysis now follows a different paradigm.

Rather than setting up a complete data analysis at once, the process has become highly interactive, with the output from each stage serving as the input for the next stage An example of a typical analysis is presented in figure 1.1. At any point, the cycles may include transforming the data, imputing missing values, adding or deleting variables, and looping back through the whole process again. The process stops when the analyst believes he or she understands the data intimately and has answered all the relevant questions that can be answered.

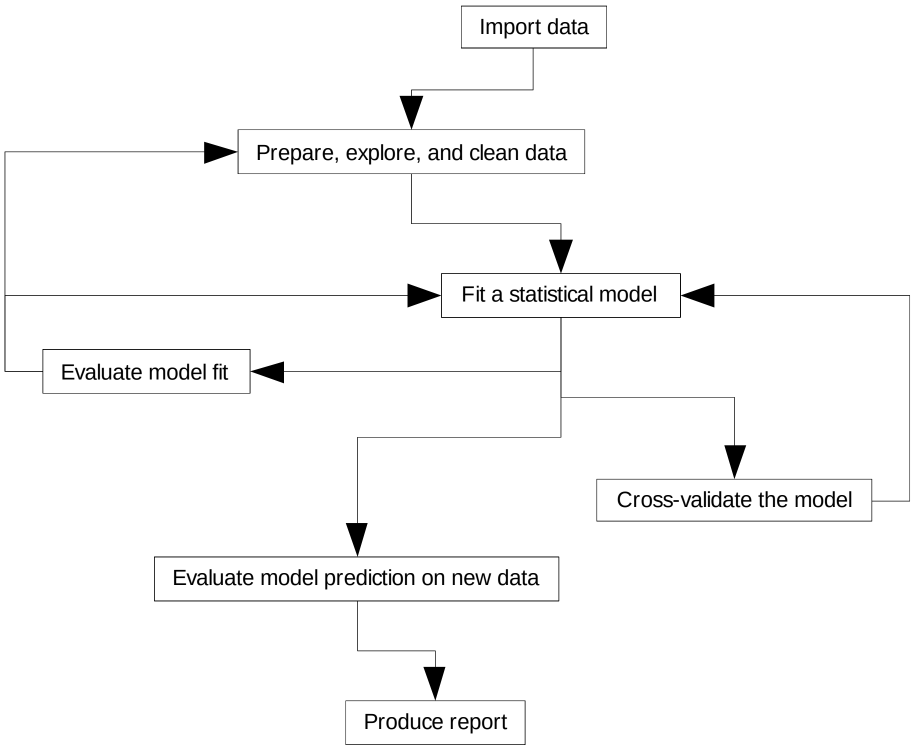


Figure 1.1 Steps in a typical data analysis

The advent of personal computers (and especially the availability of high resolution monitors) has also had an impact on how results are understood and presented. A picture *really can* be worth a thousand words, and human beings are very adept at extracting useful information from visual presentations. Modern data analysis increasingly relies on graphical presentations to uncover meaning and convey results.

Today's data analysts need to be able to access data from a wide range of sources (database management systems, text files, statistical packages, and spreadsheets), merge them together, clean and annotate them, analyze them with the latest methods, present the findings in meaningful and graphically appealing ways, and incorporate the results into attractive reports that can be distributed to stakeholders and the public. As you will see in the following pages, R is a comprehensive software package that is ideally suited to accomplish these goals.

1.1 Why use R?

R is a language and environment for statistical computing and graphics, similar to the S language originally developed at Bell Labs. It is an open source solution to data analysis that is supported by a large and active worldwide research community. R has many features to recommend it:

- Most commercial statistical software platforms cost thousands, if not tens of thousands of dollars. R is free! If you are a teacher or a student, the benefits are obvious.
- R runs on a wide variety of platforms including Windows, UNIX and MacOS X.
- R is a comprehensive statistical platform, offering all manner of data analytic techniques.
- R has state-of-the-art graphics capabilities.
- R provides an unparalleled platform for programming new statistical methods in an easy and straightforward manner.
- R contains advanced statistical routines not yet available in other packages.

An example of R's graphic capabilities can be seen in figure 1.2. This graph, created with a single line of code, describes the relationships between income, education, and prestige for blue collar, white collar, and professional jobs. Technically, it is a scatterplot matrix with groups displayed by color and symbol, two types of fit lines (linear and loess), confidence ellipses, and two types of density display (kernel density estimation, and rug plots). If these terms are unfamiliar to you, don't worry. We will cover them in later chapters. For now, trust me that they are really cool (and that the statisticians reading this are salivating).

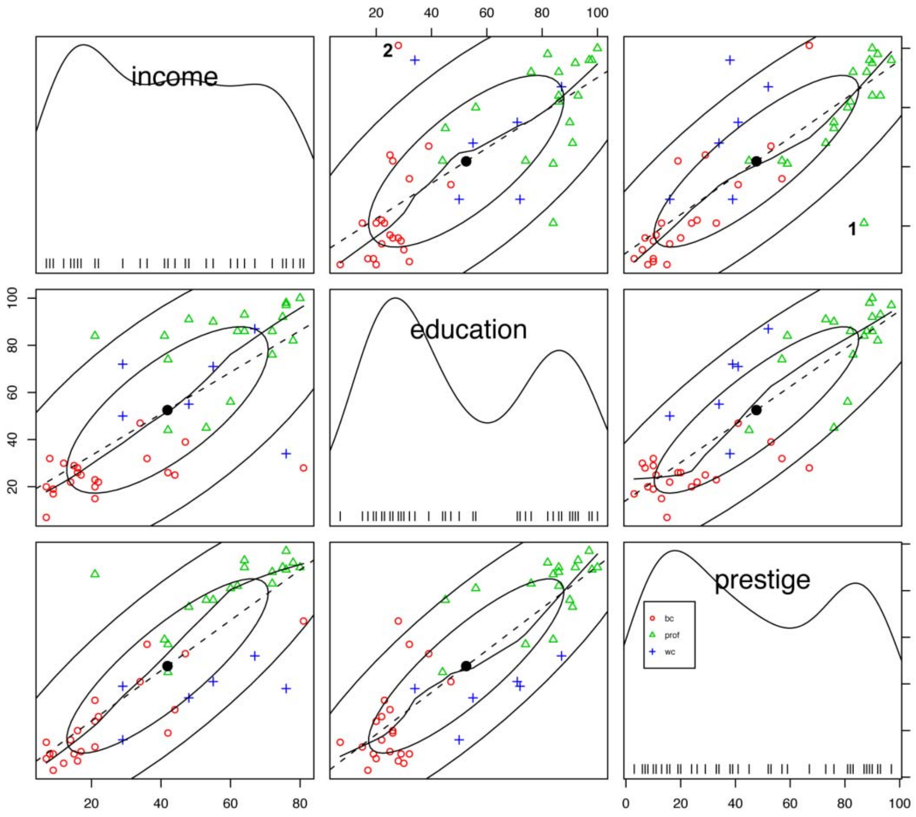


Figure 1.2 Relationships between income, education, and prestige for blue collar, white collar, and professional jobs. Source: car package written by John Fox.

Basically, this graph indicates that:

- Education, income and job prestige are linearly related.
- In general, blue collar jobs involve lower education, income and prestige, while professional jobs involve higher education, income, and prestige. White collar jobs fall in between.
- There are some interesting exceptions. Ministers (the point labeled 1) have high prestige and low income. RR Engineers (the point labeled 2) have high income and low education.
- Education (and possibly prestige) are distributed bimodally, with more scores as the high end and low end, than in the middle.

We will have much more to say about this type of graph in chapter 5.

Unfortunately, R can have a steep learning curve. Because it can do so much, the documentation and help files available can be voluminous. Additionally, because much of the functionally comes from optional modules created by independent contributors, this documentation can be scattered and difficult to locate. In fact, getting a handle on all that R can do is a challenge.

The goal of this book is to make access to R quick and easy. We will tour the many features of R, covering enough material to get you started on your data, with pointers on where to go when you need to learn more. Let's begin by installing the program.

1.2 Obtaining and installing R

R is freely available from the Comprehensive R Archive Network (CRAN) at http://cran.r-project.org. Precompiled binaries are available for Linux, MacOS X, and Windows. Follow directions for installing the base product on the platform of your choice. Later we'll talk about adding additional functionality through optional modules called packages (also available from CRAN).

1.3 Working with the R interface

R is a case-sensitive, interpreted language. You can enter commands one at a time at the command prompt (>) or run a set of commands from a source file. There are a wide variety of data types, including vectors, matrices, dataframes (similar to datasets), and lists (collections of objects). We will discuss each of these data types in chapter 2.

Most functionality is provided through built-in and user-created functions and all data objects are kept in memory during an interactive session. Basic functions are available by default. Other functions are contained in packages that can be attached to a current session as needed.

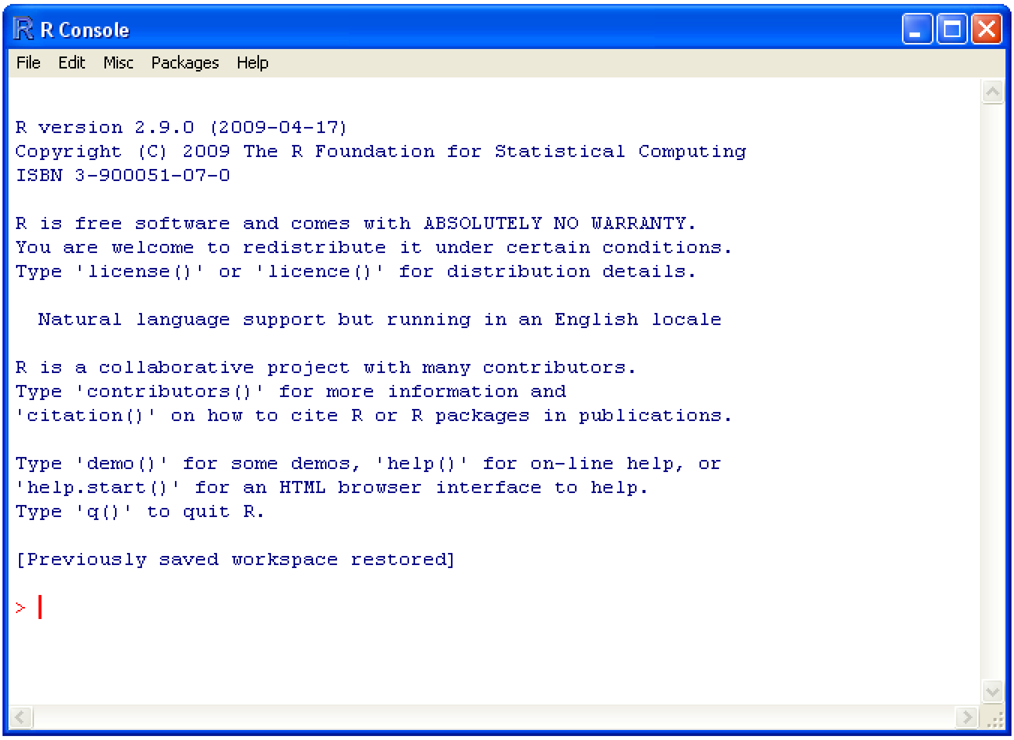
Statements consist of functions and assignments. R uses the symbol <- for assignments, rather than the typical = sign. For example, the statement

```
x <- rnorm(5)
```

creates a vector named x containing 5 random deviates from a standard normal distribution. Comments are preceded by the # symbol. Any text appearing after the # is ignored by the R interpreter.

1.3.1 Getting Started

If you are using Microsoft Windows, launch R from the Start Menu. On a Mac, double click the R icon in the Applications folder. For Linux, type R at the command prompt of a terminal window. This will start the R interface (see figure 1.3 for an example).



```
R version 2.9.0 (2009-04-17)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

>
```

Figure 1.3 Example of the R interface on Microsoft Windows XP.

To get a feel for the interface, let's work through a simple contrived example. Let's say that we are studying physical development and we have collected the ages and weights of 10 infants in their first year of life (see table 1.1). We are interested in the distribution of the weights and their relationship to age.

Table 1.1 The heights and weights of ten infants.

Age (mo.)	Weight (kg.)
01	4.4
03	5.3

05	7.2
02	5.2
11	8.5
09	7.3
03	6.0
09	10.4
12	10.2
03	6.1

Note: These are fictional data.

We will enter the age and weight data as vectors, using the function c(), which combines its arguments into a vector or list. Then we will get the mean and standard deviation of the weights, the correlation between age and weight, and plot the relationship between age and weight so that we can inspect any trend visually. The q() function will end the session and allow us to quit.

Listing 1.1 A sample R session

```
> # A two variable example
> age <- c(1,3,5,2,11,9,3,9,12,3)
> weight <- c(4.4,5.3,7.2,5.2,8.5,7.3,6.0,10.4,10.2,6.1)
> mean(weight)
[1] 7.06
> sd(weight)
[1] 2.077498
> cor(age,weight)
[1] 0.9075655
> plot(age,weight)
> q()
```

We can see from listing 1.1, that the mean weight for these 10 infants is 7.06 kilograms, that the standard deviation is 2.08 kilograms, and that there is strong linear relationship between age in months and weight in kilograms (correlation = 0.91). The relationship can also be seen in scatterplot in figure 1.4. Not surprisingly, as infants get older, they tend to weigh more.

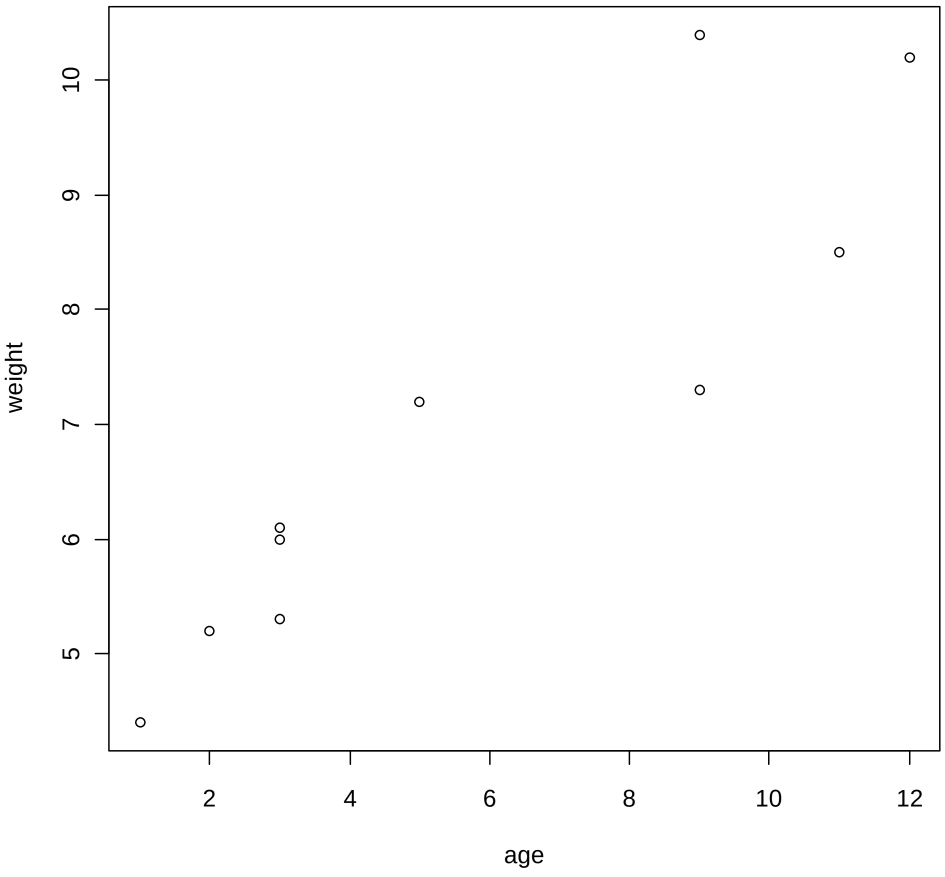


Figure 1.4 Scatterplot of infant age (mo) by weight (kg).

DEMONSTRATIONS

To get a sense of what R can do graphically, enter `demo(graphics)`. A sample of the graphs produced is included in figure 1.5. Other demonstrations include demo(Hershey), demo(persp), and demo(image). To see a complete list of demonstrations, enter demo() without parameters.

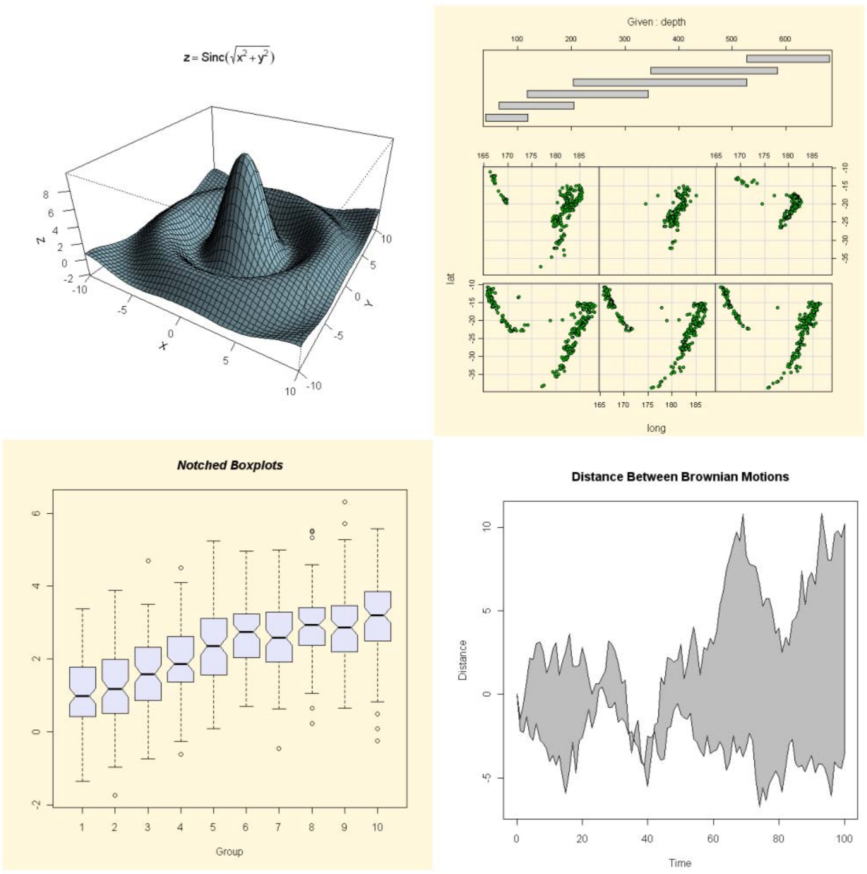


Figure 1.5 A sample of the graphs created with the `demo` function

1.3.2 Getting help

R provides extensive help facilities and learning to navigate them will help you significantly in your programming efforts. The built-in help system provides details, references, and examples of any function contained in a currently installed package. Help is obtained using the functions listed in table 1.2.

Table 1.2 R help functions.

Function	**Action**
`help.start()`	General help
`help(foo) or`	Help on function `foo`

`?foo`	
`help.search("foo")` or `??foo`	Search the help system for instances of the string `foo` (note the quotation marks)
`example(foo)`	Examples of function `foo`
`RSiteSearch("foo")`	Search for the string `foo` in help manuals and archived mailing lists (note the quotation marks)
`vignette()`	List all available vignettes for currently installed packages
`vignette("foo")`	Display specific vignettes for topic `foo (note the quotation marks)`

The function `help.start()` opens a browser window with access to introductory and advanced manuals, FAQs, and reference materials. The `RSiteSearch()` function searches for a given topic in online help manuals and archives of the R-Help discussion list and returns the results in a browser window. The vignettes returned by the `vignette()` function are practical introductory articles provided in PDF format. Not all packages will have vignettes. As you can see, R provides extensive help facilities and learning to navigate them will definitely aid your programming efforts.

1.3.3 The workspace

The workspace is your current R working environment and includes any user-defined objects (vectors, matrices, functions, dataframes, lists). At the end of an R session, you can save an image of the current workspace that is automatically reloaded the next time R starts. Commands are entered interactively at the R user prompt. You can use the up and down arrow keys to scroll through your command history. This allows you to select a previous command, edit it if desired, and resubmit it using the enter key.

The current working directory is the directory R will read files from and save results to by default. You can find out what the current working directory is by using the `getwd()` function. You can set the current working directory by using the `setwd()` function. If you need to input a file that is not in the current working directory, use the full path name in the call. Always enclose the names of files and directories from the operating system in quote marks.

Some standard commands for managing your workspace are listed in table 1.3.

Table 1.3 Functions for managing the R workspace

Function	**Action**
`getwd()`	List the current working directory

`ls()`	List the objects in the current workspace
`setwd("mydirectory")`	Change the current working directory to `mydirectory`
`help(options)`	Learn about available options
`options()`	View or set current option settings
`history(#)`	Display your last # commands (default = 25)
`savehistory("myfile")`	Save the commands history to `myfile` (default = `.Rhistory`)
`loadhistory("myfile")`	Reload a commands history (default = `.Rhistory`)
`save.image("myfile")`	Save the workspace to myfile (default = .`RData`)
`save(objectlist,file="myfile")`	Save specific objects to a file
`load("myfile")`	Load a workspace into the current session (default = .`RData`)
`q()`	Quit R. You will be prompted to save the workspace.

To see these commands in action, take a look at listing 1.2.

Listing 1.2 An example of commands used to manage the R workspace

```
setwd(“C:/myprojects/project1”)          1
options()                                2
options(digits=3)
x <- runif(20)                           3
summary(x)                               4
hist(x)
savehistory()                            5
save.image()
q()
```

1 Set the current working directory to C:/myprojects/project1
2 View currently set options and set numbers to display with 3 digits after the decimal place
3 Create a vector with 20 uniform random deviates
4 Print summary statistics and a histogram
5 Save a your commands history to the file .Rhistory and your workspace (including the vector x) to the file .RData

Note the forward slashes in the path name of the setwd() command. R treats the backslash "\" as an escape character. Even when using R on a Windows platform, use forward slashes in path names. Also note that the setwd() function will not create a directory that does not exist. If necessary, you can use the dir.create() function to create a directory, and then use setwd() to change to its location.

It is a good idea to keep your projects in separate directories. I typically start an R session by issuing the setwd() command with the appropriate path to a project, followed by the load() command without options. This lets me start up where I left off in my last session and keeps the data and settings separate between projects. On Windows and MacOS X platforms it is even easier. Just navigate to the project directory and double click on the saved image file. This will start R, load the saved workspace, and set the current working directory to this location.

1.3.4 Input and Output

By default, launching R starts an interactive session with input from the keyboard and output to the screen. However, you can also process commands from a script file (a file containing R statements) and direct output to a variety of destinations.

INPUT

The source("filename") function submits a script to the current session. If the filename does not include a path, the file is assumed to be in the current working directory. For example, source("myprog") runs a set of R statements contained in file myprog.

TEXT OUTPUT

The sink("filename") function redirects output to the file filename. By default, if the file already exists, its contents are overwritten. Include the option append=TRUE to append text to the file rather than overwriting it. Including the option split=TRUE will send output to both the screen and the output file. The command sink() by itself, returns output to the terminal.

GRAPHIC OUTPUT

Although sink()redirects text output, it has no effect on graphic output. To redirect graphic output use one of the functions listed in table 1.4. Use dev.off() to return output to the terminal.

Table 1.4 Functions for Saving Graphic Output

Function	Output
pdf("filename.pdf")	pdf file
win.metafile("filename.wmf")	windows metafile
png("filename.pgn")	png file
jpeg("filename.jpg")	jpeg file
bmp("filename.bmp")	bmp file
postscript("filename.ps")	postscript file

Let’s put it all together by looking at the example in listing 1.3

Listing 1.3 Using various input and output in an R session

```
source("myfile1")                                        1

sink(“myoutput”, append=TRUE, split=TRUE)
pdf(“mygraphs.pdf”)
source(“myfile2”)                                        2

sink()
dev.off()
source(“myfile3”)                                        3
```

In the code above, R statements from `myfile1` are submitted to the current session #1 and the results appear on the screen. When the statements from `myfile2` are submitted #2, results appear on the screen, the text output is appended to the file `myoutput`, and the graphic output is saved to `mygraphs.pdf`. Finally the statements from `myfile3` are submitted #3 and the results appear on screen.

R provides quite a bit of flexibility and control over where input comes from and where it goes. In section 1.5 we will see how to run a program in batch mode.

1.4 Packages

R comes with extensive capabilities right out of the box. However, some of its most exciting features are available as optional modules that you can download and install. There are over 1800 user contributed modules called packages that you can download from http://cran.r-project.org/web/packages. They provide a tremendous range of new capabilities, from the analysis of geostatistical data to protein mass spectra processing to the analysis of educational tests! We will use many of these optional packages in this book.

1.4.1 What are packages?

Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored on your computer is called the library. The function `.libPaths()` will show you where your library is located, while the function `library()` will show you what packages you have saved in your library.

As we've said, R comes with a standard set of packages, while others are available for download and installation. Once installed, they have to be loaded into the session in order to be used. The command `search()` will tell you which packages are loaded and ready to use.

