

# INSTANT R

An Introduction to R for Statistical Analysis

SARAH STOWELL

**Instant R: An Introduction to R for Statistical Analysis**

by Sarah Stowell

Copyright ©2012 Sarah Stowell

Published by Jotunheim Publishing

*[www.JotunheimPublishing.com](http://www.JotunheimPublishing.com)*

No part of this book may not be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, microfilming and recording, or by any information storage or retrieval system, without prior permission in written form from the publisher.

Whilst effort has been made to make this book as complete and accurate as possible, the author and publisher cannot assume responsibility for the validity of the materials or for any loss or damages arising from the use of the information contained herein.

Cover design by Eddie Chung

*[www.eddiechungdesign.com](http://www.eddiechungdesign.com)*

ISBN 978-0-9574649-0-2

# Contents

---

<b>Preface</b>	<b>vii</b>
<b>1 R fundamentals</b>	<b>1</b>
1.1 Downloading and installing R . . . . .	1
1.2 Getting orientated . . . . .	2
1.3 The R console and command prompt . . . . .	3
1.4 Functions . . . . .	5
1.5 Objects . . . . .	6
1.6 Vectors . . . . .	8
1.7 Data frames . . . . .	9
1.8 The data editor . . . . .	13
1.9 Workspaces . . . . .	14
1.10 Error messages . . . . .	15
1.11 Script files . . . . .	15
1.12 Chapter summary: R Fundamentals . . . . .	18
<b>2 Working with data files</b>	<b>19</b>
2.1 Entering data directly . . . . .	19
2.2 Importing plain text files . . . . .	20
2.2.1 CSV and tab-delimited files . . . . .	21
2.2.2 DIF files . . . . .	23
2.2.3 Other plain text files . . . . .	24
2.3 Importing Excel files . . . . .	24
2.4 Importing files from other software . . . . .	26
2.5 Using relative file paths . . . . .	28
2.6 Exporting datasets . . . . .	28
2.7 Chapter summary: Working with data files . . . . .	30
<b>3 Preparing and manipulating your data</b>	<b>31</b>
3.1 Rearranging and removing variables . . . . .	32
3.2 Renaming variables . . . . .	33
3.3 Variable classes . . . . .	33
3.4 Calculating new numeric variables . . . . .	35
3.5 Dividing a continuous variable into categories . . . . .	36
3.6 Working with factor variables . . . . .	37
3.7 Manipulating character variables . . . . .	38
3.7.1 Concatenating character strings . . . . .	38
3.7.2 Extracting a substring . . . . .	39
3.7.3 Searching a character variable . . . . .	40
3.8 Working with dates and times . . . . .	41

3.9	Adding and removing observations . . . . .	43
3.10	Removing duplicate observations . . . . .	44
3.11	Selecting a subset of the data . . . . .	44
3.12	Selecting a random sample from a dataset . . . . .	47
3.13	Sorting a dataset . . . . .	47
3.14	Chapter summary: Preparing and manipulating your data . . . . .	48
<b>4</b>	<b>Combining and restructuring datasets</b>	<b>49</b>
4.1	Appending rows . . . . .	49
4.2	Appending columns . . . . .	50
4.3	Merging datasets by common variables . . . . .	51
4.4	Stacking data . . . . .	54
4.5	Unstacking data . . . . .	55
4.6	Reshaping a dataset . . . . .	56
4.7	Chapter summary: Combining and restructuring datasets . . . . .	58
<b>5</b>	<b>Summary statistics for continuous variables</b>	<b>59</b>
5.1	Univariate statistics . . . . .	59
5.2	Statistics by group . . . . .	61
5.3	Measures of association . . . . .	63
5.3.1	Covariance . . . . .	64
5.3.2	Pearson's correlation coefficient . . . . .	64
5.3.3	Spearman's rank correlation coefficient . . . . .	65
5.3.4	Hypothesis test of correlation . . . . .	66
5.4	Shapiro-Wilk test . . . . .	67
5.5	Kolmogorov-Smirnov test . . . . .	68
5.6	Confidence intervals and prediction intervals . . . . .	70
5.7	Chapter summary: Summary statistics for continuous variables . . . . .	72
<b>6</b>	<b>Tabular data</b>	<b>73</b>
6.1	Frequency tables . . . . .	73
6.2	Chi-square goodness-of-fit test . . . . .	77
6.3	Chi-square test of association . . . . .	79
6.4	Fisher's exact test . . . . .	81
6.5	Chapter summary: Tabular data . . . . .	82
<b>7</b>	<b>Probability distributions</b>	<b>83</b>
7.1	Probability density functions and probability mass functions . . . . .	85
7.2	Finding probabilities . . . . .	86
7.3	Finding quantiles . . . . .	88
7.4	Generating random numbers . . . . .	89
7.5	Chapter summary: Probability distributions . . . . .	91