1.4.2 Installing a package

There are a number of R functions that let you manipulate packages. To install a package for the first time, use the `install.packages()` command. For example, `install.packages()` without options will bring up a list of CRAN mirror sites. Once you select a site, you will be presented with a list of all available packages. Selecting one will download and install it. If you know what package you want to install, you can do so directly by providing it as an argument to the function. For example, the `gclus` package contains functions for creating enhanced scatter plots. You can download and install the package with the command `install.packages("glus")`.

You only need to install a package once. However, like any software, packages are often updated by their authors. Use the command `update.packages()` to update any packages that you have installed. To see details on your packages, you can use the `installed.packages()` command. It will list the packages you have, along with their version numbers, dependencies, and other information.

1.4.3 Loading a package

Installing a package downloads it from a CRAN mirror site and places it in your library. To actually use it in an R session, you need to load the package using the library() command. For example, to use the packaged `gclus`, issue the command `library(gclus)`. Of course, you must have installed a package before you can load it. You will have to load a package once in each session you want to use it. However, you can customize your start-up environment to automatically load the packages you use most often. Customizing your start-up is covered in appendix x.

1.4.4 Learning about a package

When you load a package, a new set of functions and datasets become available. Small illustrative datasets are provided along with sample code, allowing you to try out the new functionalities. The help system contains a description of each function (along with examples), and information on each dataset included. Entering `help(package="name")` will provide a brief description of the package named and an index of the functions and datasets included. Using `help()`with any of these function or dataset names will provide further details. The same information can be downloaded as a PDF manual from CRAN.

Common mistakes in R programming

There are some common mistakes made frequently by both beginning and experienced R programmers. If your program generates an error be sure the check for the following:

Using the wrong case. `help()`, `Help()`, and `HELP()` are three different functions (only the first will work).

Forgetting to use quote marks when they are needed. `install.packages("gclus")` will work, while `install.packages(gclus)` will generate an error.

Forgetting to include the parentheses in a function call. `help()` rather than `help`. Even if there are no options, you still need the ().

Using the `\` in a path name on Windows. R sees the backslash character as an escape character. `setwd("c:\mydata")` will generate an error. Use `setwd("c:/mydata")` or `setwd("c:\\mydata")` instead.

Using a function from a package that is not loaded. The function `order.clusters()` is contained in the `gclus` package. If you try to use it before loading the package, you will get an error.

The error message in R can be cryptic, but if you are careful to follow the points above, you should avoid seeing many of them.

In the sidebar above, these should be bullet points, with the text of the first sentence bolded for each. Bullets start with "Using the wrong case" and end with "Using a function".

1.5 Batch Processing

Most of the time, you will be running R interactively, entering commands at the command prompt and seeing the results of each statement as it is processed. Occasionally, you may want to run an R program in a repeated, standard, and possibly unattended fashion. For example, you may need to generate the same report once a month. You can write your program in R and run it in batch mode.

How you run R in batch mode depends on your operating system. On Linux or MacOS X systems, you can use the following command in a terminal window:

```
R CMD BATCH options infile outfile
```

where `infile` is the name of the file containing R code to executed, `outfile` is name of the file receiving the output and `options` lists options that control execution. By convention, the infile is given extension `.R` and the outfile is given extension `.Rout`.

For Windows, use

```
"C:\Program Files\R\R-2.9.0\bin\R.exe" CMD BATCH [CA]
   --vanilla --slave "c:\my projects\myscript.R"
```

adjusting the paths to match the location of your R.exe binary and your script file. For more details on how to invoke R, including command line options, see an "Introduction to R" from CRAN (http://cran.r-project.org).

1.6 Using output as input - Reusing results

One of the most useful design features of R is that the output of analyses can easily be saved and used as input to additional analyses. Let's walk through an example. If you don't understand the statistics involved, don't worry. We are focusing on the general principle here.

This following code will run a simple linear regression of miles per gallon (mpg) on car weight (wt) using the dataset mtcars. Results are sent to the screen. Nothing is saved.

```
lm(mpg~wt, data=mtcars)
```

This time, the same regression is performed but the results are saved under the name fit.

```
fit <- lm(mpg~wt, data=mtcars)
```

No output is sent to the screen. However, you now can manipulate the results.

The assignment has actually created a list called "fit" that contains a wide range of information from the analysis (including the predicted values, residuals, regression coefficients, and more). Typing `summary(fit)` provides details of the analysis, while `plot(fit)` produces diagnostic plots. You can generate and save influence statistics with `cook<-cooks.distance(fit)`. `plot(cook)` will graph these influence statistics. To predict miles per gallon from car weight in a new set of data use `predict(fit, mynewdata)`.

To see what a function returns, look at the value section of the online help for that function. Here we would look at `help(lm)`. This will tell you what is saved when you assign the results of that function to a name.

1.7 Working through an example

We will finish this chapter with an example that ties many of these ideas together. Here is the task:

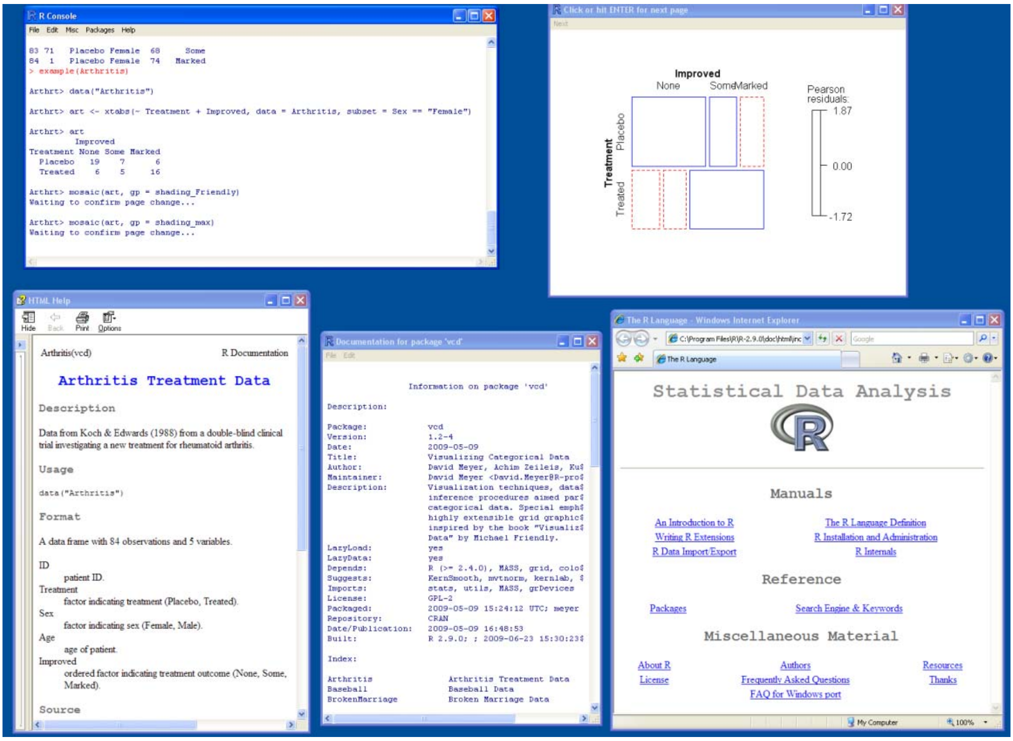
1. Open up the general help and look at the Introduction to R section.
2. Install the vcd package (a package for visualizing categorical data that we will be using in future chapters).
3. List the functions and datasets available in this package.
4. Load the package and read the description of the dataset Arthritis.
5. Print out the Arthritis data set (entering the name of an object will list it).
6. Run the example that comes with the Arthritis dataset. Don't worry if you don't understand the results. It basically shows that arthritis patients receiving treatment improved much more than patients receiving a placebo.
7. Quit

The code required is provided in listing 1.4, with a sample of the results displayed in figure 1.6.

Listing 1.4 Working with a new package

```
help.start() # look at introduction and preliminaries
install.packages("vcd")
```

```
help(package="vcd")
library(vcd)
help(Arthritis)
Arthritis
example(Arthritis)
q()
```



```
83 71   Placebo Female  68     Some
84  1   Placebo Female  74   Marked
> example(Arthritis)

Arthrt> data("Arthritis")

Arthrt> art <- xtabs(~ Treatment + Improved, data = Arthritis, subset = Sex == "Female")

Arthrt> art
         Improved
Treatment None Some Marked
  Placebo   19    7      6
  Treated    6    5     16

Arthrt> mosaic(art, gp = shading_Friendly)
Waiting to confirm page change...

Arthrt> mosaic(art, gp = shading_max)
Waiting to confirm page change...
```

Arthritis(vcd) R Documentation

Arthritis Treatment Data

Description

Data from Koch & Edwards (1988) from a double-blind clinical trial investigating a new treatment for rheumatoid arthritis.

Usage

```
data("Arthritis")
```

Format

A data frame with 84 observations and 5 variables.

ID
: patient ID.

Treatment
: factor indicating treatment (Placebo, Treated).

Sex
: factor indicating sex (Female, Male).

Age
: age of patient.

Improved
: ordered factor indicating treatment outcome (None, Some, Marked).

Source

```
                 Information on package 'vcd'

Description:

Package:             vcd
Version:             1.2-4
Date:                2009-05-09
Title:               Visualizing Categorical Data
Author:              David Meyer, Achim Zeileis, Ku$
Maintainer:          David Meyer <David.Meyer@R-pro$
Description:         Visualization techniques, data$
                     inference procedures aimed par$
                     categorical data. Special emph$
                     highly extensible grid graphic$
                     inspired by the book "Visualiz$
                     Data" by Michael Friendly.
LazyLoad:            yes
LazyData:            yes
Depends:             R (>= 2.4.0), MASS, grid, colo$
Suggests:            KernSmooth, mvtnorm, kernlab, $
Imports:             stats, utils, MASS, grDevices
License:             GPL-2
Packaged:            2009-05-09 15:24:12 UTC; meyer
Repository:          CRAN
Date/Publication:    2009-05-09 16:48:53
Built:               R 2.9.0; ; 2009-06-23 15:30:23$

Index:

Arthritis                Arthritis Treatment Data
Baseball                 Baseball Data
BrokenMarriage           Broken Marriage Data
```

The R Language - Windows Internet Explorer
C:\Program Files\R\R-2.9.0\doc\html\inc
Google
The R Language
My Computer
100%

Statistical Data Analysis

Manuals

An Introduction to R · The R Language Definition · Writing R Extensions · R Installation and Administration · R Data Import/Export · R Internals

Reference

Packages · Search Engine & Keywords

Miscellaneous Material

About R · Authors · Resources · License · Frequently Asked Questions · Thanks · FAQ for Windows port

Figure 1.6 Output from listing 1.4

As this short exercise demonstrates, you can accomplish a great deal with a small amount of code.

1.8 Summary

In this chapter, we have looked as some of the strengths that make R an attractive option for students, researchers, statistician, and data analysts trying to understand the meaning of their data. We have walked through the program's installation and talked about how to enhance R's capabilities by downloading additional packages. We have explored the basic interface, running programs interactively and in batch, and produced a few sample graphs. We have also learned how to save our work to both text and graphic files. Since R can be a complex program, we have spent some time looking at how to access the extensive help that is available. Hopefully, you're getting a sense of how powerful this freely available software can be.

Now that we have R up and running, it's time to get our data into the mix. In the next chapter, we will look at the types of data R can handle and how to import them into R from text files, other programs, and database management systems.

2 *Creating a dataset*

This Chapter covers:

- R data structures
- Data entry
- Importing data
- Annotating datasets

The first step in any data analysis is the creation of a dataset containing the information to be studied, in a format that meets our needs. In R, this will involve

- Selecting a data structure to hold our data
- Entering or importing our data into the data structure

The first part of this chapter (sections 2.1-2.2) describes the wealth of structures that R can use for holding data. In particular, section 2.2 describes scalars, vectors, matrices, dataframes, factors, and lists. Understanding these structures (and the notation used to access elements within them) will be help tremendously in understanding how R works. You might want to take your time working through this section.

The second part of this chapter (section 2.3) covers the many methods available for importing data into R. Data can be entered manually, or imported from an external source. These data sources can include text files, spreadsheets, statistical packages, and database management systems. For example, the data that I work with typically come from SQL databases. However, on occasion, I receive data from legacy DOS systems, and from current SAS and SPSS databases. It is likely that you will only have to use one or two of the methods described in this section, so feel free to pick and choose those that fit for your situation.

Once a dataset is created, we will typically annotate it, adding descriptive labels for variables and variable codes. The third portion of this chapter will look at annotating datasets (2.4) and reviews some useful functions for working with datasets (2.5). Let's start with the basics.

2.1 Understanding datasets

A dataset is usually a rectangular array of data with rows representing observations and columns representing variables. An example of a hypothetical patient dataset is given in table 2.1

Table 2.1 A patient dataset

PatientID	AdmDate	Age	Diabetes	Status
1	10/15/2009	25	Type1	Poor
2	11/01/2009	34	Type2	Improved
3	10/21/2009	28	Type1	Excellent
4	10/28/2009	52	Type1	Poor

Different traditions have different names for the rows and columns of a dataset. Statisticians refer to them as observations and variables, database analysts call them records and fields, and those from the data mining/machine learning disciplines call them examples and attributes. We will use the terms observations and variables throughout the rest of this book.

We can distinguish between the structure of the dataset (in this case a rectangular array) and the contents or data types included. In the dataset above, `PatientID` is a row or case identifier, `AdmDate` is a date variable, `Age` is a continuous variable, `Diabetes` is nominal variable, and `Status` is an ordinal variable.

R contains a wide variety of structures for holding data including scalars, vectors, arrays, dataframes, and lists. The table above corresponds to a dataframe in R. This diversity of structures provides the R language with a great deal of flexibility in dealing with data.

The data types or modes that R can handle include numeric, character, logical (TRUE/FALSE), complex (imaginary numbers), and raw (bytes). In R, `PatientID`, `AdmDate`, and `Age` would be numeric variables, while `Diabetes` and `Status` would be character variables. Additionally, we will need to tell R that `PatientID` is a case identifier, `AdmDate` contains dates, and that `Diabetes` and `Status` are nominal and ordinal variables, respectively. R refers to case identifiers as `rownames` and categorical variables (nominal, ordinal) as `factors`. We will cover each of these in the next section. Dates will be discussed in chapter 3.

2.2 Data structures

As we have said, R has a wide variety of objects for holding data, including scalars, vectors, matrices, dataframes and lists. They differ in terms of the type of data they can hold, how they are created, their structural complexity, and the notation used to identify and access individual elements. Figure 2.1 presents a diagram of these data structures.

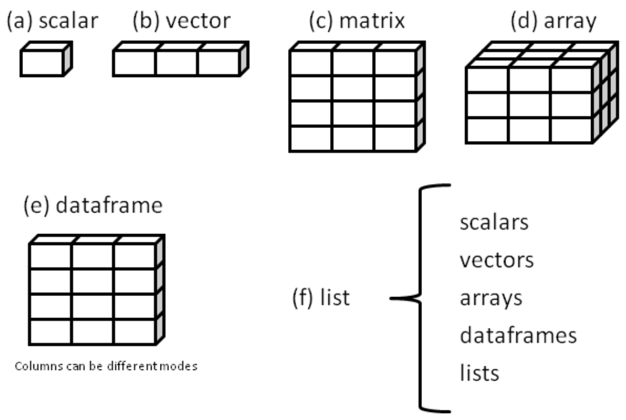


Figure 2.1 R data structures

We will look at each structure in turn, starting with vectors.

Some Definitions

There are several terms that are idiosyncratic to R, and thus confusing to new users.

In R, an **object** is anything that can be assigned to a variable. This includes constants, data structures, functions, and even other objects. Objects have a mode (which describes how the object is stored), and a class (which tells generic functions like print, how to handle it).

A **dataframe** is a structure in R that holds data and similar to the datasets found in standard statistical packages (e.g., SAS, SPSS, and Stata). The columns are variables and the rows are observations. We can have variables of different types (e.g., numeric, character) in the same dataframe. Dataframes are the main structures we will use to store datasets.

Factors are nominal or ordinal variables. They are stored and treated specially in R. We will have much to say about factors in section 2.2.5.

Most other terms should be familiar and follow the terminology used in statistics and computing in general.

2.2.1 Vectors

Vectors are one dimensional arrays that can hold numeric data, character data, or logical data. The combine function `c()` is used to form the vector (see listing 2.1).

Listing 2.1 Creating vectors

```
# a numeric vector
a <- c(1, 2, 5, 3, 6, -2, 4)

# a character vector
b <- c("one", "two", "three")

# a logic vector
c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)
```

Note that the data in a vector must only be one type or mode (numeric, character, or logical). You cannot mix modes in the same vector.

SCALARS

Scalars are simply one element vectors. Examples include `f <- 3`, `g <- "US"` and `h <- TRUE`. They are used to hold constants.

You can refer to elements of a vector using a numeric vector of positions within brackets. For example `a[c(2, 4)]` refer to the 2nd and 4th element of vector `a`. The following code provides examples (listing 2.2).

Listing 2.2 Using vector subscripts

```
> a <- c(1, 2, 5, 3, 6, -2, 4)
> a[3]

[1] 5

> a[c(1, 3, 5)]

[1] 1 5 6

> a[2:6]                              1

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

#1 In this example we use the `colon` operator, which takes the form `from:to`. For example:

```
a <- c(2:6)

is equivalent to

a <- c(2,3,4,5,6).
```

2.2.2 Matrices

A matrix is a two dimensional array where each element has the same mode (numeric, character, or logical). Matrices are created with the `matrix` function. The general format is

```
myymatrix <- matrix(vector, nrow=r, ncol=c, byrow=logical_value,
[CA] dimnames=list(char_vector_rownames, char_vector_colnames))
```

where `vector` contains the elements for the matrix, `r` and `c` give the row and column dimensions, and `dimnames` contains optional row and column labels. The option `byrow` indicates whether the matrix should be filled in by row (`byrow=TRUE`) or by column (`byrow=FALSE`). The default is by column. Listing 2.3 demonstrates the `matrix` function.

Listing 2.3 Creating Matrices

```
> # create 5 x 4 matrix
> y <- matrix(1:20, nrow=5, ncol=4)
> y
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20

> # create a 2 x 2 matrix with labels
> # fill in the matrix by rows
> cells    <- c(1,26,24,68)
> rnames   <- c("R1", "R2")
> cnames   <- c("C1", "C2")
> mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=TRUE,[CA]
    dimnames=list(rnames, cnames))
> mymatrix
   C1 C2
R1  1 26
R2 24 68

> # this time fill in the matrix by columns
> mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=FALSE, [CA]
    dimnames=list(rnames, cnames))
```

```
> mymatrix
   C1 C2
R1  1 24
R2 26 68
```

You can identify rows, columns or elements of a matrix by using subscripts and brackets. `X[i,]` refers to the ith row of matrix X, while `X[,j]` refers to jth column, and `X[i,j]` refers to the ijth element respectively. The subscripts i and j can be numeric vectors in order to select multiple rows or columns. Examples are given in listing 2.4.

Listing 2.4 Using matrix subscripts

```
> x <- matrix(1:10, nrow=2)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

> # selecting the 2nd row
> x[2,]
[1]  2  4  6  8 10

> # selecting the 2nd column
> x[,2]
[1] 3 4

> # selecting the 1st row, 4th column element
> x[1,4]
[1] 7

> # selecting the first row, 4 & 5th columns
> x[1, c(4,5)]
[1] 7 9
```

Matrices are two dimensional and, like vectors, can contain only one data type. When there are more than two dimensions, we will use arrays (section 2.2.3). When there are multiple modes of data, we will use dataframes (section 2.2.4).

2.2.3 Arrays

Arrays are similar to matrices but can have more than two dimensions. They are created with an array function of the following form:

```
myarray <- array(vector, dimensions, dimnames)
```

where `vector` contains the data for the array, `dimensions` is a numeric vector giving the maximal index for each dimension, and `dimnames` is an optional list of dimension labels. Listing 2.5 gives an example of creating a three dimensional (2 x 3 x 4) array of numbers.

Listing 2.5 Creating an array

```
> dim1 <- c("A1", "A2")
> dim2 <- c("B1", "B2", "B3")
> dim3 <- c("C1", "C2", "C3", "C4")
> z <- array(1:24, c(2,3,4), dimnames=list(dim1,dim2,dim3))
> z
, , C1

   B1 B2 B3
A1  1  3  5
A2  2  4  6

, , C2

   B1 B2 B3
A1  7  9 11
A2  8 10 12

, , C3

   B1 B2 B3
A1 13 15 17
A2 14 16 18

, , C4

   B1 B2 B3
A1 19 21 23
A2 20 22 24
```

As you can see, arrays are a natural extension of matrices. They can be very useful in programming new statistical methods. Like matrices, they must be a single mode. Identifying elements follows what we have seen for matrices. In the example above, the z[1,2,3] element is 15.

2.2.4 Dataframes

A dataframe is more general than a matrix, in that different columns can contain different modes of data (numeric, character, etc.). It is similar to the datasets you would typically see in SAS, SPSS, and Stata. Dataframes are the most common data structure we will deal with in R.

The patient dataset in table 2.1 consists of numeric and character data. Because there are multiple modes of data, we cannot contain this data in matrix. In this case, a dataframe would be the structure of choice.

A dataframe is created with the `dataframe` function:

```
mydata <- data.frame(col1, col2, col3,…)
```

where col1, col2, col3, … are column vectors of any type (character, numeric, logical, etc.). Names for each column can be provided with the names function. An example (listing 2.6) will make this clear.

Listing 2.6 Creating a dataframe

```
> patientID <- c(1, 2, 3, 4)
> age <- c(25, 34, 28, 52)
> diabetes <- c("Type1", "Type2", "Type1", "Type1")
> status <- c("Poor", "Improved", "Excellent", "Poor")
> patientdata <- data.frame(patientID, age, diabetes, status)
> patientdata

  patientID age diabetes    status
1         1  25    Type1      Poor
2         2  34    Type2  Improved
3         3  28    Type1 Excellent
4         4  52    Type1      Poor
```

Each column must have only one mode. However, you can put columns of different modes together to form the dataframe. Since dataframes are very close to what analysts typically think of as datasets, we will use the terms columns and variables interchangeably when discussing dataframes.

There are several ways to identify the elements of a dataframe. You can use the subscript notation we have used previously (e.g. with matrices) or you can specify column names. Take a look at the following three examples in listing 2.7.

Listing 2.7 Specifying elements of a dataframe

```
> # continuing the last example
> patientdata[1:2]

  patientID age
1         1  25
2         2  34
3         3  28
4         4  52

> patientdata[c("diabetes","status")]

  diabetes    status
1    Type1      Poor
2    Type2  Improved
3    Type1 Excellent
4    Type1      Poor

> patientdata$age                       1

[1] 25 34 28 52
```

#1 The $ notation in the third example is new. It is used to indicate a particular variable from a given dataframe. For example, if you want to get descriptive statistics on the variables `age`, `diabetes`, and `status` from the `patientdata` dataframe, you could use the following code:

```
summary(patientdata$age, patientdata$diabetes, patientdata$status)
```

This can get tiresome to type, so a shortcut available.

```
attach(patientdata)
summary(age, diabetes, status)
```

The `attach` function adds the dataframe to the R search path. When a variable name is encountered, dataframes in the search path are checked in order to locate the variable. We will use the attach function often.

CASE IDENTIFIERS

In the patient data example, `patientID` is used to identify individuals in the dataset. In R, case identifiers can be specified with a `rowname` option in the dataframe function. For example, the statement

```
patientdata <- data.frame(patientID, age, diabetes, status,
[CA]rownames=patientID)
```

specifies patientID as the variable to use in labeling cases on various printouts and graphs produced by R.

2.2.5 Factors

As we have seen, variables can be described as nominal, ordinal, or continuous. Nominal variables are categorical, without an implied order. Diabetes (Type1, Type2) is an example of a nominal variable. Even if Type1 is coded as a 1 and Type2 is coded as a 2 in the data, no order is implied. Ordinal variables imply order but not amount. Status (poor, improved, excellent) is a good example of an ordinal variable. We know that a patient with a "poor" status is not doing as well as a patient with an "improved" status, but not by how much. Continuous variables can take on any value within some range and both order and amount is implied. Age in years is a continuous variable and can take on values such as 14.5 or 22.8 and any value in between. We know that someone who is fifteen is one year older than someone who is fourteen.

Many R functions will handle data differently if one or more variables are nominal or ordinal rather than continuous. Categorical (nominal) and ordered categorical (ordinal) variables in R are called factors. The function `factor` stores the categorical values as a vector of integers in the range [1... k] (where k is the number of unique values in the nominal variable), and an internal vector of character strings (the original values) mapped to these integers.

For example, assume that we have the vector

```
diabetes <- c("Type1", "Type2", "Type1", "Type1")
```

The statement `diabetes <- factor(diabetes)` stores this vector as (1, 2, 1, 1) and associates it with 1=Type1 and 2=Type2 internally (the assignment is alphabetical). Any analyses performed on the vector `diabetes` will treat the variable as nominal and select the statistical methods appropriate for this level of measurement.

For vectors representing ordinal variables, we add the parameter `ordered=TRUE` to the `factor` function. For the vector

```
status <- c("Poor", "Improved", "Excellent", "Poor")
```

The statement `status <- factor(status, ordered=TRUE)` will encode the vector as (3, 2, 1, 3) and associate these values internally as 1=Excellent, 2=Improved, and 3=Poor. Additionally, any analyses performed on this vector will treat the variable as ordinal and select the statistical methods appropriately. For compatibility with the S language, the statement above could also have been written as `status <- ordered(status)`.

Listing 2.8 demonstrates how specifying factors and ordered factors impact data analyses.

Listing 2.8 Using factors

```
> # enter the variables as vectors
> patientID <- c(1, 2, 3, 4)
> age <- c(25, 34, 28, 52)
> diabetes <- c("Type1", "Type2", "Type1", "Type1")
> status <- c("Poor", "Improved", "Excellent", "Poor")

> # specify the vectors as factors
> diabetes <- factor(diabetes)
> status <- factor(status, order=TRUE)

> # create the dataframe
> patientdata <- data.frame(patientID, age, diabetes, status)

> # view the structure of the dataframe
> str(patientdata)

'data.frame':   4 obs. of  4 variables:                                 1
 $ patientID: num  1 2 3 4
 $ age      : num  25 34 28 52
 $ diabetes : Factor w/ 2 levels "Type1","Type2": 1 2 1 1
 $ status   : Ord.factor w/ 3 levels "Excellent"<"Improved"<..: 3 2 1 3

> # get summary statistics on the variables                              2
> summary(patientdata)

   patientID         age          diabetes       status
 Min.   :1.00   Min.   :25.00   Type1:3   Excellent:1
 1st Qu.:1.75   1st Qu.:27.25   Type2:1   Improved :1
 Median :2.50   Median :31.00             Poor     :2
 Mean   :2.50   Mean   :34.75
 3rd Qu.:3.25   3rd Qu.:38.50
 Max.   :4.00   Max.   :52.00
```

The function `str(object)` #1 provides information on an object in R (the dataframe in this case). It clearly shows that `diabetes` is a factor and `status` in an order factor, along with how it is coded internally. Note that the summary function #2 treats the variables differently. It provides the minimum, maximum, mean, and quartiles for the continuous variable age, and frequency counts for the categorical variables diabetes and status.

2.2.6 Lists

Lists are the most complex of the R data types. Basically, a list is an ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name. For example, a list may contain a combination of vectors, matrices, dataframes, and even other lists. A list is created with the `list` function:

```
mylist <- list(object1, object2, …)
```

where the objects are any of the structures we have seen so far. Optionally, you can name the objects in a list.

```
mylist <- list(name1=object1, name2=object2, …)
```

An example is given in listing 2.9.

Listing 2.9 Creating a list

```
# Example of a list with 4 components -
# a string, a numeric vector, a matrix, and character vector

> g <- "My First List"
> h <- c(25, 26, 18, 39)
> j <- matrix(1:10, nrow=5)
> k <- c("one", "two", "three")
> mylist <- list(title=g, ages=h, j, k)

> print the contents
> mylist

$title
[1] "My First List"

$ages
[1] 25 26 18 39

[[3]]
     [,1] [,2]
[1,]    1    6
[2,]    2    7
```

```
[3,]    3    8
[4,]    4    9
[5,]    5   10

> mylist[[2]]
[1] 25 26 18 39

> mylist[["age"]]
[[1] 25 26 18 39
```

This simple example shows that any number of objects can be combined and saved as a list. You can specify elements of the list by specifying a component number or a name within double brackets. In this example, `mylist[[2]]` and `mylist[["ages"]]` both refer to the same 4 element numeric vector. Lists are very important R structures for two reasons. First, they allow you to organize and recall disparate information in a simple way. Second, the results of many R functions return lists. It is up to the analyst to pull out the components that are needed. We will see numerous examples of this in later chapters.

2.3 Data input

Now that we have data structures, we need to put some data in them! As data analysts, we are typically faced with data that comes to us from a variety of sources and in a variety of formats. Our task is to import the data into our tools, analyze the data, and report on the results. R provides a wide range of tools for importing data.

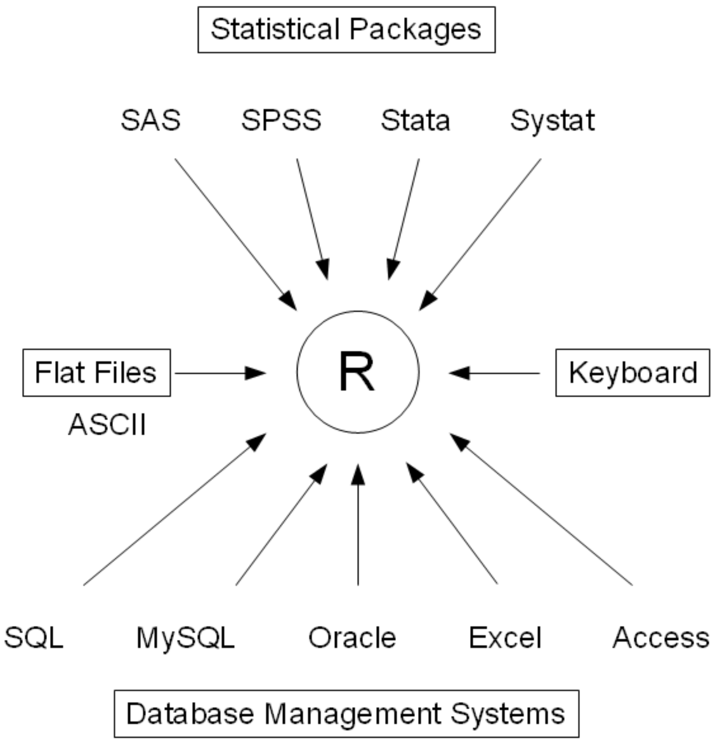


Figure 2.2 Sources of data that can be imported into R

As you can see from figure 2.2, R can import data from the keyboard, from flat files, from Microsoft products such as Excel and Access, from popular statistical packages, and from a variety of relational database management systems. Since we never know where our data will come from next, we will cover all of them here.

2.3.1 Entering data from the keyboard

Perhaps the simplest method of data entry is from the keyboard. The `edit` function in R will invoke a text editor that will allow us to enter our data manually. The steps are:

8. Create an empty dataframe (or matrix) with the variable names and modes you want to have in the final dataset.
9. Invoke the text editor on this data object, enter your data, and save the results back to the data object.

In the following example, we will create a dataframe named `mydata` with three variables: `age` (numeric), `gender` (character), and `weight` (numeric). We will then invoke the text editor, add our data, and save the results (see listing 2.10).

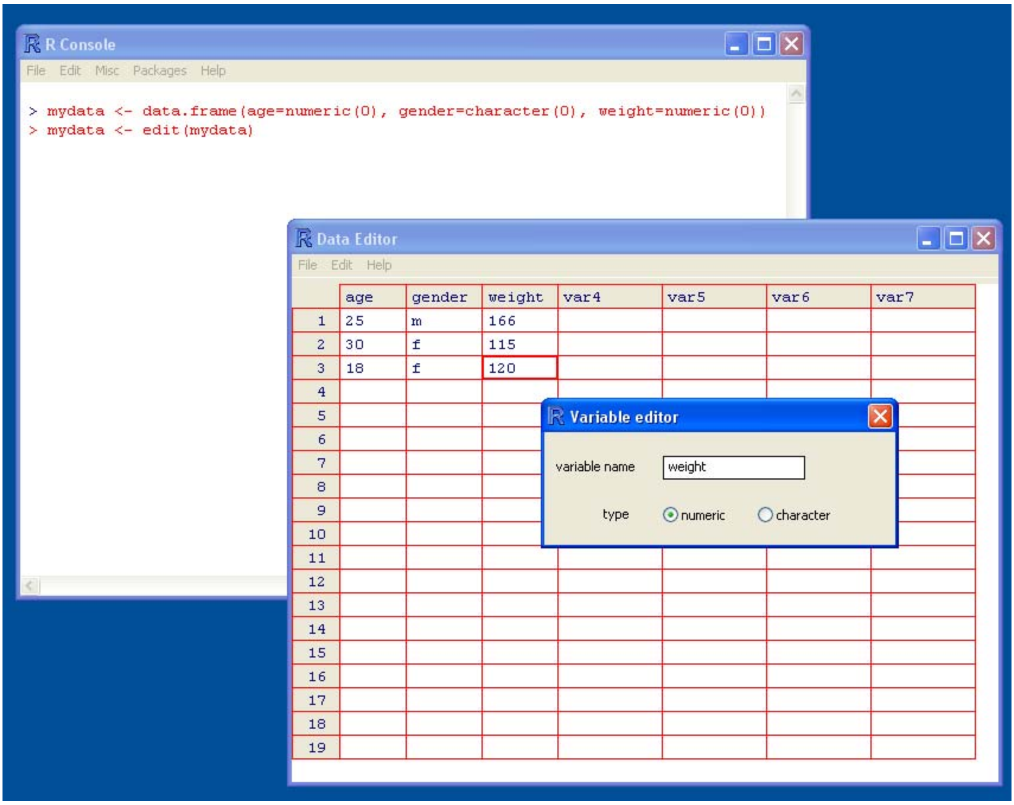
Listing 2.10 Entering data in R via text editor

```
mydata <- data.frame(age=numeric(0), [CA]        1
  gender=character(0), weight=numeric(0))

mydata <- edit(mydata)                           2
```

#1 Assignments like age=numeric(0) create a variable of a specific mode, but without actual data. #2 Note that the result of the editing is assigned back to the object itself. The edit function actually operates on a copy of the object. If you do not assign it a destination, all of your edits will be lost!

On a Windows platform, the results of invoking the edit function can be seen in Figure 2.3.



```
> mydata <- data.frame(age=numeric(0), gender=character(0), weight=numeric(0))
> mydata <- edit(mydata)
```

	age	gender	weight	var4	var5	var6	var7
1	25	m	166				
2	30	f	115				
3	18	f	120				
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							

variable name: weight
type: [✓ numeric] character

Figure 2.3 Entering data via text editor on a Windows platform.

In this figure, I've taken the liberty of adding some data. If we click on a column title, the editor gives us the option of changing the variable name and type (numeric, character). We can add additional variables by clicking on the titles of unused columns. When the text editor is closed, the results are saved to the object assigned (`mydata` in this case). Invoking `mydata <- edit(mydata)` again allows us to edit the data we have have entered and to add new data. A shortcut for `mydata <- edit(mydata)` is simply `fix(mydata)`.

This method of data entry works well for small datasets. For larger datasets, you will probably want to use one of the methods we will describe next - namely importing data from existing text files, Excel spreadsheets, statistical packages, or database management systems.

2.3.2 Importing data from a (comma) delimited text file

We can import data from comma delimited text files using the `read.table`, a function that reads a file in table format and saves it as a dataframe. The syntax is

```
mydataframe <- read.table(file, header = logical_value,
sep="delimiter", row.names = "name")
```

where `file` is a delimited ASCII file, `header` is a logical value indicating whether the first row contains variable names (TRUE) or not (FALSE), `sep` specifies the delimiter separating data values, and `row.names` is an option parameter specifying one or more variables to represent row identifiers.

For example, the statement

```
grades <- read.table("studentgrades.csv", header=TRUE, sep=",",
row.names="STUDENTID")
```

reads a comma delimited file named `studentgrades.csv` from the current working directory, gets the variable names from the first line of the file, specifies the variable `STUDENTID` as the row identifier, and saves the results as a dataframe named `grades`.

Note that the `sep` parameter allows us to import files that use a symbol other than a comma to delimit the data values. To import a tab delimited file, you could use `sep=""` which denotes whitespace (one or more spaces, tabs, new lines, or carriage returns). The `read.table` function has many additional options for fine tuning the data import. See `help(read.table)` for details.

2.3.3 Importing data from Excel

The best way to read an Excel file is to export it to a comma delimited file from within Excel and import it to R using the method above. On Windows systems you can also use the RODBC package to access Excel files. The first row of the spreadsheet should contain variable/column names.

First, download and install the RODBC package

```
install.packages("RODBC")
```

You can then use the following code to import the data.

```
library(RODBC)
channel <- odbcConnectExcel("myfile.xls")
mydataframe <- sqlFetch(channel, "mysheet")
odbcClose(channel)
```

Here, `myfile.xls` is an Excel file, `mysheet` is the name of the Excel worksheet to read from the workbook, channel is an RODBC connection object returned by obcConnect, and `mydataframe` is the resulting dataframe. RODBC can also be used to import data from Microsoft Access. See `help(RODBC)` for details.

2.3.4 Importing data from SPSS

SPSS datasets can be imported into R by the `read.spss` function in the `foreign` package. However, we will be using the `spss.get` function in the Hmisc package instead. spss.get is a wrapper function that sets many parameters of `read.spss` for us automatically and makes the transfer easier and more consistent with what data analysts expect as a result.

First, download and install the `Hmisc` package (the foreign package is already installed by default).

```
install.packages("Hmisc")
```

Then use the following code to import the data.

```
library(Hmisc)
mydataframe <- spss.get("mydata.sav", use.value.labels=TRUE)
```

In the code above, mydata.sav is the SPSS datafile to be imported, `use.value.labels=TRUE` tells the function to convert variables with value labels into R factors with those same levels, and `mydataframe` is the resulting R dataframe.

2.3.5 Importing data from SAS

There are a number of functions in R designed to import SAS datasets, including `read.ssd` in the `foreign` package and `sas.get` in the `Hmisc` package. Unfortunately, if you are using a recent version of SAS (say SAS 9.1 or higher), you are likely to find that these functions do not work for you because R has not caught up with changes in SAS file structures. There are two solutions that I would recommend.

You can save the SAS dataset as a comma delimited text file from within SAS using PROC EXPORT, and read the resulting file into R using the method described in section 2.3.1. An example is given in listing 2.11.

Listing 2.11 Exporting a SAS dataset to an R dataframe

```
SAS program:

proc export data= mydata
     outfile= "mydata.csv"
     dbms=csv;
run;

R program:

mydata <- read.table("mydata.csv", header=TRUE, sep=",")
```

Alternatively, there is a commercial product call Stat Transfer (described in section 2.3.9) that does an excellent job of saving SAS datasets (including any existing variable formats) as R dataframes.

2.3.6 Importing data from Stata

Importing data from Stata to R is straightforward. The necessary code is

```
library(foreign)
mydataframe <- read.dta("mydata.dta")
```

Here, `mydata.dta` is the Stata dataset and `mydataframe` is the resulting R dataframe.

2.3.7 Importing data from Systat

Similar to Stata, the code to import Systat data in R is simple.

```
library(foreign)
mydataframe <- read.systat("mydata.syd")
```

Again, `mydata.syd` is the Systat dataset and `mydataframe` is the resulting R dataframe.

2.3.8 Accessing Database Management Systems (DBMS)

There are a number of R packages that provide access to relational database management systems including MS SQL, Oracle, and MySQL.

THE ODBC INTERFACE

The RODBC package provides access to databases (including Microsoft Access and Microsoft SQL Server) through an ODBC interface. If you have not previously installed the RODBC package, you can do so with the `install.packages("RODBC")` command. The primary functions included with this package are listed in table 2.2.

Table 2.2 RODBC functions

Function	**Description**
`odbcConnect(dsn, uid="", pwd="")`	Open a connection to an ODBC database
`sqlFetch(channel, sqtable)`	Read a table from an ODBC database into a dataframe
`sqlQuery(channel, query)`	Submit a query to an ODBC database and return the results
`sqlSave(channel, mydf, tablename = sqtable, append = FALSE)`	Write or update (`append=TRUE`) a dataframe to a table in the ODBC database
`sqlDrop(channel, sqtable)`	Remove a table from the ODBC database
`close(channel)`	Close the connection

The RODBC package allows two-way communication between R and an ODBC connected SQL database. This means that you can not only read data from a connected the database into R, but you can use R to alter the contents of the database itself. In the following example (listing 2.12) we will import two tables (Crime and Punishment) from a DBMS into two R dataframes and called crimedat and pundat, respectively.

Listing 2.12 Accessing a DBMS through an ODBC interface

```
library(RODBC)                                                1
myconn <-odbcConnect("mydsn", uid="Rob", pwd="aardvark")      2
crimedat <- sqlFetch(myconn, Crime)                           3
pundat <- sqlQuery(myconn, "select * from Punishment")        4
close(myconn)                                                 5
```

After loading the RODBC package #1, we open a connection to the ODBC database #2 through a registered data source name (`mydsn`) with a security UID (`rob`) and password (`aardvark`). The connection string is passed to `sqlFetch` #3, which copies the table `Crime` into the R dataframe `crimedat`. In #4 we run the SQL `select` statement against table `Punishment` and save the results to the dataframe `pundat`. Finally, we close the connect #5.

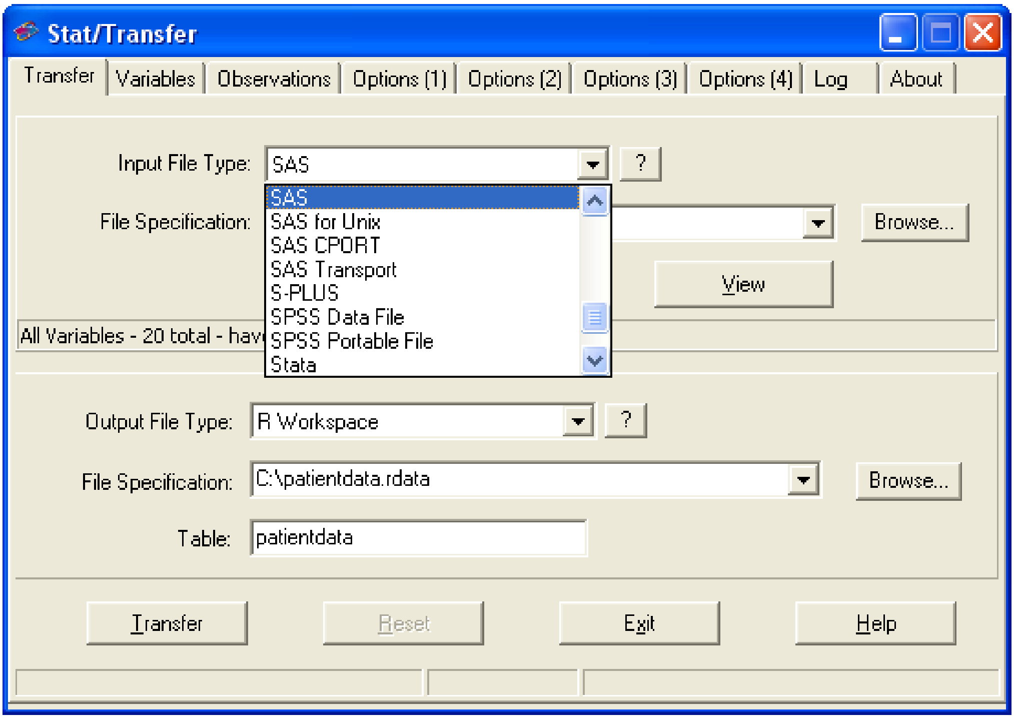
The `sqlQuery` function is very powerful because any valid SQL statement can be inserted. This allows us to select specific variables, subset the data, create new variables, and recode and rename existing variables.

OTHER INTERFACES

R provides other interfaces to DBMS. The RMySQL package provides an interface to MySQL, the `ROracle` package provides an interface to Oracle, and the RJDBC package provides access to databases through a JDBC interface. Documentation for each package is available on CRAN (http://cran.r-project.org). With variations, they are similar to the RODBC package we have just seen.

2.3.9 Importing data via Stat/Transfer

Before ending our discussion of data importing, it is worth mentioning a commercial product that can make the task significantly easier. Stat/Transfer (www.stattransfer.com) is a stand-alone application that can transfer data between 34 data formats, including R (see figure 2.4)



Input File Type: SAS

File Specification:

Output File Type: R Workspace

File Specification: C:\patientdata.rdata

Table: patientdata

Figure 2.4 Stat/Transfer main dialog on Windows.

It is available for Windows, Mac, and UNIX platforms and supports the latest versions of the statistical packages we have discussed so far, as well as ODBC accessed DBMS such as Oracle, Sybase, Informix, and DB/2.

2.4 Annotating datasets

Data analysts typically annotate datasets to make the results easier to interpret. Typically annotation includes adding descriptive labels to variable names and value labels to the codes used for categorical variables. For example, for the variable age, we might want to attach the more descriptive label "Age at hospitalization (in years)". For a new variable gender code 1 or 2, we might want to associate the labels "male" and "female".

2.4.1 Variable labels

Unfortunately, R's ability to handle variable labels is limited. One approach is to use the variable label as the variable's name and then refer to the variable by its position index. Using the example above, let's say that we have a dataframe containing patient data. The third column, named age, contains the ages at which individuals were first hospitalized. The code

```
names(patientdata)[3] <- "Age at hospitalization (in years)"
```

renames `age` to `"Age at hospitalization (in years)"`. Clearly this new name is too long to type repeatedly. Instead, we can refer to this variable as `patientdata[3]` and the string `"Age at hospitalization (in years)"` will print wherever `age` would have originally. Obviously, this is not an ideal approach, and you may be better off simply trying to come up with better names (e.g. admissionAge).

2.4.2 Value labels

The `factor` function can be used to create value labels for categorical variables. Continuing the example above, we could use the code

```
patientdata$gender <- factor(patientdata$gender,[CA]
levels = c(1,2),
labels = c("male", "female"))
```

Here `levels` indicate the actual values of the variable, and `labels` refer to a character vector containing the desired labels.

2.5 Useful functions for working with data objects

We will end this chapter with a brief summary of useful functions for working with data objects (see table 2.3).

Table 2.3 Useful functions for working with data objects

Function	Action
`length(object)`	number of elements/components
`dim(object)`	dimensions of an object
`str(object)`	structure of an object
`class(object)`	class or type of an object
`mode(object)`	how an object is stored
`names(object)`	names of components in an object
`c(object,object,...)`	combines objects into a vector
`cbind(object, object, ...)`	combines objects as columns
`rbind(object, object, ...)`	combines objects as rows
`object`	prints the object

`head(object)`	list the first part the object
`tail(object)`	list the last part of the object
`ls()`	list current objects
`rm(object)`	delete an object
`newobject <- edit(object)`	edit object and save as newobject
`fix(object)`	edit in place

We have already discussed most of these functions. The functions `head` and `tail` are useful for quickly scanning large datasets. For example, `head(patientdata)` lists the first six rows of our dataframe, while `tail(patientdata)` lists the last six. We will cover functions such as length, cbind, and rbind, in the next chapter. They are gathered here as a reference.

2.6 Summary

One of the most challenging tasks in data analysis is data preparation. We have made a good start in this chapter by outlining the various structures that R provides for holding data and the many methods available for importing data from both keyboard and external sources. In particular, we will use the definitions of mode, vector, matrix, dataframe, and list again and again in later chapters. Our ability to specify elements of these structures via the bracket notation will be particularly important in selecting, subsetting, and transforming data.

Once we get our datasets into R, it is likely that we will have to manipulate them into a more conducive format (I find guilt works well). In the next chapter, we will explore ways of creating new variables, transforming and recoding existing variables, merging datasets, and selecting observations.

3 Basic data management

This Chapter covers:

- Manipulating dates and missing values
- Data type conversions
- Creating and recoding variables
- Sorting, merging, and subsetting datasets
- Selecting and dropping variables

In the last chapter, we covered a variety of methods of importing data into R. Unfortunately, getting our data in the rectangular arrangement of a matrix or dataframe is just the first step in preparing it for analysis. To paraphrase Captain Kirk in "A Taste of Armageddon" (and proving my geekiness once and for all) "Data is a messy business - a very, very messy business." In my own work, as much as 60% of the time I spend on data analysis is actually spent preparing the data for analysis. I will go out a limb and say that this is probably true in one form or another for most real-world data analysts. Let's take a look at an example.

3.1 A Working Example

One of the topics that I study in my current job is how men and women differ in the ways that they lead their organizations. Typical questions might be:

- Do men and women in management positions differ in the degree to which they defer to superiors?
- Does this vary from country to country, or are these gender differences universal?

One way to address these questions is to have bosses in multiple countries rate their managers on deferential behavior, using questions like the one below.

This manager asks my opinion before making personnel decisions.

1	2	3	4	5
strongly disagree	disagree	neither agree nor disagree	agree	strongly agree

The numbers and anchor text in the cells above should be centered.

The resulting data might resemble those in table 3.1. Each row represents the ratings a manager by his or her boss.

Table 3.1 Gender differences in leadership behavior

manager	date	country	gender	age	q1	q2	q3	q4	q5
1	10/24/08	US	M	32	5	4	5	5	5
2	10/28/08	US	F	45	3	5	2	5	5
3	10/01/08	UK	F	25	3	5	5	5	2
4	10/12/08	UK	M	39	3	3	4		
5	05/01/09	UK	F	99	2	2	1	2	1

Here, each manager is rated by their boss on five statements (q1 to q5) related to deference to authority. For example, Manager 1 is a 32 year old male working in the US and is rated very deferential by his boss, while manager 5 is a female of unknown age (99 probably indicates missing) working in the UK and is rated by low on deferential behavior. The date column captures when the ratings were made. Although a dataset might have dozens of variables and thousands of observations, we have only included 10 columns and 5 rows to simplify our examples. Additionally, we have limited the number of items pertaining to the managers' deferential behavior to five. In a real-world study, we would probably use 10-20 such items to improve the reliability and validity of the results.

In attempting to address the questions of interest, there are several data management issues to be addressed. Here is a partial list:

- The five ratings (q1 to q5) will need to be combined, yielding a single mean deferential score from each manager.
- In surveys, respondents often skip questions. For example, the boss rating manager 4 skipped questions 4 and 5. We will need a method of handling incomplete data. We will also need to recode values like 99 for age to missing.

- There may be hundreds of variables in a dataset, but we may only be interested in a few. To simplify matters, we will want create a new dataset with only the variables of interest.
- Past research suggests that leadership behavior may change as a function of the manager's age. To examine this, we may want to recode the current values of age into a new categorical age grouping (e.g., young, middle aged, elder).
- Leadership behavior may change over time. We might want to focus on deferential behavior during the recent global financial crisis. To do this, we may want to limit the study to data gathered during a specific period of time (say January 1, 2009 to December 31, 2009).

We will work through each of these issues in the current chapter, as well other basic data management issues such as combining and sorting datasets. Then in chapter 4 we will look at some advanced topics. 3.2 Creating new variables

In a typical research project, we will need to create new variables and transform existing ones. We will use statements of the form

```
variable <- expression
```

A wide array of operators and functions can be included in the `expression` portion of the statement. Table 3.2 lists R's arithmetic operators. We will use arithmetic operators when developing formulas.

Table 3.2 Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^ or **	Exponentiation
x%%y	Modulus (x mod y) 5%%2 is 1
x%/%y	Integer division 5%/%2 is 2

Let's say that we have a dataframe named mydata, with variables x1 and x2, and we want to create a new variable `sumx` that adds these two variables and new variable called `meanx` that averages the two variables. If we use the following code

```
sumx <- x1 + x2
meanx <- (x1 + x2)/2
```

we will get an error, because R does not know that `x1` and `x2` are from dataframe `mydata`. If we use the code below instead

```
sumx <- mydata$x1 + mydata$x2
meanx <- (mydata$x1 + mydata$x2)/2
```

the statements will succeed but we will end up with a dataframe (`mydata`), and two separate vectors(`sumx` and `meanx`) . This is probably not what we want. Ultimately, we want to incorporate new variables into the original data frame. Listing 3.1 provides three separate ways to accomplish this. The one you choose is up to you - the results will be the same.

Listing 3.1 Creating new variables

```
# Three examples for doing the same computations

mydata$,sumx <- mydata$x1 + mydata$x2
mydata$meanx <- (mydata$x1 + mydata$x2)/2

attach(mydata)
mydata$sumx <- x1 + x2
mydata$meanx <- (x1 + x2)/2
detach(mydata)

mydata <- transform(mydata,
sumx = x1 + x2,
meanx = (x1 + x2)/2
)
```

Personally, I prefer the third method, exemplified by use of the `transform` function. It simplifies inclusion of as many new variables as desired and saves the results to the dataframe.

3.3 Recoding variables

Recoding involves creating new values of a variable conditional on the existing values of the same and/or other variables. For example, we may want to:

- change a continuous variable into a set of categories
- replace miscoded values with correct values
- create a pass/fail variable based on a set of cut-off scores.

In order to recode data, we can use one or more of R's logical operators (see table 3.3). Logical operators are expressions that return TRUE or FALSE.

Table 3.3 Logical Operators

Operator	**Description**
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Exactly equal to
`!=`	Not equal to
`!x`	Not x
`x | y`	x or y
`x & y`	x and y
`isTRUE(x)`	Test if x is TRUE

Let's say that we want to recode the ages of the managers in our leadership dataset from year to age category (Young, Middle Aged, Elder). We could use the code in listing 3.2

Listing 3.2 Recoding variables

```
# Create 3 age categories from the age variable
attach(leadership)
leadership$agecat[age > 75] <- "Elder"
leadership$agecat[age > 45 & age <= 75] <- "Middle Aged"
leadership$agecat[age <= 45] <- "Young"
detach(leadership)
```

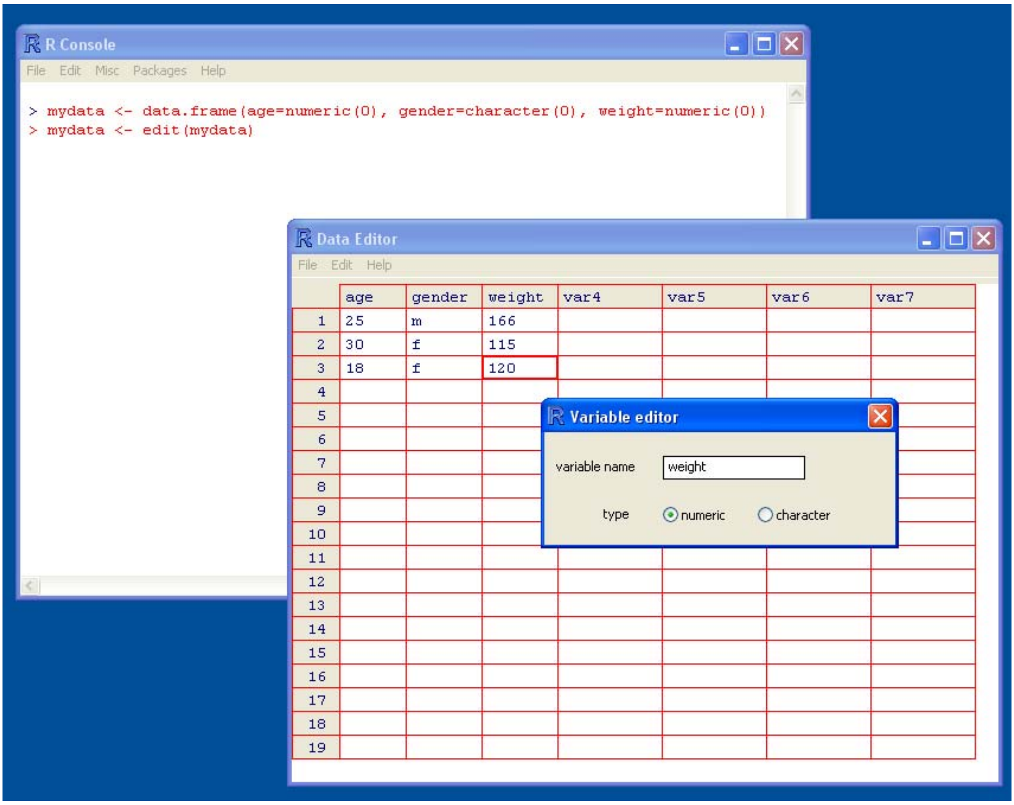
The statement `variable[condition] <- expression` will only make the assignment when `condition` is TRUE. We have included the dataframe name in `leadership$agecat` to ensure that the new variable is saved back to the dataframe. We used the `attach(leadership)` statement so that we could write age rather than `leadership$age`. We chose middle aged to be between 45 and 75 so that I wouldn't feel so old.

3.4 Renaming variables

If we are not happy with our variable names, we can change them interactively or programmatically. Let's say that we wanted to change the variables `manager` to `managerID` and `date` to `testDate`. We could use the statement

```
fix(leadership)
```

to invoke an interactive editor, click on the variable names, and rename them in the dialog



```
> mydata <- data.frame(age=numeric(0), gender=character(0), weight=numeric(0))
> mydata <- edit(mydata)
```

	age	gender	weight	var4	var5	var6	var7
1	25	m	166				
2	30	f	115				
3	18	f	120				
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							

variable name: weight

type: [✓ numeric] character

boxes that are presented (see figure 3.1).

Figure 3.1 Renaming variables interactively using the fix function

Programmatically, the `reshape` package has a `rename` function that is very useful for altering the names of variables. The format of the rename function is

```
rename(dataframe, c(oldname="newname", oldname="newname",…))
```

An example is given in listing 3.3.

Listing 3.3 Renaming variables with the reshape package

```
# rename programmatically
library(reshape)
leadership <- rename(leadership,
  c( manager="managerID", date="testDate" )
)
```

Since the `reshape` package is not installed by default, you will need to install on first use using the `install.packages("reshape")` command. The reshape package has a very powerful set of functions for altering the structure of a dataset. We will explore several in chapter 4.

Finally, you can rename variables by re-entering the variable name in order, while changing the ones that need to be altered. For example:

```
names(leadership) <- c("testDate", "country", "gender", "age",
   "managerID", "q1", "q2', "q3", "q4", "q5")
```

The limitation of this approach is the need to enter all the variable names, not just those that we want to rename. If there are dozens or hundreds of variables, this becomes impractical.

3.5 Missing values

In a project of any size, data is likely to be incomplete, because of missed questions, fautly equipment, or improperly coded data. In R, missing values are represented by the symbol NA (not available). Impossible values (e.g., dividing by zero) are represented by the symbol NaN (not a number). Unlike programs like SAS, R uses the same missing values symbol for character and numeric data.

In our leadership example, we could use the code in listing 3.4 to read the data from a tab delimited text file.

Listing 3.4 Reading data with missing values

```
> leadership <- read.table("leadership.csv", header=TRUE, sep="\t")
> leadership

  manager     date country gender age q1 q2 q3 q4 q5
1       1 10/24/08      US      M  32  5  4  5  5  5
2       2 10/28/08      US      F  45  3  5  2  5  5
3       3 10/01/08      UK      F  25  3  5  5  5  2
4       4 10/12/08      UK      M  39  3  3  4 NA NA
5       5 05/01/09      UK      F  99  2  2  1  2  1
```

Note that when blank values are read into a dataframe, they are automatically converted to missing values.

R provides a number of functions for identifying observations containing missing values. The function `is.na` allows us to test for the presence of missing values. Assume that we have a vector

```
y <- c(1, 2, 3, NA)
```

then the function

`is.na(y) returns c(FALSE, FALSE, FALSE, TRUE).`

Notice how the `is.na` function works on an object. It returns an object of the same size, with the entries replaced by TRUE if the element is a missing value, and FALSE if the element is not a missing value. Using our leadership example in listing 3.5:

Listing 3.5 Applying the is.na function

```
> is.na(leadership[,6:10])
        q1    q2    q3    q4    q5
[1,] FALSE FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE FALSE
[3,] FALSE FALSE FALSE FALSE FALSE
[4,] FALSE FALSE FALSE  TRUE  TRUE
[5,] FALSE FALSE FALSE FALSE FALSE
```

Here, `leadership[,6:10]` limited the dataframe to all rows, and columns 6 to 10, while `is.na` identified which values are missing.

IMPORTANT NOTE

Missing values are considered non-comparable, even to themselves. This means that you cannot use comparison operator to test for the presence of missing values. For example, the logical test `myvar == NA` is never TRUE. Instead, you have to use missing values functions, like those in this section, to identify the missing values in R data objects.

3.5.1 Recoding values to missing

We can use assignments to recode values to missing. In our leadership example, missing age values were coded as 99. Before analyzing this dataset, we need to let R know that the value 99 means missing in this case (otherwise the mean age for this sample of bosses will be way off!). We can accomplish this with the following code.

```
# recode 99 to missing for the variable age
attach(leadership)
leadership[age==99, "age"] <- NA
```

The code fragment `leadership[age==99, "age"]` selects the age column in the dataframe `leadership` and within this column, the rows in which age is equal to 99. The assignment <-NA then sets these age values to missing. Be sure that any missing data is properly coded as missing before analyzing the data or the results will be meaningless.

3.5.2 Excluding missing values from analyses

We need to eliminate missing values in some way before analyzing our data. This is because arithmetic expressions and functions that contain missing values yield missing values. For example

```
x <- c(1,2,NA,3)
y <- c[1] + c[2] + c[3] + c[4] # y returns NA
z <- sum(x)                    # z returns NA
```

Both `y` and `z` will be NA (missing) because the 3rd element of `x` is missing.

Luckily, most numerical functions have a `na.rm=TRUE` option that removes missing values prior to calculations, and applies the function to the remaining values.

```
x <- c(1,2,NA,3)
sum(x, na.rm=TRUE) # returns 6
```

When using functions with incomplete data, be sure to check how that function handles missing data by looking at its online help (e.g. `help(sum)`). The `sum` function is only one of many functions we will consider in chapter 4. They allow us to transform data with flexibility and ease.

We can remove *any* observation with missing data using the na.omit function. `na.omit` deletes any rows with missing data. We apply this to our leadership dataset in listing 3.6.

Listing 3.6 Using na.omit to delete incomplete observations

```
# create new dataset without missing data
> leadership

  manager     date country gender age q1 q2 q3 q4 q5
1       1 10/24/08      US      M  32  5  4  5  5  5
2       2 10/28/08      US      F  40  3  5  2  5  5
3       3 10/01/08      UK      F  25  3  5  5  5  2
4       4 10/12/08      UK      M  39  3  3  4 NA NA
5       5 05/01/09      UK      F  99  2  2  1  2  1

> newdata <- na.omit(leadership)                         1
> newdata

  manager     date country gender age q1 q2 q3 q4 q5
1       1 10/24/08      US      M  32  5  4  5  5  5
2       2 10/28/08      US      F  40  3  5  2  5  5
3       3 10/01/08      UK      F  25  3  5  5  5  2
```

```
  5       5 05/01/09      UK      F   99   2   2   1   2   1
```

Any rows containing missing data are deleted from `leadership` before the results are saved to `newdata` #1.

Deleting all observations with missing data (called listwise deletion) is one of several methods of handling incomplete datasets. If there are only a few missing values or they are concentrated in a small number of observations, listwise deletion can provide a good solution to the missing values problem. However, if missing values are spread throughout the data, or there is a great deal of missing data in a small number of variables, listwise deletion can exclude a substantial percentage of our data. We will look at several more sophisticated methods of dealing with missing values in chapter 15. Next, let's take a look at dates.

3.6 Date values

Dates are typically entered into R as character strings and then translated into date variables that are stored numerically. The function `as.Date` is used to make this translation. The syntax for is `as.Date(x, "format")`, where `x` is the character data and `format` gives the appropriate format from table 3.4.

Table 3.4 Date formats

Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a	abbreviated weekday	Mon
%A	unabbreviated weekday	Monday
%m	month (00-12)	00-12
%b	abbreviated month	Jan
%B	unabbreviated month	January
%y	2-digit year	07
%Y	4-digit year	2007

The default format is `yyyy-mm-dd`. Listing 3.7 provides two examples.

Listing 3.7 Converting character values to dates

```
# convert character data in format 'mm/dd/yyyy' to dates
strDates <- c("01/05/1965", "08/16/1975")
dates <- as.Date(strDates, "%m/%d/%Y")

# convert character data to dates using the default format
mydates <- as.Date(c("2007-06-22", "2004-02-13"))
```

When stored internally, dates are represented as the number of days since 1970-01-01, with negative values for earlier dates. This lets us perform arithmetic operations on dates such as the one in listing 3.8.

Listing 3.8 Calculations with with dates

```
> startdate <- as.Date("2004-02-13")
> enddate   <- as.Date("2009-06-22")
> days      <- enddate - startdate
> days

Time difference of 1956 days
```

In our leadership dataset, date is coded as a character variable in mm/dd/yy format. We could use a commands to transform them into date values.

```
myformat <- “%m/%d/%y”
leadership$date <- as.date(leadership$date, myformat)
```

Here, we use the specified format to read the character variable and replace it in the dataframe as a date variable. Once in date format, we can analyze and plot the dates using the wide range of analytic techniques that we cover in later chapters.

There are two useful functions that take no arguments, and return the current date and/or time. Specifically

- `Sys.Date()` returns today's date
- `Date()` returns the current date and time.

We can use these functions to time stamp events, or to calculate the amount of time that has passed between an event and the present. Listing 3.9 provides two examples of their use.

Listing 3.9 Date functions and formatted printing

```
> # print today's date
> today <- Sys.Date()
> format(today, format="%B %d %Y")               1

[1] "July 07 2009"

> # day I was born (not really)
> dob <- as.Date(“1956-10-10”)
> format(dob, format="%A")                        2

   [1] “Monday”
```

These examples also use the `format` function. The `format` function takes an argument (a date in this case), and applies a format (in this case assembled from the symbols in table

3.2). This not only gives us control over the way the dates are printed #1, but also allows us to extract portions of the date values #2.

3.6.1 Converting dates to character variables

Although less commonly used, we can also convert date variables to a character format. Date values can be converted to character variables using the `as.Character` function. For example

```
# convert dates to character data
strDates <- as.character(dates)
```

The conversion allows us to apply a range of character functions to the data values (subsetting, replacement, concatenation, etc.). We will cover character functions in detail in chapter 4.

3.6.2 Going further

To learn more about converting character data to dates, take a look at `help(as.Date)` and `help(strftime)`. To learn more about formatting dates and times, see `help(ISOdatetime)`. If you need to do complex calculations with dates, the `fCalendar` package can help. It provides a myriad of functions for dealing with dates, can handle multiple time zones at once, and provides sophisticated calendar manipulations that recognize business days, weekends, and holidays.

3.7 Type conversions

In the previous section, we discussed how to convert character data to date values and vice-versa. R provides a set of functions to identify an object's data type, and convert it to a different data type.

Type conversions in R work in a similar fashion to those in other statistical programming languages. For example, adding a character string to a numeric vector converts all the elements in the vector to character values. We can use the functions listed in table 3.5 to test for a data type and to convert that to a given type.

Table 3.5. Type conversion functions

Test	Convert
`is.numeric`	`as.numeric`
`is.character`	`as.character`
`is.vector`	`as.vector`
`is.matrix`	`as.matrix`
`is.data.frame`	`as.data.frame`

Functions of the form `is.`*`datatype`* return TRUE or FALSE, while `as.`*`datatype`* converts the argument to that type. Listing 3.10 provides an example.

Listing 3.10 Converting from one data type to another

```
> a <- c(1,2,3)
> a

[1] 1 2 3

> is.numeric(a)

[1] TRUE

> is.vector(a)

[1] TRUE


> a <- as.character(a)
> a

[1] "1" "2" "3"

> is.numeric(a)

[1] FALSE

> is.vector(a)

[1] TRUE

> is.character(a)

[1] TRUE
```

When combined with the flow controls (e.g., if-then) that we will discuss in chapter 4, the `is.`*`datatype`* function can be a powerful tool, allowing us to handle data in different ways, depending on its type. Additionally, some R functions require data of a specific type (character or numeric, matrix or dataframe) and the `as.`*`datatype`* will allow us to transform our data into the format required prior to analyses.

3.8 Sorting data

Sometimes, just viewing a dataset in a sorted order can tell us quite a bit about the data. For example, which managers are most deferential? To sort a dataframe in R, use the `order` function. By default, the sorting order is `ASCENDING`. Prepend the sorting variable with a minus sign to indicate a `DESCENDING` order. Some examples are provided in listing 3.11.

Listing 3.11 Sorting a dataset

```
# sorting examples using the leadership dataset

# sort by age
newdata <- leadership[order(age),]                  1

# sort by gender and age
newdata <- leadership[order(gender, age),]          2

#sort by gender (ascending) and age (descending)   3
newdata <-leadership[order(gender, -age),]
```

In #1 the dataset is sorted from youngest manager to oldest manager. In #2 the dataset is sorted into female followed by male, and age is sorted (younger first) within each of the gender groups. In #3 age is sorted from oldest to youngest manager within each gender.

3.9 Merging datasets

If our data exist in multiple locations, we will need to combine them before moving forward.

3.9.1 Adding Columns

To merge two dataframes (datasets) horizontally, we use the `merge` function. In most cases, two dataframes are joined by by one or more common key variables (i.e., an inner join). Two examples are given in listing 3.12.

Listing 3.12 Merging datasets horizontally

```
# merge two dataframes by ID
total <- merge(dataframeA,dataframeB,by="ID")

# merge two dataframes by ID and Country
total <- merge(dataframeA,dataframeB,by=c("ID","Country"))
```

Horizontal joins like this are usually used to add variables to a dataframe.

NOTE

If you are simply joining two matrices or dataframes horizontally and do not need to specify a common key, you can use the `cbind` function:

```
total <- cbind(A, B)
```

This will horizontally concatenate the objects A and B. For this to work properly, each object has to have the same number of rows and be sorted in the same order.

3.9.2 Adding Rows

To join two dataframes (datasets) vertically, use the `rbind` function. The two dataframes **must** have the same variables, but they do not have to be in the same order (see listing 3.13).

Listing 3.13 Merging dataset vertically

```
# merge two dataframes vertically
total <- rbind(dataframeA, dataframeB)
```

If dataframeA has variables that dataframeB does not, then either:

- delete the extra variables in dataframeA or
- create the additional variables in dataframeB and set them to NA (missing)

before joining them. Vertical concatenation is usually used to add observations to a dataframe.

3.10 Subsetting datasets

R has powerful indexing features for accessing the elements of an object. These features can be used to select and exclude variables, observations, or both. The following sections demonstrate several methods for keeping or deleting variables and observations.

3.10.1 Selecting (Keeping) Variables

It is a common practice to create a new dataset from a limited number of variables chosen from a larger dataset. Listing 3.14 describes three different ways of accomplishing the same selection of variables.

Listing 3.14 Selecting variables

```
# select variables q1, q2, q3, q4, q5 from the leadership dataframe

# method 1
newdata <- leadership[, c(6:10)]               1

# method 2
myvars <- c("q1", "q2", "q3", "q4", "q5")      2
newdata <-leadership[myvars]

# method 3
myvars <- paste("q", 1:5, sep="")              3
newdata <- leadership[myvars]
```

In chapter 2, we saw that the elements of a dataframe are accessed using the notation `dataframe[row indices, column indices]`. In #1 we left the row indices blank (,) which selected all rows by default. For the column indices, we selected columns 6 through 10 which translated to variables q1 through q5. In #2, we entered variable names (in quotes) as column indices, thereby selecting those columns. When variable names are entered as column indices, the row indices are assumed and can be left out. Finally, in #3, we used the `paste` function to create the same character vector as in the previous example. The `paste` function will be covered in chapter 4.

3.10.2 Excluding (dropping) Variables

There are many reasons to exclude variables. For example, if a variable has many missing values, we may want to drop the entire variable prior to further analyses. Several methods of excluding variables are presented in listing 3.15.

Listing 3.15 Dropping variables

```
# exclude variables q3 and q4 three different ways

myvars <- names(leadership) %in% c("q3", "q4")      1
newdata <- leadership[!myvars]

# exclude 8th and 10th variable
newdata <- leadership[c(-8,-9)]                     2

# delete variables q3 and q4
leadership$q3 <- leadership$q4 <- NULL              3
```

In order to understand why #1 works, we need to break it down:

10. `names(leadership)` produces a character vector containing the variable names.
    `c("managerID","testDate","country","gender","age","q1","q2", "q3","q4","q5")`
11. `names(leadership) %ini% c("q3", "q4")` returns a logical vector with TRUE for each element in `names(leadership)`that matches `q3` or `q4` and FALSE otherwise.
    `c(FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,  TRUE,  TRUE, FALSE)`
12. The not (!) operator reverses the logical values
    `c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE,  FALSE, FALSE, TRUE)`
13. `leadership[c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE,  FALSE, FALSE, TRUE)] selects columns with TRUE logical values, so q3 and q4 are excluded.`

The code in #2 works because prepending a column index with a minus sign (-) excludes that column. The third example #3 accomplishes the same goal by setting the columns q3 and q5 to undefined (NULL). Note that NULL is not the same as NA (missing).

Dropping variables is simply the converse of keeping variables. The choice will depend on which is easier to code. If there are many variables to drop, it may be easier to keep the ones that remain, or vice versa.

3.10.3 Selecting Observations

Selecting or excluding observations (rows) is typical a key aspect of successful data preparation and analysis. Several examples are given in listing 3.16.

Listing 3.16 Selecting observations

```
# first 5 observerations
newdata <- leadership[1:5,]                              1

# based on variable values
newdata <- leadership[which(leadership$gender=="M"       2
& leadership$age > 30),]

# or
attach(leadership)
newdata <- leadership[which(gender=='M' & age > 30),]     3
detach(leadership)
```

In each of these examples, we provide the row indices and leave the column indices blank (therefore choosing all columns). In #1 we ask for rows 1 through 5 (the first 5 observations). We need to break #2 down to understand it:

14. The logical comparison `leadership$gender=="F"` produces the vector `c(TRUE, FALSE, FALSE, TRUE, FALSE)`
15. The logical comparison `leadership$age > 30` produces the vector `c(TRUE, TRUE, FALSE, TRUE, TRUE)`
16. The logical comparison `c(TRUE, FALSE, FALSE, TRUE, TRUE) & c(TRUE, TRUE, FALSE, TRUE, TRUE)` produces the vector `c(TRUE, FALSE, FALSE, TRUE, FALSE)`
17. The function `which` gives the indices of a vector that are TRUE. Thus `which(c(TRUE, FALSE, FALSE, TRUE, FALSE))` produces the vector `c(1, 4)`
18. `leadership[c(1,4),]` selects the first and fourth observations from the dataframe. This matches our criteria.

#3 is identical to #2 but uses the `attach` function so that we do not have to prepend the variable names with the dataframe names.

At the beginning of this chapter, we suggested that we might want to limit our analyses to observations collected between January 1, 2009 and December 31, 2009. How can we do this? One solution is presented in listing 3.17.

Listing 3.17 Selecting observations based on dates

```
# select observations recorded between Jan 1 2009 and Dec 31 2009
leadership$date <- as.Date(leadeship$date, "%m/%d/%y")          1
startdate <- as.Date("2009-01-01")                              2
enddate   <- as.Date("2009-01-31")
newdata <- leadership[which(leadership$date >= startdate & [CA]   3
    leadership$date <= enddate),]
```

#1 We convert the date values read in originally as character values to date values using the format `mm/dd/yy`. #2 We create starting and ending dates. Since the default for the `as.Date` function is `yyyy-mm-dd`, we don't have to supply it here. #3 Finally, we select cases meeting our desired criteria as we did in the previous example.

3.10.4 The Subset Function

The examples in the previous two sections are important because they help describe the ways in which logical vectors and comparison operators are interpreted within R. Understanding how these examples work will go a long way to making R more understandable for you. Now that we have done things the hard way, let's look at a shortcut.

The `subset` function is probably the easiest way to select variables and observation. Two examples are given in listing 3.18.

Listing 3.18 Using the subset function

```
# using subset function
newdata <- subset(leadership, age >= 35 | age < 24,       1
select=c(q1, q2, q3, q4))

# using subset function (another example)
newdata <- subset(leadership, sex=="M" & age > 25,        2
select=gender:q4)
```

In #1 we select all rows that have a value of age greater than or equal to 35 *or* age less than 24. We keep the variables q1 through q4. In the second example #2, we select all men over the age of 25 and we keep variables gender through q5 (gender, q4, and all columns between them). We have seen the `colon` operator `from:to` in chapter 2. Here, it provides all variables in a dataframe between the `to` variable and the `from` variable, inclusive.

3.10.5 Random Samples

Sampling from larger datasets is common practice in data mining and machine learning. For example, we may want to select two random samples, creating a predictive model from one, and validating its effectiveness on another. The `sample` function allows us to take a random sample (without or without replacement) of size `n` from a dataset. An example is provided in listing 3.19.

Listing 3.19 Taking a random sample

```
# take a random sample of size 50 from the leadership dataset
# sample without replacement
mysample <- leadership[sample(1:nrow(leadership), 50,[CA]     1
   replace=FALSE),]
```

#1 The first argument to the `sample` function is a vector listing elements to be randomly chosen from. Here, the vector is 1 to the number of observations in the dataframe. The second argument is indices to be selected, and the third argument indicates sampling without replacement. The `sample` function returns the randomly sampled indices, which are then used to select rows from the dataframe.

GOING FURTHER

R has extensive facilities for sampling, including drawing and calibrating survey samples (see the `sample` package) and analyzing complex survey data (see the `survey` package). Bootstrapping is described in appendix d.

3.11 Summary

We have covered a great deal of ground in this chapter. We have looked at the way R stores missing and date values and explored various ways of handling them. We have seen how to determine the data type of an object and how to convert it to other types. We have used simple formulas to create new variables and recode existing variables. We have sorted our data and renamed our variables. We have seen how to merge our data with other datasets both horizontally (adding variables) and vertically (adding observations). Finally, we have seen how to keep or drop variables and how to select observations based on a variety of criteria.

Actually, we have only scratched the surface when it comes to handling incomplete data. In the next chapter, we will address the "missing value problem" in more detail and discuss more sophisticated methods of dealing with it. Then we will look at the myriad of arithmetic, character, and statistical functions that R makes available for creating and transforming variables. After exploring ways of controlling program flow, we will see how to write our own functions. Finally, we will explore how we can use these functions to aggregate and summarize our data.

By the end of chapter 4 you will have most of the tools necessary to manage complex datasets. (And you will be the envy of data analysts everywhere!)

4 Advanced data management

This Chapter covers:

- Mathematical and statistical functions
- Character functions
- Looping and conditional execution
- User-written functions
- Aggregating and reshaping data

In chapter 3, we reviewed the basic techniques used for managing datasets within R. In this chapter, we will focus on advanced topics. The chapter is divided into three basic parts. In the first part we will take a whirlwind tour of R's many functions for mathematical, statistical, and character manipulation. In order to give this section relevance, we begin with a data management problem that can be solved using these functions. After covering the functions themselves, we will look at one possible solution to the problem we raised.

In the second part we will look at how we can write our own functions to accomplish data management and analysis tasks. First, we will look at ways of controlling program flow, including looping and conditional statement execution. Then we will look at the structure of user-written functions and how to invoke them once created.

In the third part, we will look at ways of aggregating and summarizing our data, along with methods of reshaping and restructuring our datasets. When aggregating data, we can specify the use of any appropriate built-in or user-written function to accomplish the summarization, so the topics we learned in the first two parts of the chapter will provide real benefit.

Finally, we will pause for a well deserved rest.

4.1 A data management challenge

In order to motivate our discussion of numerical and character functions, we will start with a data management problem. A group of students have taken exams in Math, Science and English. We want to combine these scores in order to determine a single performance indicator for each student. Additionally, we want to assign an “A” to the top 20% of students, “B” to the next 20%, etc. Finally, we want to sort the students alphabetically. The data are presented in table 4.1.

Table 4.1 Student exam data

Student	**Math**	**Science**	**English**
John Davis	502	95	25
Angela Williams	600	99	22
Bullwinkle Moose	412	80	18
David Jones	358	82	15
Janice Markhammer	495	75	20
Cheryl Cushing	512	85	28
Reuven Ytzrhak	410	80	15
Greg Knox	625	95	30
Joel England	573	89	27
Mary Rayburn	522	86	18

Looking at this dataset, several obstacles are immediately evident. First, scores on the three exams are not comparable. They have widely different means and standard deviations, so simply averaging them does not make sense. We must transform the exam scores into comparable units before combining them. Second, we will need a method of determining a student's percentile rank on this score, in order to assign a grade. Third, there is single field for name, complicating the task of sorting students. We will need to break apart their names into first name and last name in order to sort them properly.

Each of these tasks can be accomplished through the judicious use of R's numerical and character functions. After we work through the functions described next section, we will consider a possible solution to this data management challenge.

4.2 Numerical and character functions

In this section we will review functions in R that can be used as the basic building blocks for manipulating data. We can divide them into numerical (mathematical, statistical, probability) and character functions. After we review each type, we will look at how to apply functions to the columns (variables) and rows (observations) of matrices and dataframes.

4.2.1 Mathematical functions

Table 4.2 lists common mathematical functions along with short examples.

Table 4.2 Mathematical functions

Function	Description
`abs(x)`	Absolute value `abs(-4)` is 4
`sqrt(x)`	Square root `sqrt(25)` is 5
`ceiling(x)`	Smallest integer not less than x `ceiling(3.475)` is 4
`floor(x)`	Largest integer not greater than x `floor(3.475)` is 3
`trunc(x)`	Integer formed by truncating values in x toward 0 `trunc(5.99)` is 5
`round(x, digits=n)`	Round x to the specified number of decimal places `round(3.475, digits=2)` is 3.48
`signif(x, digits=n)`	Round x to the specified number of significant digits `signif(3.475, digits=2)` is 3.5
`cos(x), sin(x), tan(x)`	Cosine, sine, and tangent `cos(2)` is -0.416
`acos(x), asin(x), atan(x)`	Arc-cosine, arc-sine, and arc-tangent `acos(-0.416)` is 2
`cosh(x), sinh(x), tanh(x)`	Hyperbolic cosine, sine, and tangent `sinh(2)` is 3.627
`acosh(x), asinh(x), atanh(x)`	Hyperbolic arc-cosine, arc-sine, and arc-tangent `asinh(3.627)` is 2
`log(x,base=n)`	Logarithm of x to the base n

`log(x)` `log10(x)`	For convenience `log(x)` is the natural logarithm `log10(x)` is the common logarithm `log(10)` is 2.3026 `log10(10)` is 1
`exp(x)`	Exponential function `exp(2.3026)` is 10

Data transformation is one of the primary uses for these functions. For example, we often transform positively skewed variables such as income to a log scale before further analyses. Mathematical functions will also be used as components in formulas, in plotting functions (e.g., x vs. sin(x)) and in formatting numerical values prior to printing.

4.2.2 Statistical Functions

Common statistical functions are presented in table 4.3. Many of these functions have optional parameters that affect the outcome. For example

```
y <- mean(x)
```

provides the arithmetic mean of the elements in object x, while

```
z <- mean(x, trim = 0.5, na.rm=TRUE)
```

provides the trimmed mean, dropping the highest and lowest 5% of scores and any missing values. Use the `help` function to learn more about each function and its arguments.

Table 4.3 Statistical functions

Function	Description
`mean(x)`	Mean `mean(c(1,2,3,4))` is `2.5`
`median(x)`	Median `median(c(1,2,3,4))` is 2.5
`sd(x)`	Standard deviation `sd(c(1,2,3,4)` is 1.29
`var(x)`	Variance `variance (c(1,2,3,4))` is 1.67
`mad(x)`	Median absolute deviation `mad(c(1,2,3,4))` is 1.48

`quantile(x, probs)`	Quantiles where `x` is the numeric vector whose quantiles are desired and `probs` is a numeric vector with probabilities in [0,1].
	`# 30th and 84th percentiles of x`
	`y <- quantile(x, c(.3,.84))`

`range(x)`	Range `x <- c(1,2,3,4)` `range(x)` is c(1,4) `diff(range(x))` is 3
`sum(x)`	Sum `sum(c(1,2,3,4)` is 10

`diff(x, lag=1)`	Lagged differences, with lag indicating which lag to use. The default lag is 1.
	`x<- c(1, 5, 23, 29)`
	`diff(x)` is `c(4, 18, 6)`

`min(x)`	Minimum `min(c(1,2,3,4))` is 1
`max(x)`	Maximum `max(c(1,2,3,4)` is 4
`scale(x, center=TRUE, scale=TRUE)`	Column center (`center=TRUE`) or standardize (`center=TRUE, scale=TRUE`) data object `x`. An example is given in listing 4.2.

To see these functions in action, look at listing 4.1. Here we demonstrate two ways to calculate the mean and standard deviation of a vector of numbers.

Listing 4.1 Calculating the mean and standard deviation

```
> x <- c(1,2,3,4,5,6,7,8)
> mean(x)

[1] 4.5

> sd(x)

[1] 2.449490

> # same thing the hard way
> n <- length(x)
> meanx <- sum(x)/n
> css <- sum((x - meanx)**2)                             1
> sdx <- sqrt(css / (n-1))
> meanx
```

```
[1] 4.5
> sdx
[1] 2.449490
```

#1 It is instructive to view how the corrected sum of squares (css) is calculated step by step:

19. x equals `c(1, 2, 3, 4, 5, 6, 7, 8)` and `meanx` equals 4.5
20. `(x – meanx)` subtracts `4.5` from each element of `x` resulting in `c(-3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5)`
21. `(x – meanx)**2` squares each element of `(x - meanx)` resulting in `c(12.25, 6.25, 2.25, 0.25, 0.25, 2.25, 6.25 12.25)`
22. `sum((x-meanx)**2)` sums each of the elements of `(x-meanx)**2)` resulting in 42

Writing formulas in R has much in common with matrix manipulation languages such as MATLAB (we will look more specifically at solving matrix algebra problems in appendix E).

By default, the `scale` function standardizes the columns of a matrix or dataframe to a mean of zero and a standard deviation of one. To standardize each column to an arbitrary mean and standard deviation you could use code similar to listing 4.2.

Listing 4.2 Standardizing the columns of a dataset

```
# standardize columns of a dataset to mean=0 and standard deviation=1
newdata <- scale(mydata)

# standardize columns of a dataset to an arbitrary
# mean M and standard deviation SD
newdata <- scale(mydata)*SD + M
```

We will use this approach as one step in solving our learning example (section 4.2.7).

4.2.3 Probability Functions

You may wonder why probability functions are not listed with the statistical functions above (it was really bothering you, wasn't it?). Although probability functions are statistical by definition, they are unique enough to deserve their own section. Probability functions are often used to generate simulated data with known characteristics and to calculate probability values within user written statistical functions.

In R, probability functions take the form

`[dpqr]`*distribution_abbreviation*

where the first letter refers to the aspect of the *distribution* returned:

`d` = density

p = distribution function
q = quantile function
r = random generation (random deviates)

The common probability functions are listed in table 4.4.

Table 4.4 Probability distributions

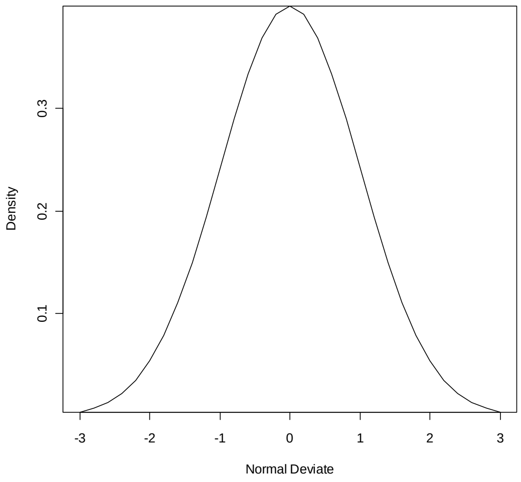
Distribution	Abbreviation	Distribution	Abbreviation
Beta	beta	Logistic	logis
Binomial	binom	Multinomial	multinom
Cauchy	cauchy	Negative binomial	nbinom
Chi-Squared (noncentral)	chisq	Normal	norm
Exponential	exp	Poisson	pois
F	f	Wilcoxon Signed Rank	signrank
Gamma	gamma	T	t
Geometric	geom	Uniform	unif
Hypergeometric	hyper	Weibull	weibull
Lognormal	lnorm	Wilcoxon Rank Sum	wilcox

To see how these work, we will look at functions related to the normal distribution. If we do not specify a mean and a standard deviation, the standard normal distribution is assumed (mean=0, sd=1). Examples of the density (`dnorm`), distribution (`pnorm`), quantile (`qnorm`) and random deviate generation (`rnorm`) functions are given in table 4.5.

Table 4.5 Normal distribution functions

Problem	Solution

Plot the standard normal curve on the interval [-3,3] (see below)	`x <- pretty(c(-3,3), 30)`
	`y <- dnorm(x)`
	`plot(x, y,`
	`  type = 'l',`
	`  xlab = Normal Deviate",`
	`  ylab = "Density",`
	`  yaxs = "i"`
	`)`



What is the area under the standard normal curve to the right of z = 1.96?	`pnorm(1.96)`equals 0.975
What is the value of the 90th percentile of a normal distribution with a mean of 500 and a standard deviation of 100?	`qnorm(.9, mean=500, sd=100)` equals 628.16
Generate 50 random normal deviates with a mean of 50 and a standard deviation of 10.	`rnorm(50, mean=50, sd=10)`

Don't worry if the plot function options are unfamiliar. We will cover them in detail in later chapters.

SETTING THE SEED FOR RANDOM NUMBER GENERATION

Each time we generate pseudo-random deviates, a different seed and therefore different results, are produced. In order to make our results reproducible, we can specify the seed explicitly, using the `set.seed` function. An example is given in listing 4.3.

Listing 4.3 Generating pseudo-random numbers from a uniform distribution

```
> # generate 5 uniform random deviates
> runif(5)

[1] 0.8725344 0.3962501 0.6826534 0.3667821 0.9255909

> runif(5)

[1] 0.4273903 0.2641101 0.3550058 0.3233044 0.6584988
```

```
> set.seed(1234)                                                     1
> runif(5)

[1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154

> set.seed(1234)                                                     1
> runif(5)

[1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154
```

#1 By setting the seed manually, we are able to reproduce our results. This can be very helpful in creating examples we can access at a future time and share with others.

GENERATING MULTIVARIATE NORMAL DATA

In simulation research and Monte Carlo studies, we often want to draw data from multivariate normal distribution with a given mean vector and covariance matrix. The `mvrnorm` function in the MASS package makes this easy. The function call is

```
mvrnorm(n, mean, sigma)
```

where n is the desired sample size, mean is the vector of means, and sigma is the variance-covariance (or correlation) matrix. In listing 4.4 we will sample 500 observations from a 3-variable multivariate normal distribution with

Mean Vector	230.7	146.7	3.6
Covariance Matrix	15360.8	6721.2	-47.1
	6721.2	4700.9	-16.5
	-47.1	-16.5	0.3

Listing 4.4 Generating data from a multivariate normal distribution

```
> mean <- c(230.7, 146.7, 3.6)                                           1
> sigma <- ( c(15360.8, 6721.2, -47.1,                                   2
                6721.2, 4700.9, -16.5,
                 -47.1,  -16.5,   0.3), nrow=4, ncol=4)
> set.seed(1234)                                                         3
> mydata <- mvrnorm(500, mean, sigma)                                    4
> mydata <- as.data.frame(mydata)                                        5
> names(mydata) <- c("y","x1","x2")                                      6
> dim(mydata                                                             7

[1] 500 3

> head(mydata, n=10)                                                     8

       y    x1   x2
```

```
1    98.8  41.2 4.33
2   244.5 205.2 3.57
3   375.7 186.7 3.66
4   -59.2  11.2 4.23
5   313.0 111.0 2.92
6   288.8 185.1 4.14
7   134.8 165.0 3.69
8   171.7  97.4 3.80
9   167.3 101.0 3.99
10  121.1  94.5 3.76
```

In the listing above, we are generating 500 observations from a multivariate normal distribution with a given #1 mean vector, and #2 variance-covariance matrix. Since a correlation matrix is a covariance matrix too, we could have specified the correlations structure directly here. We have #3 set a random number seed so that we can reproduce our results later. We #4 generate the pseudo-random data, #5 convert it to a dataframe from a matrix, and #6 name the variables. Finally, we #7 confirm that we have 500 observations and 3 variables, and #8 print out the first 10 observations.

The probability functions in R allow us to generate simulated data, sampled from distributions with known characteristics. Statistical methods that rely on simulated data have grown exponentially in recent years and we will see several examples of these in later chapters.

4.2.4 Character functions

While mathematical and statistical functions operate on numerical data, character functions extract information from textual data, or reformat textual data for printing and reporting. For example, we may want to concatenate a person's first name and last name, ensuring that the first letter of each is capitalized. Or we may want to count the instances of obscenities in open ended feedback. Some of the most useful character functions are listed in table 4.6.

Table 4.6 Character Functions

Function	Description
`nchar(x)`	Counts the number of characters of x `x <- c(“ab”, “cde”, “fghij”)` `length(x)` is 3 `nchar(x[3])` is 5

`substr(x, start, stop)`	Extract or replace substrings in a character vector.
	`x <- "abcdef"`
	`substr(x, 2, 4)` is bcd"
	`substr(x, 2, 4) <- "22222"` is `"a222ef"`

`grep(pattern, x, ignore.case=FALSE,`	Search for `pattern` in x. If `fixed=FALSE` then

`fixed=FALSE)`	pattern is a regular expression. If `fixed=TRUE` then pattern is a text string. Returns matching indices. `grep("A", c("b","A","c"), fixed=TRUE)` returns 2
`sub(pattern, replacement, x, ignore.case=FALSE, fixed=FALSE)`	Find `pattern` in `x` and substitue with `replacement` text. If `fixed=FALSE` then pattern is a regular expression. If `fixed=TRUE` then pattern is a text string. `sub("\\s",".","Hello There")` returns `Hello.There`
`strsplit(x, split)`	Split the elements of character vector `x` at `split`. `strsplit("abc", "")` returns `c("a","b","c")`

`paste(..., sep="")`	Concatenate strings after using `sep` string to separate them.
	`paste("x", 1:3, sep="")` returns `c("x1", "x2", "x3")`
	
	`paste("x",1:3,sep="M")` returns `c("xM1","xM2" "xM3")`
	
	`paste("Today is", date()) returns Today is Thu Jun 25 14:17:32 2011`
	(I changed the date to appear more current)

`toupper(x)`	Uppercase `toupper("abc")` returns "ABC"
`tolower(x)`	Lowercase `tolower("ABC")` returns "abc"

Note that the functions `grep` and `sub` can search for a text string (fixed=TRUE) or a regular expression (fixed=FALSE). Regular expressions provide a clear and concise syntax for matching a pattern of text. For example, the regular expression

```
^[hc]?at
```

matches any string that starts with zero or one occurrences of “h” or “c”, followed by “at”. The expression therefore matches "hat", "cat", and "at", but not “bat”. To learn more, see the *regular expression* entry in Wikipedia.

4.2.5 Other useful functions

The functions in table 4.7 are also quite useful for data management and manipulation, but they don't fit cleanly into the other categories.

Table 4.7 Other useful functions

Function	Description
`length(x)`	Length of object x `x <- c(2, 5, 6, 9)` `length(x)`is 4

`seq(from , to, by)`	Generate a sequence
	`indices <- seq(1,10,2)`
	`#indices is c(1, 3, 5, 7, 9)`

`rep(x, ntimes)`	Repeat x n times
	`y <- rep(1:3, 2)`
	`# y is c(1, 2, 3, 1, 2, 3)`

`cut(x, n)`	Divide continuous variable x into factor with n levels `y <- cut(x, 5)`
`pretty(x, n)`	Create pretty breakpoints. Divides a continuous variable `x` into `n` intervals, by selecting `n+ 1` equally spaced rounded values. Often used in plotting.

`cat(…)`	Concatenates the objects in … and outputs them
	`firstname <-  c("Jane")`
	`cat("Hello" , firstname, "\n")`

The last example demonstrates the use of escape characters in printing. Use \n for new lines, \t for tabs, and \' for a single quote, \b for backspace and so forth. For example, the code:

```
name <- "Bob"
cat( "Hello", name, "\b.\n", "Isn\'t R", "\t", "GREAT?\n")
```

produces

```
Hello Bob.
 Isn't R         GREAT?
```

Note that the second line is indented one space. When `cat` concatenates objects for output, it separates each by a space. That is why we included the backspace (\b) escape character before the period. Otherwise it would have produced "Hello Bob ."

How we apply the functions we have covered so far to numbers, strings, and vectors is intuitive and straightforward, but how do we apply them to matrices and dataframes? That is the subject of the next section.

4.2.6 Applying functions to matrices and dataframes

One of the interesting features of R functions is that they can be applied to a variety of data objects (scalars, vectors, matrices, arrays, and dataframes). An example is given in listing 4.5.

Listing 4.5 Apply functions to data objects

```
> a <- 5
> sqrt(5)

[1] 2.236068

> b <- c(1.243, 5.654, 2.99)
> round(b)

[1] 1 6 3

> log(c)
          [,1]     [,2]     [,3]     [,4]
[1,] 0.0000000 1.386294 1.945910 2.302585
[2,] 0.6931472 1.609438 2.079442 2.397895
[3,] 1.0986123 1.791759 2.197225 2.484907

> mean(c)

[1] 6.5
```

Notice that the mean of matrix c in the example above results in a scalar (6.5). The mean function took the average of all 12 elements in the matrix. But what if we wanted the 3 row means or the 4 column means?

R provides a function named apply that allows us to `apply` an arbitrary function to any dimension of a matrix, array, or dataframe. The format for the apply function is

```
apply(x, MARGIN, FUN, ...)
```

where `x` is our data object, `MARGIN` is the dimension index, `FUN` is a function we specify, and `. . .` are any parameters we want to pass to `FUN`. In a matrix or dataframe

MARGIN=1 indicates rows and MARGIN=2 indicates columns. Let's take a look at the examples in listing 4.6.

Listing 4.6 Applying a function to the rows (columns) of a matrix

```
> # create some data
> mydata <- matrix(rnorm(30), nrow=6)
> mydata

       [,1]   [,2]   [,3]    [,4]   [,5]
[1,]  1.138 -1.413 -0.187  0.9849 -0.788
[2,]  0.475  1.318 -0.246 -1.0987  0.504
[3,]  1.444 -0.174  2.269  0.4378  0.195
[4,] -0.631  0.493  1.179 -0.4615  2.645
[5,] -0.458  0.180 -0.760  0.0759 -0.577
[6,]  0.485  0.158  0.369 -0.1272  0.669

> apply(mydata, 1, mean)                              1

[1] -0.0531  0.1905  0.8344  0.6448 -0.3078  0.3106

> apply(mydata, 2, mean)                              2

[1]  0.4088  0.0936  0.4372 -0.0315  0.4415

> apply(mydata, 2, mean, trim=.4)                     3

[1]  0.4799  0.1689  0.0906 -0.0256  0.3495
```

In #1, we are calculating the 6 row means. In #2 we get the 5 column means. Finally, in #3, we get the column means, but this time we pass the option `trim=0.4` to the mean function, resulting in trimmed means.

Since FUN can be any R function, including a function that we write ourselves (see section 4.4), `apply` is a very powerful mechanism. While `apply` applies a function over the margins of an array, `lapply` and `sapply` apply a function over a list. We will see an example of `sapply` (which is actually a user-friendly version of `lappy`) in the next section.

We now have all the tools we need to solve the data challenge in section 4.1, so let's give it a try.

4.3 A solution for our data management challenge

Our challenge from section 4.1 is to combine subject test scores into a single performance indicator for each student, grade each student from A to F based on their relative standing (top 20%, next 20%, etc.), and sort the roster students last name, followed by first name. A solution is given in listing 4.7.

Listing 4.7 A solution to the learning example

```
# tranform student roster
options(digits=2)
```

```
# obtain performance scores
z <- scale(roster[,2:4])
score <- apply(z, 1, mean)
roster <- cbind(roster, score)

# grade students
y <- quantile(score, c(.8,.6,.4,.2))
roster$grade[score >= y[1]] <- "A"
roster$grade[score < y[1] & score >= y[2]] <- "B"
roster$grade[score < y[2] & score >= y[3]] <- "C"
roster$grade[score < y[3] & score >= y[4]] <- "D"
roster$grade[score < y[4]] <- "F"

# extract first and last name
name <- strsplit((roster$Student), " ")
lastname <- sapply(name, "[", 2)
firstname <- sapply(name, "[", 1)
roster <- cbind(firstname,lastname, roster[,-1])

# sort by last and first name
roster <- roster[order(lastname,firstname),]

# display results
> roster

    Firstname   Lastname Math Science English score grade
6      Cheryl    Cushing  512      85      28  0.35     C
1        John      Davis  502      95      25  0.56     B
9        Joel    England  573      89      27  0.70     B
4       David      Jones  358      82      15 -1.16     F
8        Greg       Knox  625      95      30  1.34     A
5      Janice Markhammer  495      75      20 -0.63     D
3  Bullwinkle      Moose  412      80      18 -0.86     D
10       Mary    Rayburn  522      86      18 -0.18     C
2      Angela   Williams  600      99      22  0.92     A
7      Reuven    Ytzrhak  410      80      15 -1.05     F
```

The code is dense so let's walk through the solution step by step.

Step1. The original student roster is given below. The options(digits=3) just limits the number of digits printed after the decimal place and makes the printouts easier to read.

```
> options(digits=3)
> roster

             Student Math Science English
1         John Davis  502      95      25
2    Angela Williams  600      99      22
3   Bullwinkle Moose  412      80      18
4        David Jones  358      82      15
5  Janice Markhammer  495      75      20
6     Cheryl Cushing  512      85      28
7     Reuven Ytzrhak  410      80      15
```

```
8          Greg Knox  625      95      30
9       Joel England  573      89      27
10      Mary Rayburn  522      86      18
```

Step 2. Since the Math, Science, and English tests are reported on different scales (with widely differing means and standard deviations), we need to make them comparable before combining them. One way to do this is to standardize the variables so that each test is reported in standard deviation units, rather than in their original scales. We can do this with the scale function.

```
> z <- scale(roster[,2:4])
> z

        Math Science English
 [1,]  0.013   1.078   0.587
 [2,]  1.143   1.591   0.037
 [3,] -1.026  -0.847  -0.697
 [4,] -1.649  -0.590  -1.247
 [5,] -0.068  -1.489  -0.330
 [6,]  0.128  -0.205   1.137
 [7,] -1.049  -0.847  -1.247
 [8,]  1.432   1.078   1.504
 [9,]  0.832   0.308   0.954
[10,]  0.243  -0.077  -0.697
```

Step 3. We can then get a performance score for each student by calculating the row means using the mean function and add it to the roster using the cbind function.

```
> score <- apply(z, 1, mean)
> roster <- cbind(roster, score)
> roster
            Student Math Science English  score
1        John Davis  502      95      25  0.559
2    Angela Williams  600      99      22  0.924
3   Bullwinkle Moose  412      80      18 -0.857
4       David Jones  358      82      15 -1.162
5  Janice Markhammer  495      75      20 -0.629
6     Cheryl Cushing  512      85      28  0.353
7     Reuven Ytzrhak  410      80      15 -1.048
8          Greg Knox  625      95      30  1.338
9       Joel England  573      89      27  0.698
10      Mary Rayburn  522      86      18 -0.177
```

Step 4. The quantile function will give us the percentile rank of each student's performance score. We see that the cutoff for an A is .74, for a B is .44, and so on.

```
> y <- quantile(roster$score, c(.8,.6,.4,.2))
> y

  80%   60%   40%   20%
 0.74  0.44 -0.36 -0.89
```

Step 5. Using logical operators, we can recode students' percentile ranks into a new categorical grade variable. This creates variable `grade`, in the `roster` dataframe.

```
> roster$grade[score >= y[1]] <- "A"
> roster$grade[score < y[1] & score >= y[2]] <- "B"
> roster$grade[score < y[2] & score >= y[3]] <- "C"
> roster$grade[score < y[3] & score >= y[4]] <- "D"
> roster$grade[score < y[4]] <- "F"
> roster

              Student Math Science English  score grade
1          John Davis  502      95      25  0.559     B
2     Angela Williams  600      99      22  0.924     A
3    Bullwinkle Moose  412      80      18 -0.857     D
4         David Jones  358      82      15 -1.162     F
5   Janice Markhammer  495      75      20 -0.629     D
6      Cheryl Cushing  512      85      28  0.353     C
7      Reuven Ytzrhak  410      80      15 -1.048     F
8           Greg Knox  625      95      30  1.338     A
9         Joel England  573      89      27  0.698     B
10      Mary Rayburn  522      86      18 -0.177     C
```

Step 6. We will use the `strsplit` function to break student names into first name and last name at the space character. Applying strsplit to a vector of strings, returns a list.

```
> name <- strsplit((roster$Student), " ")
> name

[[1]]
[1] "John"  "Davis"

[[2]]
[1] "Angela"   "Williams"

[[3]]
[1] "Bullwinkle" "Moose"

[[4]]
[1] "David" "Jones"

[[5]]
[1] "Janice"     "Markhammer"

[[6]]
[1] "Cheryl"  "Cushing"

[[7]]
[1] "Reuven"  "Ytzrhak"

[[8]]
[1] "Greg" "Knox"

[[9]]
[1] "Joel"    "England"
```

```
[[10]]
[1] "Mary"    "Rayburn"
```

Step 7. We can use the sapply function to take the first element of each component and put it in a firstname vector, and the second element of each component and put it in a lastname vector. We will use cbind to add them to the roster. Since we no longer need the student variable, we will drop it (with the -1 in the roster index).

```
> Firstname <- sapply(name, "[", 1)
> Lastname <- sapply(name, "[", 2)
> roster <- cbind(firstname,lastname, roster[,-1])
> roster

    Firstname   Lastname Math Science English  score grade
1        John      Davis  502      95      25  0.559     B
2      Angela   Williams  600      99      22  0.924     A
3  Bullwinkle      Moose  412      80      18 -0.857     D
4       David      Jones  358      82      15 -1.162     F
5      Janice Markhammer  495      75      20 -0.629     D
6      Cheryl    Cushing  512      85      28  0.353     C
7      Reuven    Ytzrhak  410      80      15 -1.048     F
8        Greg       Knox  625      95      30  1.338     A
9        Joel    England  573      89      27  0.698     B
10       Mary    Rayburn  522      86      18 -0.177     C
```

Step 8. Finally, we can sort the dataset by first and last name using the `order` function.

```
> roster[order(Lastname,Firstname),]

    Firstname   Lastname Math Science English score grade
6      Cheryl    Cushing  512      85      28  0.35     C
1        John      Davis  502      95      25  0.56     B
9        Joel    England  573      89      27  0.70     B
4       David      Jones  358      82      15 -1.16     F
8        Greg       Knox  625      95      30  1.34     A
5      Janice Markhammer  495      75      20 -0.63     D
3  Bullwinkle      Moose  412      80      18 -0.86     D
10       Mary    Rayburn  522      86      18 -0.18     C
2      Angela   Williams  600      99      22  0.92     A
7      Reuven    Ytzrhak  410      80      15 -1.05     F
```

Voila! Piece of cake!

There are many other ways to accomplish these tasks, but this code helps capture the flavor of these functions. Now it is time to look at control structures and user-written functions.

4.4 Control flow

In the normal course of events, the statements in an R program are executed sequentially from the top of the program to the bottom. However, there are times that we will want to execute some statements repetitively, while only executing other statements if certain conditions are met. This is where control-flow constructs come in.

R has the standard control structures you would expect to see in a modern programming language. First we will go through the constructs used for conditional execution, followed by the constructs used for looping.

In the syntax examples throughout this section

- `statement` is a single R statement or a compound statement (a group of R statements enclosed in curly braces { } and separated by semicolons).
- `cond` is an expression that resolves to TRUE or FALSE
- `expr` is a statement that evaluates to a number or character string
- `seq` is a sequence of numbers or character strings

After we discuss control-flow constructs, we will look at writing our functions.

4.4.1 Repetition and looping

Looping constructs repetitively execute a statement or series of statements until a condition is not true. These include the `for` and `while` structures.

FOR

The `for` loop executes a statement repetitively until a variable's value is no longer contained in the sequence `seq`. The syntax is

```
for (var in seq) statement
```

In following example

```
for (i in 1:10)  print("Hello")
```

the word Hello is printed 10 times.

WHILE

A while loop executes a statement repetitively until the condition is no longer TRUE. The syntax is

```
while (cond) statement
```

In our second example, the code

```
i = 10
while (i > 0) {print("Hello"); i <- i - 1}
```

once again prints the word Hello 10 times.

Looping in R can be inefficient and time consuming when processing the rows or column of large datasets. Whenever possible, it is better to use R's built-in numerical and character functions in conjunction with the `apply` family of functions.

4.4.2 Conditional execution

In conditional execution, a statement or statements is only executed if a specified condition is met. These constructs include `if-else`, `ifelse`, and `switch`.

IF-ELSE

The `if-else` control structure executes a statement if a given condition is TRUE. Optionally, a different statement is executed if the condition is FALSE. The syntax is

```
if (cond) statement
if (cond) statement1 else statement2
```

Here is an example.

```
if (score > 90) grade = 'A"

if (gender=="M") print("This is a man") else print("This is a woman")
```

In the first instance, the grade assignment is only made if the value of `score` is greater than 90. In the second instance, one of two statements is executed. If gender is equal to "M" then the first statement is executed. If not, the second statement is executed.

IFELSE

The ifelse construct is a compact version of the if-else construct we have seen above. The sytax is

```
ifelse(cond, statement1, statement2)
```

The first statement is executed if `cond` is TRUE. If `cond` is FALSE, the second statement is executed. Here is an example.

```
ifelse (score > 50, outcome <-  "passed", outcome <- "failed")
```

We use `ifelse` when we want to take a binary action.

SWITCH

Switch chooses statements based on the value of the expression. The syntax is

```
switch(expr, ...)
```

where `the ...` are statements tied to the possible outcome values of `expr`. It is easiest to understand how `switch` works with by looking at an example. An example is given in listing 4.8.

Listing 4.8 A switch example

```
> feelings <- c("sad", "afraid")
> for (i in feelings)
    print(
      switch(i,
         happy  = "I am glad you are happy",
         afraid = "There is nothing to fear",
         sad    = "Cheer up",
         angry  = "Calm down now"
      )
    )

[1] "Cheer up"
[1] "There is nothing to fear"
```

This is a silly example but shows the main features. We will see how to use `switch` a user-written functions in the next section.

4.5 User-written functions

One of R's great strengths is the user's ability to add functions. In fact, many of the functions in R are actually functions of existing functions. The structure of a function is given below.

```
myfunction <- function(arg1, arg2, ... ){
statements
return(object)
}
```

Objects in the function are local to the function. The object returned can be any data type from scalar to list. Let's take a look at an example.

We would like to have a function that calculates the central tendency and spread of data objects. The function should give us a choice between parametric (mean and standard deviation) and nonparametric (median and median absolute deviation) statistics. The results should be returned as a named list. Additionally, the user should have the choice of automatically printing the results or not. Unless unwise specified, the function's default behavior should be to calculate parametric statistics and not print the results. One solution is given in listing 4.9.

Listing 4.9 mystats: a user-written function for summary statistics

```
mystats <- function(x, parametric=TRUE, print=FALSE) {
  if (parametric) {
    center <- mean(x); spread <- sd(x)
  } else {
    center <- median(x); spread <- mad(x)
  }
  if (print & parametric) {
    cat("Mean=", center, "\n", "SD=", spread, "\n")
  } else if (print & !parametric) {
    cat("Median=", center, "\n", "MAD=", spread, "\n")
  }
  result <- list(center=center, spread=spread)
  return(result)
}
```

Now that we have our function, let's see it in action (listing 4.10).

Listing 4.10 mystats in action

```
# create some data (random sample from a normal distribution)
set.seed(1234)
x <- rnorm(500)
y <- mystats(x)

# no output is produced
# y$center is the mean (0.001838821)
# y$spread is the standard deviation (1.034814)

y <- mystats(x, parametric=FALSE, print=TRUE)

Median = -0.02070734
MAD = 1.000984

# y$center is the median (-0.02070734)
# y$spread is the median absolute deviation (1.000984)
```

Next, let's look at a user-written function that uses the switch construct (listing 4.11). This function gives the user a choice regarding the format of today's date. The long format is specified as the default.

Listing 4.11 mydate: a user-written function using switch

```
mydate <- function(type="long") {
  switch(type,
    long =  format(Sys.time(), "%A %B %d %Y"),
    short = format(Sys.time(), "%m-%d-%y"),
    cat(type, "is not a recognized type\n")        1
   )
}
```

Here is the function in action:

```
> mydate("long")

[1] "Saturday July 25 2009"

> mydate("short")

[1] "07-25-09"

> mydate()
```

```
[1] "Saturday July 25 2009"

> mydate("medium")

medium is not a recognized type
```

#1 Note that the `cat` function is only executed if the entered type does not match "long" or "short". It is usually a good idea to have an expression that catches user supplied arguments that have been entered incorrectly.

There are several functions than can help add error trapping and correction to your functions. You can use the function `warning` to generate a warning message, `message` to generate a diagnostic message, and `stop` to stop execution of the current expression and carry out an error action. See each function's online help for more details.

After creating our own functions, we may want to make them available in every session. Appendix B describes how to customize the R environment so that our functions are loaded automatically at start-up. We will look at additional examples of user-written functions in chapters 5 and 7.

We can accomplish a great deal using the basic techniques provided in this section. However, if you would like to explore the subtleties of function writing, or would like to write professional level code that you can distribute to other, I would recommend two excellent books:

- Venables, W. N., & Ripley, B. D. (2000). S Programming. New York: Springer.
- Chambers, J. M. (2008). Software for data analysis: Programming with R. New York: Springer.

Together, they provide a level of detail, and breadth of examples that goes well beyond what is possible in the current text.

Now that we have covered user-written functions, we will end this chapter with a discussion of data aggregation and reshaping.

4.6 Aggregation and restructuring

R provides a number of powerful methods for aggregating and reshaping data. When we aggregate data, we replace groups of observations with summary statistics based on those observations. When we reshape data, we alter the structure (rows and columns) determining how the data is organized. This section will describe a variety of methods for accomplishing these tasks.

In the next two sections, we will use the `mtcars` dataframe that is included with the base installation of R. This dataset, extracted from *Motor Trend* magazine (1974), describes the design and performance characteristics (number of cylinders, displacement, horsepower, mpg, etc.) for 34 automobiles. To learn more about the dataset, see `help(mtcars)`.

4.6.1 Transpose

The transpose (reversing rows and columns) is perhaps the simplest method of reshaping a dataset. Use the `t` function to transpose a matrix or a dataframe. In the later case, row names become variable (column) names. An example is presented in listing 4.12.

Listing 4.12 Transposing a dataset

```
> cars <- mtcars[1:5,1:4]                                     1
> cars

                   mpg cyl disp  hp
Mazda RX4         21.0   6  160 110
Mazda RX4 Wag     21.0   6  160 110
Datsun 710        22.8   4  108  93
Hornet 4 Drive    21.4   6  258 110
Hornet Sportabout 18.7   8  360 175

> t(cars)

     Mazda RX4 Mazda RX4 Wag Datsun 710 Hornet 4 Drive Hornet Sportabout
mpg         21            21       22.8           21.4              18.7
cyl          6             6        4.0            6.0               8.0
disp       160           160      108.0          258.0             360.0
hp         110           110       93.0          110.0             175.0
```

#1 We are using a subset of the mtcars dataset in order to conserve space on the page. We will see a more flexible way of transposing data when we look at the `shape` package later in this section.

4.6.2 Aggregating data

It is relatively easy to collapse data in R using one or more BY variables and a defined function. The format is

```
aggregate(x, by, FUN)
```

where `x` is the data object to be collapsed, `by` is a list of variables that will be crossed to form the new observations, and `FUN` is the scalar function used to calculate summary statistics that will make up the new observation values.

As an example, we will aggregate the `mtcars` data by number of cylinders and gears, returning means on each of the numeric variables (see listing 4.13).

Listing 4.13 Aggregating data

```
> options(digits=3)
> attach(mtcars)
> aggdata <-aggregate(mtcars, by=list(cyl,gear), FUN=mean, na.rm=TRUE)
> aggdata

  Group.1 Group.2  mpg cyl disp  hp drat   wt qsec  vs   am gear carb     1
1       4       3 21.5   4  120  97 3.70 2.46 20.0 1.0 0.00    3 1.00
```

```
  2        6        3 19.8   6  242 108 2.92 3.34 19.8 1.0 0.00    3 1.00
  3        8        3 15.1   8  358 194 3.12 4.10 17.1 0.0 0.00    3 3.08
  4        4        4 26.9   4  103  76 4.11 2.38 19.6 1.0 0.75    4 1.50
  5        6        4 19.8   6  164 116 3.91 3.09 17.7 0.5 0.50    4 4.00
  6        4        5 28.2   4  108 102 4.10 1.83 16.8 0.5 1.00    5 2.00
  7        6        5 19.7   6  145 175 3.62 2.77 15.5 0.0 1.00    5 6.00
  8        8        5 15.4   8  326 300 3.88 3.37 14.6 0.0 1.00    5 6.00
```

#1 In these results, Group.1 represents the number of cylinders (4,6, or 8) and Group.2 represents the number of gears (3, 4, or 5). For example, cars with 4 cylinders and 3 gears have get a mean of 21.5 miles per gallon (mpg).

When using the aggregate function, the by variables must be in a list (even if there is only one). The function specified can be any built-in or user provided function. This gives the aggregate command a great deal of power. But when it comes to power, nothing beats the `reshape` package.

4.6.3 The reshape package

The reshape package is a tremendously versatile approach to both restructuring and aggregating datasets. Because of this versatility, it can be a bit challenging to learn. We will go through the process slowly and use a very small dataset so that it is clear what is happening. Since `reshape` is not included in the standard installation of R, we will need to install it one time, using `install.packages("reshape")`.

Basically, we will "melt" data so that each row is a unique id-variable combination. Then we "cast" the melted data into any shape we desire. During the cast, we can aggregate the data with any function we wish.

The dataset we will be working with is in table 4.5.

Table 4.5 The original dataset (mydata)

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

In this dataset, the *measurements* are the values are the values in the last two columns (5, 6, 3, 5, 6, 1, 2, and 4). Each measurement is uniquely identified by a combination of id variables (in this case ID, Time, and whether the measurement is on X1 or X2). For example, the measured value 5 in the first row is uniquely identified by knowing that it is from observation (ID) 1, at Time 1, and on variable X1.

Melting

When we melt a dataset, we restructure it into a format where each measured variable is in its own row, along with the id variables needed to uniquely identify it. If we melt the data from table x.x, using the following code

```
   Library(reshape)
   md <- melt(mydata, id=(c("id", "time"))
```

we end up with the structure given in table 4.6.

Table 4.6 The melted dataset

ID	Time	Variable	Value
1	1	X1	5
1	2	X1	3
2	1	X1	6
2	2	X1	2
1	1	X2	6
1	2	X2	5
2	1	X2	1
2	2	X2	4

Note that we have specified the variables needed to uniquely identify each measurement (ID and Time) and that the variable indicating the measurement variable names (X1 or. X2) is created for us automatically.

Now that we have our data in a melted form, we can recast it into any shape, using the cast function.

Cast

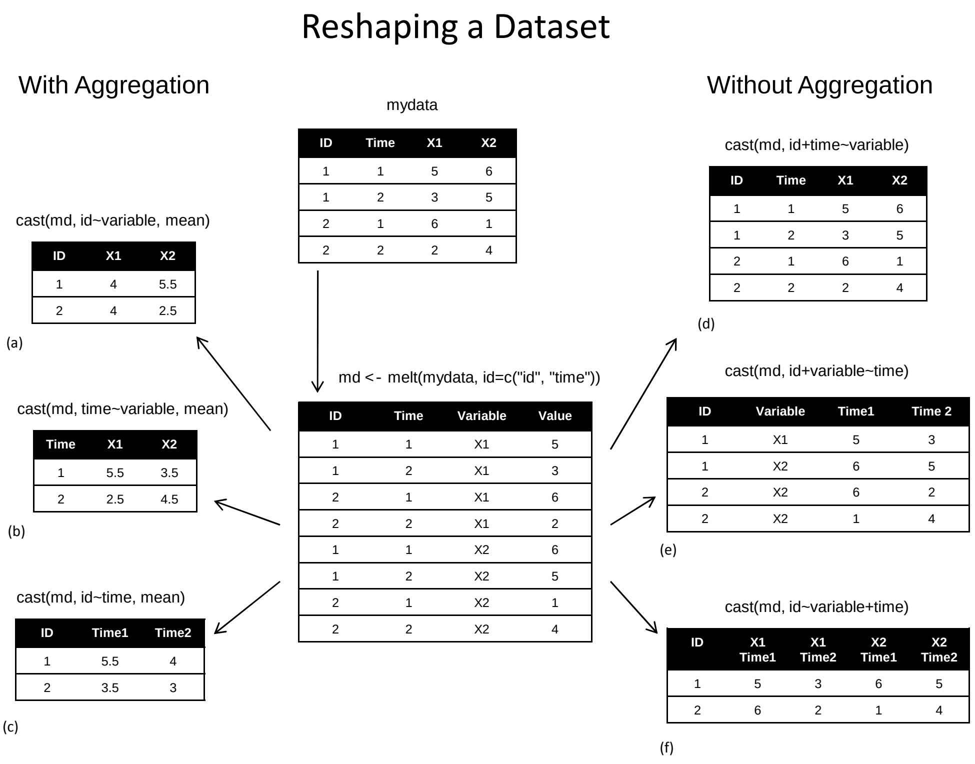
The `cast` function starts with melted data and reshapes using the it using a formula that we provide, and an (optional) function used to aggregate the data. The format is

```
   newdata <- cast(md, formula, FUN)
```

where `md` is the melted data, `formula` describes the desired end result, and FUN is the (optional) aggregating function. The formula takes the form

```
   rowvar1 + rowvar2 + … ~ colvar1 + colvar2 + …
```

In this formula, `rowvar1 + rowvar2 + ...` define the set of crossed variables that define the rows, while `colvar1 + colvar2 + ...` define the set of crossed variables that define the columns. This is easiest to see by looking at the examples in figure 4.1.



mydata

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

md <- melt(mydata, id=c("id", "time"))

ID	Time	Variable	Value
1	1	X1	5
1	2	X1	3
2	1	X1	6
2	2	X1	2
1	1	X2	6
1	2	X2	5
2	1	X2	1
2	2	X2	4

cast(md, id~variable, mean)

ID	X1	X2
1	4	5.5
2	4	2.5

(a)

cast(md, time~variable, mean)

Time	X1	X2
1	5.5	3.5
2	2.5	4.5

(b)

cast(md, id~time, mean)

ID	Time1	Time2
1	5.5	4
2	3.5	3

(c)

cast(md, id+time~variable)

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

(d)

cast(md, id+variable~time)

ID	Variable	Time1	Time 2
1	X1	5	3
1	X2	6	5
2	X2	6	2
2	X2	1	4

(e)

cast(md, id~variable+time)

ID	X1 Time1	X1 Time2	X2 Time1	X2 Time2
1	5	3	6	5
2	6	2	1	4

(f)

Figure 4.1 Reshaping data with the melt and cast functions

Since the formulas on the right side (d, e, and f) do not include a function, the data is simply reshaped. In contrast, the examples on the left side (a, b, and c) specify the mean as an aggregating function. Thus the data are not only reshaped but aggregated. For example (a) gives the means on X1 and X2 averaged over time for each observation. Example (b) gives the mean scores of X1 and X2 at Time 1 and Time 2, averaged over observations. In (c) we have the mean score for each observation at Time 1 and Time 2, averaged over X1 and X2. As you can see, the flexibility of the `reshape` functions is amazing.

4.7 Summary

In this chapter, we have reviewed dozens of mathematical, statistical, and probability functions that are useful for manipulating data. We have seen how to apply these functions to a wide range of data objects, including vectors, matrices, and dataframes. We have learned to use control-flow constructs for looping and branching to execute some statements repetitively and execute other statements only when certain conditions are met. We then had a chance to write our own functions and apply them to data. Finally, we have explored ways of collapsing, aggregating, and restructuring our data.

Now that we have gathered the tools we need to get our data into shape (no pun intended), we are ready to bid Part I goodbye for now, and enter the exciting world of data analysis! In the next chapter, we will begin to explore the many statistical methods available for turning data into understanding.

5 Basic Statistics

This Chapter covers:

- Descriptive statistics
- Frequency and contingency tables
- Correlations and covariances
- t-tests
- Nonparametric statistics

In previous chapters, we have seen how to import data into R and use a variety of functions to organize and transform the data into a useful format. Our next step will be to examine the distribution of each variable collected, followed by an exploration of the relationships among the variables two at a time. The goal of the present chapter is to describe how to accomplish these tasks in R.

First we will look at measures of location and scale for quantitative variables. Then we will look at frequency and contingency tables (and associated Chi-square tests) for categorical variables. Next, we will examine the various forms of correlation coefficients available for continuous and ordinal variables. Finally, we will turn to a study of group differences via both parametric (t-tests) and nonparametric (Mann-Whitney U test, Kruskal Wallis test) methods. Although our focus is on numerical results, accompanying graphical methods for visualizing these results will be described throughout.

5.1 What you need to know

In this chapter, we apply R programming techniques to the statistical methods typically taught in a first year undergraduate statistics course. If these methodologies are unfamiliar, two excellent references are McCall (2000), and Snedecor & Cochran (1989). Alternatively, there are many informative resources available online (e.g. Wikipedia) for each of the topics covered.

Whenever possible, we have separated the discussion of statistical methods and graphical methods into separate chapters, with extensive cross-references between them. This allows readers who are primarily interested in R as a language for creating graphs (and who may have a less extensive statistical background) to focus on these aspects of the software. For example, this chapter begins with statistical methods for summarizing data numerically, while chapter 6 describes ways of presenting this information graphically. However, in some later chapters (e.g., chapter 7), the statistical and graphical methods are so intertwined that a separation will not be possible.

5.2 Descriptive statistics

In this section, we will look at measures of central tendency, variability, and distribution shape for continuous variables. For illustrative purposes, we will use several of the variables from the with the Motor Trend Car Road Tests (`mtcars`) dataset we first saw in chapter 4.

```
> mt <- mtcars[c("mpg", "hp", "wt", "am")]
> head(mt)
                   mpg  hp   wt am
Mazda RX4         21.0 110 2.62  1
Mazda RX4 Wag     21.0 110 2.88  1
Datsun 710        22.8  93 2.32  1
Hornet 4 Drive    21.4 110 3.21  0
Hornet Sportabout 18.7 175 3.44  0
Valiant           18.1 105 3.46  0
```

In this dataset, miles per gallon (`mpg`), horse power (`hp`), and weight (`wt`) are quantitative variables, and transmission (`am`) is a dichotomous variable coded 0=automatic and 1=manual. We will use the `am` variable in section 5.2.2, when we look at generating descriptive statistics for subgroups.

5.2.1 A menagerie of methods

When it comes to calculating descriptive statistics, R has an embarrassment of riches. Let's start with functions that are included in the base installation. Then we will look at extensions that are available through the use of user-contributed packages.

In the base installation, we can use the `summary` function to obtain descriptive statistics. An example is presented in listing 5.1.

Listing 5.1 Descriptive statistics via `summary`

```
> summary(mt)
      mpg             hp              wt              am       
 Min.   :10.4   Min.   : 52.0   Min.   :1.51   Min.   :0.000  
 1st Qu.:15.4   1st Qu.: 96.5   1st Qu.:2.58   1st Qu.:0.000  
 Median :19.2   Median :123.0   Median :3.33   Median :0.000  
 Mean   :20.1   Mean   :146.7   Mean   :3.22   Mean   :0.406
 3rd Qu.:22.8   3rd Qu.:180.0   3rd Qu.:3.61   3rd Qu.:1.000
 Max.   :33.9   Max.   :335.0   Max.   :5.42   Max.   :1.000
```

The `summary` function provides the minimum, maximum, quartiles, and the mean. We can use the `sapply` or `apply` functions from chapter 4 to provide *any* descriptive statistics we choose. For the sapply function, the format is

```
sapply(x, FUN)
```

where `x` is our dataframe (or matrix) and `FUN` is an arbitrary function. Typical functions that we can plug in here are `mean`, `sd`, `var`, `min`, `max`, `med`, `range`, and `quantile`. The function `fivenum` returns Tukey's five number summary (minimum, lower-hinge, median, upper-hinge, and maximum).

Surprisingly, the base installation does not provide functions for skew and kurtosis, but we can add our own. An example that provides several descriptive statistics, including skew and kurtosis is given in listing 5.2.

Listing 5.2 Descriptive statistics via sapply

```
> skew <- function(x)(sum((x-mean(x))**3/sqrt(var(x))**3)/length(x))
> kurtosis <- function(x)(sum((x-mean(x))**4/var(x)**2)/length(x) - 3)
> descript <- function(x)(c(mean=mean(x), stdev=sd(x), [CA]
  skew=skew(x), kurtosis=kurtosis(x))
> sapply(mt, descript)

            mpg      hp      wt     am
mean     20.091 146.688  3.2172  0.406
stdev     6.027  68.563  0.9785  0.499
skew      0.611   0.726  0.4231  0.364
kurtosis -0.373  -0.136 -0.0227 -1.925              1
```

#1 For cars in this sample, the mean mpg is 20.1, with a standard deviation of 6.0. The distribution is skewed to the right (+0.61) and somewhat flatter than a normal distribution (-0.37). This will be most evident when we graph the data in chapter 6.

EXTENSIONS

Several user-contributed packages offer functions for descriptive statistics, including `Hmisc`, `pastecs`, and `psych`. Since these packages are not included in the base distribution, you will need to install them on first use (see section 1.4).

The `describe` function in the `Hmisc` package returns the number of variables and observations, number of missing and unique values, mean, quantiles, and five highest and lowest values. An example is provided in listing 5.3.

Listing 5.3 Descriptive statistics via describe (Hmisc package)

```
> library(Hmisc)
> describe(mt)
```

```
mt

 4  Variables      32  Observations
---------------------------------------------------------------------------
mpg
 n missing  unique   Mean    .05    .10    .25    .50    .75    .90   .95
32       0      25  20.09  12.00  14.34  15.43  19.20  22.80  30.09 31.30

lowest : 10.4 13.3 14.3 14.7 15.0, highest: 26.0 27.3 30.4 32.4 33.9
---------------------------------------------------------------------------
hp
 n missing  unique   Mean    .05    .10    .25    .50    .75    .90    .95
32       0      22  146.7  63.65  66.00  96.50 123.00 180.00 243.50 253.55

lowest :  52  62  65  66  91, highest: 215 230 245 264 335
---------------------------------------------------------------------------
wt
 n missing  unique   Mean    .05    .10    .25    .50    .75    .90   .95
32       0      29  3.217  1.736  1.956  2.581  3.325  3.610  4.048 5.293

lowest : 1.513 1.615 1.835 1.935 2.140, highest: 3.845 4.070 5.250 5.345
5.424
---------------------------------------------------------------------------
am
 n missing  unique     Sum    Mean
32       0       2      13  0.4062
---------------------------------------------------------------------------
```

The `pastecs` package offers a function named `stat.desc` that provides a wide range of descriptive statistics. The format is

```
stat.desc(x, basic=TRUE, desc=TRUE, norm=FALSE, p=0.95)
```

If `basic=TRUE` (the default), the number of values, null values, missing values, minimum, maximum, range and sum are provided. If `desc=TRUE` (also the default), the median, mean, standard error of the mean, 95% confidence interval for the mean, variance, standard deviation, and coefficient of variation are also provided. Finally, if `norm=TRUE` (not the default), normal distribution statistics are returned, including skewness and kurtosis (and their statistical significance), and the Shapiro-Wilks test of normality. The p value option is used to calculate the confidence interval for the mean (.95 by default). An example is given in listing 5.4.

Listing 5.4 Descriptive statistics via `stat.desc` (`pastecs` package)

```
> library(pastecs)
> stat.desc(mt)
                 mpg        hp       wt        am
nbr.val        32.00    32.000   32.000  32.0000
nbr.null        0.00     0.000    0.000  19.0000
nbr.na          0.00     0.000    0.000   0.0000
min            10.40    52.000    1.513   0.0000
```

```
max            33.90  335.000   5.424  1.0000
range          23.50  283.000   3.911  1.0000
sum           642.90 4694.000 102.952 13.0000
median         19.20  123.000   3.325  0.0000
mean           20.09  146.688   3.217  0.4062
SE.mean         1.07   12.120   0.173  0.0882
CI.mean.0.95    2.17   24.720   0.353  0.1799
var            36.32 4700.867   0.957  0.2490
std.dev         6.03   68.563   0.978  0.4990
coef.var        0.30    0.467   0.304  1.2283
```

As if this isn't enough, the `psych` package also has a function called `describe` that provides the number of non-missing observations, mean, standard deviation, median, trimmed mean, median absolute deviation, minimum, maximum, range, skew, kurtosis, and standard error of the mean. An example is given in listing 5.5.

Listing 5.5 Descriptive statistics via `describe` (psych package)

```
> library(psych)

Attaching package: 'psych'

        The following object(s) are masked from package:Hmisc :
         describe

> describe(mt)

    n   mean    sd median trimmed   mad   min    max  range skew kurtosis
mpg 32  20.09  6.03  19.20   19.70  5.41 10.40  33.90  23.50 0.61    -0.37
hp  32 146.69 68.56 123.00  141.19 77.10 52.00 335.00 283.00 0.73    -0.14
wt  32   3.22  0.98   3.33    3.15  0.77  1.51   5.42   3.91 0.42    -0.02
am  32   0.41  0.50   0.00    0.38  0.00  0.00   1.00   1.00 0.36    -1.92
       se
mpg  1.07
hp  12.12
wt   0.17
am   0.09
```

I told you that it was an embarrassment of riches but you didn't believe me.

A NOTE ON MASKING

In the examples above, the packages psych and Hmisc both provided functions named `describe`. How does R know which one to use? Simply put, the package last loaded takes precedence, as seen in listing 5.5. Here, `psych` is loaded after `Hmisc`, and a message is printed indicating that the describe function in `Hmisc` is masked by the function in `psych`. When you type in the describe function and R searches for it, it comes to the psych package first and executes it. If we wanted the `Hmisc` version instead, we could have typed `Hmisc::describe(mt)`. The function is still there. We just have to give R more information to find it.

Now that we know how to generate descriptive statistics for the data as a whole, let's review how to obtain statistics by subgroups of the data.

5.2.2 Descriptive statistics by group

When comparing groups of individuals or observations, our focus is usually on the descriptive statistics of each group, rather than the total sample. Again, there are several ways to accomplish in R.

In chapter 4, we discussed methods of aggregating data. We could use the `aggregate` function (section 4.5.2) to obtain descriptive statistics by group (listing 5.6).

Listing 5.6 Descriptive statistics by group via `aggregate`

```
> aggregate(mt,by=list(mt$am),mean)
  Group.1  mpg  hp   wt am
1       0 17.1 160 3.77  0
2       1 24.4 127 2.41  1

> aggregate(mt, by=list(mt$am),sd)
  Group.1  mpg   hp    wt am
1       0 3.83 53.9 0.777  0
2       1 6.17 84.1 0.617  0
```

Unfortunately, `aggregate` only allows us to use single value functions such as mean, standard deviation, and the like in each call. It will not return several statistics at once. For that task, we can use the `by` function. The format is

```
by(data, INDICES, FUN)
```

where data is a dataframe or matrix, INDICES is a factor or list of factors that define the groups, and FUN is an arbitrary function. An example is given in listing 5.6.

Listing 5.6 Descriptive statistics by group via by

```
> by(mt,mt$am,function(x)(c(mean=mean(x),sd=sd(x))))

mt$am: 0
mean.mpg  mean.hp  mean.wt  mean.am   sd.mpg    sd.hp    sd.wt    sd.am
  17.147  160.263    3.769    0.000    3.834   53.908    0.777    0.000
------------------------------------------------------------
mt$am: 1
mean.mpg  mean.hp  mean.wt  mean.am   sd.mpg    sd.hp    sd.wt    sd.am
  24.392  126.846    2.411    1.000    6.167   84.062    0.617    0.000
```

EXTENSIONS

The `doBy` package and the `psyc` package also provide functions for descriptive statistics by group. Again, they are not distributed in the base installation, and must be installed before first use. The `summaryBy` function in the `doBy` package has the format

```
summaryBy(formula, data=dataframe, FUN=)
```

The formula takes the form

```
var1+var2+var3+...+varN~groupvar1+groupvar2+…+groupvarN
```

where variables on the left of the ~ are the numeric variables to be analyze and variables on the right are categorical grouping variables. An example using the `descript` function we created in section 5.2.1 is given in listing 5.7.

Listing 5.7 Summary statistics by group via `summaryBy` (doBy package)

```
> library(doBy)
> summaryBy(mpg+hp+wt~am,data=mt,FUN=descript)

  am mpg.mean mpg.stdev mpg.skew mpg.kurtosis hp.mean hp.stdev hp.skew
1  0     17.1      3.83   0.0140       -0.803     160     53.9 -0.0142
2  1     24.4      6.17   0.0526       -1.455     127     84.1  1.3599
  hp.kurtosis wt.mean wt.stdev wt.skew wt.kurtosis
1      -1.210    3.77    0.777   0.976       0.142
2       0.563    2.41    0.617   0.210      -1.174
```

The `describe.by` function provided by the `psych` package provides the same descriptive statistics as `describe`, stratified by one or more grouping variables (see listing 5.8).

Listing 5.8 Summary statistics by group via `describe.by` (psych package)

```
> library(psych)
> describe.by(mt, mt$am)
$`0`
     n   mean    sd median trimmed   mad   min    max  range  skew kurtosis
mpg 19  17.15  3.83  17.30   17.12  3.11 10.40  24.40  14.00  0.01    -0.80
hp  19 160.26 53.91 175.00  161.06 77.10 62.00 245.00 183.00 -0.01    -1.21
wt  19   3.77  0.78   3.52    3.75  0.45  2.46   5.42   2.96  0.98     0.14
am  19   0.00  0.00   0.00    0.00  0.00  0.00   0.00   0.00   NaN      NaN
       se
mpg  0.88
hp  12.37
wt   0.18
am   0.00

$`1`
     n   mean    sd median trimmed   mad   min    max  range skew kurtosis
mpg 13  24.39  6.17  22.80   24.38  6.67 15.00  33.90  18.90 0.05    -1.46
hp  13 126.85 84.06 109.00  114.73 63.75 52.00 335.00 283.00 1.36     0.56
wt  13   2.41  0.62   2.32    2.39  0.68  1.51   3.57   2.06 0.21    -1.17
am  13   1.00  0.00   1.00    1.00  0.00  1.00   1.00   0.00 NaN       NaN
       se
mpg  1.71
hp  23.31
wt   0.17
am   0.00
```

Unlike the previous example, the `describe.by` function does not allow us to specify an arbitrary function, so it is less generally applicable. If there is more than one grouping variable, we can write them as `list(groupvar1, groupvar2, … , groupvarN)`. However, this will only work if there are no empty cells when the grouping variables are crossed.

5.2.3 Visualizing results

Numerical summaries of a distribution's characteristics are important, but are no substitute for a visual representation. For quantitative variables we have histograms (section 6.2), density and dot plots (section 11.1), and boxplots (section 11.2). They can provide insights that are easily missed by reliance on a small set of descriptive statistics.

The functions we have considered so far provide summaries of quantitative variables. The functions in the next section allow us to examine the distributions of categorical variables.

5.3 Frequency and contingency tables

In this section we look at frequency and contingency tables from categorical variables, along with tests of independence, measures of association, and methods for graphically displaying results. We will be using functions in the basic installation, along with functions from the `vcd` and `gmodels` package. In the following examples, assume that A, B, and C represent categorical variables.

The first example comes from the `Arthritis` dataset included with the `vcd` package. The data are from Kock & Edward (1988) and represents a double-blind clinical trial of new treatments for rheumatoid arthritis. Here are the first few observations:

```
> library(vcd)
> head(Arthritis)

  ID Treatment  Sex Age Improved
1 57   Treated Male  27     Some
2 46   Treated Male  29     None
3 77   Treated Male  30     None
4 17   Treated Male  32   Marked
5 36   Treated Male  46   Marked
6 23   Treated Male  58   Marked
```

Treatment (Placebo, Treated), Sex (Male, Female), and Improved (None, Some, Marked), are all categorical factors. In the next section, we will look at creating frequency and contingency tables (cross-classifications) for the data.

5.3.1 Generating frequency tables

R provides several methods for creating frequency and contingency tables. We will look at simple frequencies, followed by two-way contingency tables, and multi-way contingency tables.

ONE WAY TABLES

We can generate simple frequency counts using the `table` function. For example

```
> attach(Arthritis)
> table(Improved)

Improved
  None   Some Marked
    42    14    28
```

TWO WAY TABLES

For two-way tables, the format for the `table` function is given below.

```
attach(mydata)
mytable <- table(A,B) # A will be rows, B will be columns
mytable               # print table
```

Alternatively, the `xtabs` function allows us to create a contingency table using formula style input. The format is

```
mytable <- xtabs(~A+B, data=mydata)
```

In general, the variables to be cross classified appear on the right of the formula (i.e., to the right of the ~) separated by + signs. If a variable is included on the left side of the formula, it is assumed to be a vector of frequencies (useful if the data have already been tabulated).

We can generate tables of proportions using the `prop.table` function, and marginal frequencies using `margin.table`. The formats are given below.

```
margin.table(mytable, 1) # A frequencies (summed over B)
margin.table(mytable, 2) # B frequencies (summed over A)

prop.table(mytable)    # cell proportions
prop.table(mytable, 1) # row proportions
prop.table(mytable, 2) # column proportion
```

As you can see, the first step is to create a table using the `table` or `xtabs` function. We can then manipulate it using the other functions. Here is an example (listing 5.9).

Listing 5.9 Two-way contingency table

```
> attach(Arthritis)
> mytable <- table(Treatment,Improved)
> mytable

         Improved                     A
Treatment None Some Marked            A
  Placebo   29    7      7            A
  Treated   13    7     21            A

> margin.table(mytable,1)

Treatment                             B
Placebo Treated                       B
     43      41                       B

> margin.table(mytable,2)

Improved                              C
  None   Some Marked                  C
    42     14     28                  C


> prop.table(mytable)

         Improved                     D
Treatment   None   Some Marked        D
  Placebo 0.3452 0.0833 0.0833        D
  Treated 0.1548 0.0833 0.2500        D

> prop.table(mytable,1)

         Improved                     E
Treatment  None  Some Marked          E
  Placebo 0.674 0.163  0.163          E
  Treated 0.317 0.171  0.512          E

> prop.table(mytable,2)

         Improved                     F
Treatment  None  Some Marked          F
  Placebo 0.690 0.500  0.250          F
  Treated 0.310 0.500  0.750          F

A cell frequencies
B row marginals
C column marginals
D cell proportions
E row proportions
F column proportions
```

Looking at the row proportions, we can see that 16% of patients receiving the placebo had some improvement. Looking at the column proportions, we see that 75% of those patients with marked improvement were in the Treated group.

THE TABLE FUNCTION IGNORES MISSING VALUES

To include NA as a valid category in the frequency counts, include the table option `exclude=NULL` if the variable is a vector. If the variable is a factor, we have to replace it with a new factor via `newfactor <- factor(oldfactor, exclude=NULL)`.

A third method for creating two-way tables is the `CrossTable` function in the `gmodels` package. The `CrossTable` function produces two-way tables modeled after PROC FREQ in SAS or CROSSTABS in SPSS. An example is given in listing 5.10.

Listing 5.10 Two-way table using `CrossTable`

```
> library(gmodels)
> CrossTable(Arthritis$Treatment, Arthritis$Improved)


   Cell Contents
|-------------------------|
|                       N |
| Chi-square contribution |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  84


                    | Arthritis$Improved
Arthritis$Treatment |      None |      Some |    Marked | Row Total |
--------------------|-----------|-----------|-----------|-----------|
            Placebo |        29 |         7 |         7 |        43 |
                    |     2.616 |     0.004 |     3.752 |           |
                    |     0.674 |     0.163 |     0.163 |     0.512 |
                    |     0.690 |     0.500 |     0.250 |           |
                    |     0.345 |     0.083 |     0.083 |           |
--------------------|-----------|-----------|-----------|-----------|
            Treated |        13 |         7 |        21 |        41 |
                    |     2.744 |     0.004 |     3.935 |           |
                    |     0.317 |     0.171 |     0.512 |     0.488 |
                    |     0.310 |     0.500 |     0.750 |           |
                    |     0.155 |     0.083 |     0.250 |           |
--------------------|-----------|-----------|-----------|-----------|
       Column Total |        42 |        14 |        28 |        84 |
                    |     0.500 |     0.167 |     0.333 |           |
--------------------|-----------|-----------|-----------|-----------|
```

The `CrossTable` function has options to report percentages (row, column, cell), specify decimal places, produce Chi-square, Fisher, and McNemar tests of independence, report expected and residual values (Pearson, standardized, adjusted standardized), include missing values as valid, annotate with row and column titles, and format as SAS or SPSS style output! See `help(CrossTable)` for details. If we have more than two categorical variables, we will be interested in multidimensional tables, which are considered next.

MULTIDIMENSIONAL TABLES

Both the `table` and `xtabs` function can be used generate multidimensional tables based on three or more categorical variables. The `margin.table` and `prop.table` functions extend naturally to more than two dimensions. The `ftable` function can be used to print multidimensional tables in a compact and attractive manner. An example is given in listing 5.11.

Listing 5.11 Three-way contingency table

```
> attach(Arthritis)
> mytable <- table(Treatment, Improved, Sex)             1
> ftable(mytable)                                        2

                   Sex Female Male
Treatment Improved
Placebo   None             19   10
          Some              7    0
          Marked            6    1
Treated   None              6    7
          Some              5    2
          Marked           16    5

> margin.table(mytable, 1)                               3

Treatment
Placebo Treated
     43      41

> margin.table(mytable, 2)                               4
Improved
  None   Some Marked
    42     14     28

> margin.table(mytable, c(1:2))                          5
         Improved
Treatment None Some Marked
  Placebo   29    7      7
  Treated   13    7     21

> ftable(prop.table(mytable, c(1,2)))                    6
                   Sex Female  Male
Treatment Improved
Placebo   None          0.655 0.345
          Some          1.000 0.000
          Marked        0.857 0.143
Treated   None          0.462 0.538
          Some          0.714 0.286
          Marked        0.762 0.238
```

We could have also written #1 as

`mytable <- xtabs(~Treatment+Improved+Sex,data=Arthritis)`

and obtained the same results. The code in #2 produces cell frequencies for the 3-way classification, while #3 and #4 produce the marginal frequencies for Treatment and

Improved respectively. The code in #5 produces the marginal frequencies for the Treatment x Improved classification, summed over Sex. The proportion of men and women for each Treatment x Improved combination is provided in #6. In general, the proportions will add to one over the indices not included in the `prop.table` call (the 3rd index or Sex in this case).

While contingency tables tell us the frequency or proportions of cases with each combination of the variables that comprise the table, our interests usually extend to whether the variables in the table are related or independent. This is the subject of the next section.

5.3.2 Tests of independence

R provides several methods of testing the independence of the categorical variables. The three tests described in this section are the Chi-square test of independence, the Fisher exact test, and the Mantel-Haenszel test. A fourth approach, log-linear models, will be discussed in chapter 14.

CHI-SQUARE TEST OF INDEPENDENCE

The function `chisquare.test` can be applied to a two-way table in order to produce a Chi-square test of independence of the row and column variables. An example is provided in listing 5.12.

Listing 5.12 Chi-square test of independence

```
> mytable <- table(Treatment, Improved)
> chisq.test(mytable)

        Pearson's Chi-squared test

data:  mytable
X-squared = 13.1, df = 2, p-value = 0.001463        1

> mytable <- table(Improved, Sex)
> chisq.test(mytable)

        Pearson's Chi-squared test

data:  mytable
X-squared = 4.84, df = 2, p-value = 0.0889           2

Warning message:
In chisq.test(mytable) : Chi-squared approximation may be incorrect
```

From the results above, there appears to be a relationship between treatment received and level of improvement (p<.01) #1, but not between patient sex and improvement (p>.05) #2.

FISHER EXACT TEST

We can produce a Fisher's exact test via the `fisher.test` function. Fisher's exact test evaluates the null hypothesis of independence of rows and columns in a contingency table with fixed marginals. The format is `fisher.test(mytable)`, where `mytable` is a two-way table. An example is given in listing 5.13.

Listing 5.13 Fisher exact test

```
> mytable <- table(Treatment,Improved)
> fisher.test(mytable)

        Fisher's Exact Test for Count Data

data:  mytable
p-value = 0.001393
alternative hypothesis: two.sided
```

In contrast to many statistical packages, the `fisher.test` function can be applied to any rxc table, not just a 2x2 table.

COCHRAN-MANTEL-HAENSZEL TEST

The `mantelhaen.test` function provides a Cochran-Mantel-Haenszel chi-squared test of the null hypothesis that two nominal variables are conditionally independent in each stratum of a third variable. Listing 5.14 tests the hypothesis that Treatment and Improved is independent within each level Sex. The test assumes that there is no three-way (Treatment x Improved x Sex) interaction.

Listing 5.13 Cochran-Mantel-Haenszel test

```
> mytable <- table(Treatment, Improved, Sex)
> mantelhaen.test(mytable)

        Cochran-Mantel-Haenszel test

data:  mytable
Cochran-Mantel-Haenszel M^2 = 14.6, df = 2, p-value = 0.0006647
```

The results suggest that the treatment received and the improvement reported is not independent within each level of sex (i.e., treated individuals improved more than those receiving placebos when controlling for sex).

5.3.3 Measures of association

The significance tests in the previous section evaluate whether or not there is sufficient evidence to reject a null hypothesis of independence between variables. If we can reject the null hypothesis, our interests turn naturally to measures of association in order to gauge the strength of the relationships present. The `assocstats` function in the `vcd` package can be used to calculate the phi coefficient, contingency coefficient, and Cramer's V for a two-way table. An example of `assocstats` is given in listing 5.14.

Listing 5.14 Measures of association for a two-way table

```
> library(vcd)
> attach(Arthritis)
> mytable <- table(Treatment, Improved)
> assocstats(mytable)
```

```
                     X^2 df  P(> X^2)
Likelihood Ratio 13.530  2 0.0011536
Pearson          13.055  2 0.0014626

Phi-Coefficient   : 0.394
Contingency Coeff.: 0.367
Cramer's V        : 0.394
```

In general, larger magnitudes indicated stronger associations. The vcd package also provides a kappa function that can calculate Cohen's kappa and weighted kappa for a confusion matrix (for example, the degree of agreement between two judges classifying a set of objects into categories).

5.3.4 Visualizing results

R has mechanisms for visually exploring the relationships among categorical variables that go well beyond those found in most other statistical platforms. We typically use bar charts to visualize frequencies in one dimension (section 6.3). The vcd package has excellent functions for visualizing relationships among categorical variables in multi-dimensional datasets using mosaic and association plots (section 15.6). Finally, correspondence analysis functions in the ca package allow us to visually explore relationships between rows and columns in contingency tables (section 14.4) using various geometric representations. Feel free to jump to those sections at any time!

5.3.5 Converting tables to flat files

We will end this section with a topic that is rarely covered in books on R, but that can be very useful. What happens if we have a table, but need the original raw data file? For example, we have

```
                   Sex Female Male
Treatment Improved
Placebo   None             19   10
          Some              7    0
          Marked            6    1
Treated   None              6    7
          Some              5    2
          Marked           16    5
```

but we need:

```
  ID Treatment  Sex Age Improved
1 57   Treated Male  27     Some
2 46   Treated Male  29     None
3 77   Treated Male  30     None
4 17   Treated Male  32   Marked
5 36   Treated Male  46   Marked
6 23   Treated Male  58   Marked
[78 more rows go here]
```

There are many statistical functions in R that expect the later format, rather than the former. We can use the function provided listing 5.15 to convert an R table back into a flat data file.

Listing 5.15 Converting a table into a flat file via `table2flat`

```
table2flat <- function(mytable) {
  df <- as.data.frame(mytable)
  rows  <- dim(df)[1]
  cols  <- dim(df)[2]
  x <- NULL
  for (i in 1:rows){
    for (j in 1:df$Freq[i]){
      row <- df[i,c(1:(cols-1))]
      x <- rbind(x,row)
    }
  }
  row.names(x)<-c(1:dim(x)[1])
  return(x)
}
```

This function takes an R table (with any number of dimensions) and returns a dataframe in flat file format. We can also use this function to input tables from published studies. For example, let's say that we came across table 5.1 in a journal and we wanted to save it into R as a flat file.

Table 5.1 Contingency table for treatment vs. improvement from the Arthritis dataset

Treatment	Improved		
	None	Some	Marked
Placebo	29	7	7
Treated	13	17	21

Listing 5.16 describes a method that would do the trick.

Listing 5.16 Using the `table2flat` function from published data

```
> treatment <- rep(c("Placebo", "Treated"), 3)
> improved <- rep(c("None","Some","Marked"), each=2)
> Freq <- c(29,13,7,7,7,21)
> mytable <- as.data.frame(cbind(treatment,improved, Freq))
> mydata <- table2flat(mytable)
> head(mydata)

  treatment improved
1   Placebo     None
2   Placebo     None
3   Placebo     None
```

```
   4   Treated     None
   5   Placebo     Some
   6   Placebo     Some
   [12 more rows go here]
```

This will end our discussion of contingency tables, until we take up more advanced topics in chapter 14. Next, let's look at various types of correlation coefficients.

5.4 Correlations

Correlation coefficients are used to explore relationships among quantitative variables. In this section we will look at a variety of correlation coefficients, as well as tests of significance. We will use is the `state.x77` dataset available in the base R installation. It provides data on the population, income, illiteracy rate, life expectancy, murder rate, and high school graduation rate for the 50 US states in 1977. There are also temperature and land area measures, but we'll drop them to save space. Use `help(state.x77)` to learn more about the file. In addition to the base installation, we will be using the `psych` and `ggm` packages.

5.4.1 Type of correlations

R can produce a variety of correlation coefficients, including Pearson, Spearman, Kendall, partial, polychoric, and polyserial. Let's look at each in turn.

PEARSON, SPEARMAN, AND KENDALL CORRELATIONS

The Pearson product moment correlation assesses the degree of linear relationship between two quantitative variables. Spearman's Rank Order correlation coefficient assesses the degree of relationship between two rank ordered variables. Kendall's Tau is also a nonparametric measure of rank correlation.

The `cor` function produces these correlation coefficients, while the `cov` function provides covariances. There are many options, but a simplified format for producing correlations is

```
cor(x, use= , method= )
```

where the options are described in table 5.2.

Table 5.2 cor/cov options

Option	Description
`x`	Matrix or dataframe
`use`	Specifies the handling of missing data. The options are `all.obs` (assumes no missing data, missing data will produce an error), `everything` (any correlation involving a case with missing values will be set to missing), `complete.obs` (listwise deletion), and `pairwise.complete.obs` (pairwise deletion)
`method`	Specifies the type of correlation. The options are `pearson`, `spearman`, or

kendall.

The default options are use="everything" and method="pearson". An example is given in listing 5.17.

Listing 5.17 Covariances and Correlations

```
> states<- state.x77[,1:6]
> cov(states)

           Population Income Illiteracy Life Exp  Murder  HS Grad
Population   19931684 571230    292.868 -407.842 5663.52 -3551.51
Income         571230 377573   -163.702  280.663 -521.89  3076.77
Illiteracy        293   -164      0.372   -0.482    1.58    -3.24
Life Exp         -408    281     -0.482    1.802   -3.87     6.31
Murder           5664   -522      1.582   -3.869   13.63   -14.55
HS Grad         -3552   3077     -3.235    6.313  -14.55    65.24

> cor(states)

           Population Income Illiteracy Life Exp Murder HS Grad
Population     1.0000  0.208      0.108   -0.068  0.344 -0.0985
Income         0.2082  1.000     -0.437    0.340 -0.230  0.6199
Illiteracy     0.1076 -0.437      1.000   -0.588  0.703 -0.6572
Life Exp      -0.0681  0.340     -0.588    1.000 -0.781  0.5822
Murder         0.3436 -0.230      0.703   -0.781  1.000 -0.4880
HS Grad       -0.0985  0.620     -0.657    0.582 -0.488  1.0000

> cor(states,method="spearman")

           Population Income Illiteracy Life Exp Murder HS Grad
Population      1.000  0.125      0.313   -0.104  0.346  -0.383
Income          0.125  1.000     -0.315    0.324 -0.217   0.510
Illiteracy      0.313 -0.315      1.000   -0.555  0.672  -0.655
Life Exp       -0.104  0.324     -0.555    1.000 -0.780   0.524
Murder          0.346 -0.217      0.672   -0.780  1.000  -0.437
HS Grad        -0.383  0.510     -0.655    0.524 -0.437   1.000
```

The first call produces the variances and covariances. The second provides Pearson Product Moment correlation coefficients, while the third produces Spearman Rank Order correlation coefficients. We can see, for example, that there is a strong positive correlation between income and high school graduation rate, and a strong negative correlation between illiteracy rates and life expectancy.

Notice, that we get square matrices by default (all variables crossed with all other variables). We can also produce non-square matrices. An example is given in listing 5.18.

Listing 5.18 Correlating two sets of variables

```
> x <- states[,c("Population","Income","Illiteracy","HS Grad")]
> y <- states[,c("Life Exp", "Murder")]
> cor(x,y)
```

```
              Life Exp Murder
Population   -0.068  0.344
Income        0.340 -0.230
Illiteracy   -0.588  0.703
HS Grad       0.582 -0.488
```

This version of the function is particularly useful when our interest centers on the relationships between one set of variables and another. Notice that the results printed above do not tell us if the correlations differ significantly from zero. For that, we need tests of significance (section 5.4.2).

PARTIAL CORRELATIONS

A partial correlation is a correlation between two quantitative variables, controlling for one or more other quantitative variables. The `pcor` function in the `ggm` package can be used to provide partial correlation coefficients. The `ggm` package is not installed by default, so be sure to install it on first use. The format is

```
pcor(u, S)
```

where `u` is a vector of numbers, with the first two numbers being the indices of the variables to be correlated, and the remaining numbers being the indices of the conditioning variables (i.e., the variables being partialled out). S is the covariance matrix among the variables. An example (listing 5.19) will clarify this.

Listing 5.19 Partial correlations with the pcor function [ggm package]

```
> library(ggm)
> # partial correlation of population and murder rate, controlling
> # for income, illiteracy rate, and HS graduation rate
> pcor(c(1,5,2,3,6), cov(states))

[1] 0.346              1
```

#1 0.346 is the correlation between population and murder rate, controlling for the influence of income, illiteracy rate, and HS graduation rate. The use of partial correlations is common in the social sciences.

OTHER TYPES OF CORRELATIONS

The `hetcor` function in the `polycor` package can compute a heterogeneous correlation matrix containing Pearson product-moment correlations between numeric variables, polyserial correlations between numeric and ordinal variables, polychoric correlations between ordinal variables, and tetrachoric correlations between two dichotomous variables. Polyserial, polychoric, and tetrachoric correlations assume that the ordinal or dichotomous variables are derived from underlying normal distributions. See the documentation that accompanies this package for more information.

5.4.2 Testing correlations for significance

Once we have generated correlation coefficients, how do we test them for statistical significance? The typical null hypothesis is no relationship (i.e., the correlation in the population is zero). We can use the `cor.test` function to test an individual Pearson, Spearman, and Kendall correlation coefficient. A simplified format is

```
cor.test(x, y,  alternative = , method = )
```

where `alternative` specifies a two-tailed or one-tailed test (`"two.side"`, `"less"`, or `"greater"`) and `method` specifies the type of correlation (`"pearson"`, `"kendall"`, or `"spearman"`). An example is given in listing 5.20.

Listing 5.20 Testing a correlation coefficient for significance

```
> cor.test(states[,3],states[,5])

        Pearson's product-moment correlation

data:  states[, 3] and states[, 5]
t = 6.85, df = 48, p-value = 1.258e-08
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.528 0.821
sample estimates:
  cor
0.703
```

This code tests the null hypothesis that the Pearson correlation between life expectancy and murder rate is zero. We see that the sample correlation of 0.70 is large enough to reject the null hypothesis at any reasonable alpha level (here p < .00000001).

Unfortunately, we can only test one correlation at a time using `cor.test`. Luckily, the `corr.test` function provided in the `psych` package allows us to go farther. The `corr.test` function produces correlations and significance levels for matrices of Pearson, Spearman, orKendall correlations. An example is given in listing 5.21.

Listing 5.21 Correlation matrix and tests of significance via `corr.test` [psych package]

```
> library(psych)
> corr.test(states,use="complete")

Call:corr.test(x = states, use = "complete")
Correlation matrix
           Population Income Illiteracy Life Exp Murder HS Grad
Population       1.00   0.21       0.11    -0.07   0.34   -0.10    1
Income           0.21   1.00      -0.44     0.34  -0.23    0.62
Illiteracy       0.11  -0.44       1.00    -0.59   0.70   -0.66
Life Exp        -0.07   0.34      -0.59     1.00  -0.78    0.58
Murder           0.34  -0.23       0.70    -0.78   1.00   -0.49
HS Grad         -0.10   0.62      -0.66     0.58  -0.49    1.00
```

```
Sample Size
[1] 50
Probability value
           Population Income Illiteracy Life Exp Murder HS Grad
Population       0.00   0.15       0.46     0.64   0.01     0.5      1
Income           0.15   0.00       0.00     0.02   0.11     0.0
Illiteracy       0.46   0.00       0.00     0.00   0.00     0.0
Life Exp         0.64   0.02       0.00     0.00   0.00     0.0
Murder           0.01   0.11       0.00     0.00   0.00     0.0
HS Grad          0.50   0.00       0.00     0.00   0.00     0.0
```

The `use=` options can be `"pairwise"` or `"complete"` (for pairwise or listwise deletion of missing values respectively). The `method=` option is `"pearson"` (the default), `"spearman"`, or `"kendall"`. #1 Here we see that the correlation between population size and high school graduation rate (-0.10) is not significantly different from zero (p=0.5).

OTHER TESTS OF SIGNIFICANCE

In section 5.4.1 we looked at partial correlations. The `pcor.test` function in the `psych` package can be used to test the conditional independence of two variables controlling for one or more additional variables, assuming multivariate normality. The format is

```
pcor.test(r, q, n)
```

where `r` is the partial correlation produced by the `pcor` function, `q` is the number of variables being controlled, and `n` is the sample size.

Before leaving this topic, it should be mentioned that the `r.test` function in the `psych` package also provides a number of useful significance tests. The function can be used to test the significance of a correlation coefficient, the difference between two independent correlations, the difference between two dependent correlations sharing one single variable, and the difference between two dependent correlations based on completely different variables. See `help(r.test)` for details.

5.4.3 Visualizing correlations

The bivariate relationships underlying correlations can be visualized through scatterplots and scatterplot matrices (section 11.3), while corrgrams (section 15.7) provide a unique and powerful method for comparing a large numbers of correlation coefficients in a meaningful way.

5.5 Comparing Groups

The most common activity in research is the comparison of groups. Do patients receiving a new drug show greater improvement than patients using an existing medication? Does one manufacturing process produce fewer defects than another? Which of three teaching methods is most cost effective? If our outcome variable is categorical, we can use the methods described in section 5.3 or chapter 14. If the outcome variable is a survival time, we would use the methods described in section 12.4. Here, we will focus on group comparisons, where the outcome variable is continuous or ordinal. First we will look at the two group case. Then we will consider designs involving more than two groups. Finally, we will look at ways of visualizing our results.

5.5.1 Two Groups

To illustrate the two group case, we will use the `UScrime` dataset distributed with the MASS package. It contains information on the effect of punishment regimes on crime rates in 47 US states in 1960. The variables of interest will be `Prob` (the probability of imprisonment), `So` (an indicator variable for southern states), `U1` (the unemployment rate for urban males age 14-24) and U2 (the unemployment rate for urban males age 35-39). The data have been rescaled by the original authors. (Note: I considered naming this section "Crime and Punishment in the Old South", but cooler heads prevailed.)

T-TESTS

Are we more likely to be imprisoned if we commit a crime in the South? The comparison of interest is southern vs. non-southern states and the dependent variable is the probability of incarceration. A 2-group independent t-test can be used to test the hypothesis that the two population means are equal. Here, we assume that the two groups are independent and that the data are sampled from normal populations. The format is either

```
t.test(y~x)
```

where `y` is numeric and `x` is a dichotomous factor or

```
t.test(y1,y2)
```

where `y1` and `y2` are numeric vectors (the outcome variable for each group). In contrast to most statistical packages, the default test assumes unequal variance and applies the Welsh degrees of freedom modification. We can add a `var.equal=TRUE` option to specify equal variances and a pooled variance estimate. By default, a two-tailed alternative is assumed (i.e., the means differ but the direction is not specified). We can add the option `alternative="less"` or `alternative="greater"` to specify a directional test.

In listing 5.22, we compare southern (group 1) and non-southern (group 0) states on the probability of imprisonment using a two-tailed test without the assumption of equal variances.

Listing 5.22 Independent groups t-test

```
> t.test(Prob~So)

        Welch Two Sample t-test

data:  Prob by So
t = -3.8954, df = 24.925, p-value = 0.0006506                          1
alternative hypothesis: true difference in means is not equal to 0
```

```
95 percent confidence interval:
 -0.03852569 -0.01187439
sample estimates:
mean in group 0 mean in group 1
     0.03851265      0.06371269
```

#1 We reject the hypothesis that southern states and non-southern states have equal probabilities of imprisonment (p < .001).

Note: Since the outcome variable is a proportion, we might seek to transform it towards normality before carrying out the t-test. In the current case, all reasonable transformations of the outcome variable (Y/1-Y, log(Y/1-Y), arcsin(Y), arcsin(sqrt(Y)) would have led to the same conclusions.

As a second example, we might ask if unemployment rate for younger males (14-24) is greater than for older males (35-39). In this case, the two groups are not independent. We would not expect the unemployment rate for younger and older males in Alabama to be unrelated. When observations in the two groups are related, we have a dependent groups design. Pre-post or repeated measures designs also produce dependent groups.

A dependent t-test assumes that the difference between groups is normally distributed. In this case, the format is

```
 t.test(y1, y2, paired=TRUE)
```

where y1 and y2 are the numeric vectors for the two dependent groups. The results are provided in listing 5.23.

Listing 5.23 Dependent groups t-test

```
> sapply(UScrime[c("U1","U2")],function(x)(c(mean=mean(x),sd=sd(x))))

       U1    U2
mean 95.5 33.98
sd   18.0  8.45

> t.test(U1,U2,paired=TRUE)

        Paired t-test

data:  U1 and U2
t = 32.4066, df = 46, p-value < 2.2e-16                             1
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 57.67003 65.30870
sample estimates:
mean of the differences
               61.48936                                             1
```

#1 The mean difference (61.5) is large enough to warrant rejection of the hypothesis that the mean unemployment rate for older and younger males is the same. Younger males have a higher rate.

NONPARAMETRIC TESTS

If we are unable to meet the parametric assumptions of a t-test, we can turn to nonparametric approaches. For example, if the outcome variables are severely skewed or ordinal in nature, we may wish to use the techniques in this section.

If the two groups are independent, we can use the Wilcoxon rank sum test (more popularly known as the Mann-Whitney U test) to assess whether the observations are sampled from the same probability distribution (i.e., whether the probability of obtaining higher scores is greater in one population than the other). The format is either

```
wilcox.test(y~x)
```

where y is numeric and x is a dichotomous factor or

```
wilcox.test(y1,y2)
```

where y1 and y2 are the outcome variables for each group. The default is a two-tailed test. We can add the option `exact` to produce an exact test, and `alternative="less"` or `alternative="greater"` to specify a directional test.

If we apply the Mann-Whitney U test to the question of incarceration rates from the previous section, we would get the results in section 5.24.

Listing 5.24 Mann-Whitney U Test

```
> by(Prob,So,median)

So: 0
[1] 0.0382
--------------------
So: 1
[1] 0.0556

> wilcox.test(Prob~So)

        Wilcoxon rank sum test

data:  Prob by So
W = 81, p-value = 8.488e-05                                   1
alternative hypothesis: true location shift is not equal to 0
```

#1 Again, we can reject the hypothesis that incarceration rates are the same in southern and non-southern states (p<.001).

The Wilcoxon Signed Rank Test provides a nonparametric alternative to the dependent sample t-test. It is appropriate in situations where the groups are paired and the assumption of normality is unwarranted. The format is identical to the Mann-Whitney U test, but we add the `paired=TRUE` option. Let's apply it to the unemployment question from the previous section (listing 5.25).

Listing 5.25 Wilcoxon Signed Rank Test

```
> sapply(UScrime[c("U1","U2")],median)

U1 U2
92 34

> wilcox.test(U1,U2,paired=TRUE)

        Wilcoxon signed rank test with continuity correction

data:  U1 and U2
V = 1128, p-value = 2.464e-09                                      1
alternative hypothesis: true location shift is not equal to 0
```

#1 Again, we would reach the same conclusion reached with the paired t-test.

In this case, the parametric t-tests and their nonparametric equivalents reach the same conclusions. When the assumptions for the t-tests are reasonable, the parametric tests will be more powerful (more likely to find a difference if it exists). The non-parametric tests are more appropriate when the assumptions are grossly unreasonable (e.g., rank ordered data).

5.5.2 More than two groups

When there are more than two groups to be compared, we must turn to other methods. Consider the `state.x77` dataset from section 5.4. It contains population, income, illiteracy rate, life expectancy, murder rate, and high school graduation rate data for US states. What if want to compare the illiteracy rates in four regions of the country (Northeast, South, North Central, and West)? This is called a one-way design, and there are both parametric and nonparametric approaches available to address the question.

ANALYSIS OF VARIANCE

If we can assume that the data are independently sampled from normal populations, we can use analysis of variance (ANOVA) to compare groups. ANOVA is a comprehensive methodology that covers many experimental and quasi-experimental designs. As such, it has earned its own chapter. Feel free to abandon us and jump to chapter 8 at any time.

NONPARAMETRIC TESTS

If we can't meet the assumptions of ANOVA designs, we can use nonparametric methods to evaluate group differences. If the groups are independent, a Kruskal-Wallis test will provide us with a useful approach. If the groups are dependent (e.g., repeated measures or randomized block design), the Friedman test is more appropriate.

The format for the Kruskal Wallis test is

```
kruskal.test(y~A)
```

where y is a numeric outcome variable and A is a group factor with 2 or more levels (if there are two levels, it is equivalent to the Mann-Whitney U test). For the Friedman test, the format is

```
friedman.test(y~A|B)
```

where y is the numeric outcome variable, A is a group factor, and B is a blocking factor that identifies matched observations.

Let's apply the Kruskal Wallis test to the illiteracy question above. First, we will have to add the region designations to the dataset. These are contained in the dataset `state.region` distributed with the base installation of R.

```
states <- as.data.frame(cbind(state.region, state.x77))
```

Now we can apply our test (see listing 5.26).

Listing 5.26 Kruskal Wallis test - One Way Anova by Ranks

```
> attach(states)
> kruskal.test(Illiteracy~state.region)

        Kruskal-Wallis rank sum test

data:  states$Illiteracy by states$state.region
Kruskal-Wallis chi-squared = 22.7, df = 3, p-value = 4.726e-05    1
```

#1 The significance test suggests that the illiteracy rate is not the same in group of the four regions of the country (p <.001).

Although we can reject the null hypothesis of no difference, the test does not tell us which regions differ significantly from which others. To answer this question, we could compare groups two at a time using the Mann-Whitney U test. A more elegant approach is to apply a simultaneous multiple comparisons procedure that makes all pairwise comparisons, while controlling the type I error rate (the probability of finding a difference that isn't there). The `npmc` package provides the nonparametric multiple comparisons we need.

To be honest, we are stretching our definition of "Basic" in the chapter title quite a bit, but since it really fits well here, I hope you will bear with me. First, be sure to install the `npmc` package. The `npmc` function in this package expects input to be a two column dataframe with a column named `var` (the dependent variable) and `class` (the grouping variable). We can accomplish this with the code in listing 5.27.

Listing 5.27 Nonparametric multiple comparisons

```
> class <- state.region
> var <- state.x77[,c("Illiteracy")]
> mydata <- as.data.frame(cbind(class, var))
> summary(npmc(mydata, type="BF"))

$`Data-structure`
              group.index   class.level nobs
Northeast               1     Northeast    9
South                   2         South   16
North Central           3 North Central   12
West                    4          West   13

$`Results of the multiple Behrens-Fisher-Test`                       1
  cmp effect lower.cl upper.cl p.value.1s p.value.2s
1 1-2 0.8750  0.66149   1.0885   0.000665    0.00135
2 1-3 0.1898 -0.13797   0.5176   0.999999    0.06547
3 1-4 0.3974 -0.00554   0.8004   0.998030    0.92004
4 2-3 0.0104 -0.02060   0.0414   1.000000    0.00000
5 2-4 0.1875 -0.07923   0.4542   1.000000    0.02113
6 3-4 0.5641  0.18740   0.9408   0.797198    0.98430

> aggregate(mydata, by=list(mydata$class),median)            2

  Group.1 class  var
1       1     1 1.10
2       2     2 1.75
3       3     3 0.70
4       4     4 0.60
```

#1 The `npmc` call generates six statistical comparisons (Northeast vs. South, Northeast vs. North Central, northeast vs. West, South vs. North Central, South vs. West, and North Central vs. West). We can see from the two-sided p-values (p.value.2s) that the South differs significantly from the other three regions, and that the other three regions do not differ from each other. From #2 we see that the South has a higher median illiteracy rate.

5.5.3 Visualizing group differences

Examining group differences visually is a crucial part of a comprehensive data analysis strategy. It allows us to assess the magnitude of the differences, identify any distributional characteristics that influence the results (e.g., skew, bimodality, outliers), and evaluate the appropriateness of our test assumptions. R provides a wide range of graphical methods for comparing groups including box plots (simple, notched, violin, and bagplots) covered in section 11.2, overlapping kernel density plots, covered in section 11.1, and graphical methods of assessing test assumptions, discussed in section 8.6.

5.6 Summary

In this chapter, we have reviewed the functions in R that provide basic statistical summaries and tests. It has included sample statistics and frequency tables, tests of independence and measures of association for categorical variables, correlations between quantitative variables (and their associated significance tests), and comparisons of two or more groups on a quantitative outcome variable.

In the next chapter we take up the topic of basic graphs. They form a natural partnership with the topics we have just covered. As we will see throughout this book, there is a yin and a yang between numerical summaries and statistical tests, and visual depictions of relationships and differences. However, the graphical chapters are more fun to look at.

References

Koch, G & S. Edwards, S. (1988), Clinical efficiency trials with categorical data. In K. E. Peace (ed.), *Biopharmaceutical Statistics for Drug Development*, 403–451. Marcel Dekker, New York.

McCall, R. B. (2000). Fundamental statistics for the behavioral sciences (8th ed.). Wadsworth Publishing, New York.

Snedecor, G. W., & Cochran, W.G. (1988). Statistical methods (8th ed.). Iowa State University Press, Iowa.