<b>8</b>	<b>Creating plots</b>	<b>93</b>
8.1	Simple plots . . . . .	93
8.2	Histograms . . . . .	95
8.3	Normal probability plots . . . . .	96
8.4	Stem-and-leaf plots . . . . .	97
8.5	Bar charts . . . . .	98
8.6	Pie charts . . . . .	100
8.7	Scatter plots . . . . .	101
8.8	Scatterplot matrices . . . . .	102
8.9	Box plots . . . . .	104
8.10	Plotting a function . . . . .	105
8.11	Exporting and saving plots . . . . .	106
8.12	Chapter summary: Creating plots . . . . .	107
<b>9</b>	<b>Customising your plots</b>	<b>109</b>
9.1	Titles and labels . . . . .	109
9.2	Axes . . . . .	111
9.3	Colours . . . . .	112
9.4	Plotting symbols . . . . .	113
9.5	Plotting lines . . . . .	114
9.6	Shaded areas . . . . .	115
9.7	Adding additional items to plots . . . . .	116
9.7.1	Adding additional straight lines . . . . .	116
9.7.2	Adding a mathematical function curve . . . . .	117
9.7.3	Adding labels and text . . . . .	117
9.7.4	Adding a grid . . . . .	118
9.7.5	Adding arrows . . . . .	119
9.8	Overlaying plots . . . . .	119
9.9	Adding a legend . . . . .	122
9.10	Multiple plots in the plotting area . . . . .	124
9.11	Changing the default plot settings . . . . .	124
9.12	Chapter summary: Customising your plots . . . . .	126
<b>10</b>	<b>Hypothesis tests</b>	<b>127</b>
10.1	Student's t-tests . . . . .	128
10.1.1	One-sample t-test . . . . .	129
10.1.2	Two-sample t-test . . . . .	130
10.1.3	Paired t-test . . . . .	132
10.2	Wilcoxon rank-sum test . . . . .	134
10.3	Analysis of variance . . . . .	136
10.4	Kruskal-Wallis test . . . . .	138
10.5	Multiple comparison methods . . . . .	139
10.5.1	Tukey's HSD test . . . . .	139
10.5.2	Other pairwise t-tests . . . . .	140
10.5.3	Pairwise Wilcoxon rank-sum tests . . . . .	141

10.6 Hypothesis tests for variance . . . . .	142
10.6.1 F-test . . . . .	142
10.6.2 Bartlett's test . . . . .	143
10.8 Chapter summary: Hypothesis tests . . . . .	145
<b>11 Regression and general linear models</b>	<b>147</b>
11.1 Building the model . . . . .	147
11.1.1 Simple linear regression . . . . .	148
11.1.2 Multiple linear regression . . . . .	149
11.1.3 Interaction terms . . . . .	149
11.1.4 Polynomial terms . . . . .	151
11.1.5 Transformations . . . . .	151
11.1.6 The intercept term . . . . .	152
11.1.7 Including factor variables . . . . .	152
11.1.8 Updating a model . . . . .	153
11.1.9 Stepwise model selection procedures . . . . .	154
11.2 Summarising the model . . . . .	156
11.3 Coefficient estimates . . . . .	158
11.4 Plotting the line of best fit . . . . .	158
11.5 Model diagnostics . . . . .	160
11.5.1 Residual analysis . . . . .	160
11.5.2 Leverage . . . . .	163
11.5.3 Cook's distances . . . . .	163
11.6 Making predictions . . . . .	164
11.7 Chapter summary: General linear models . . . . .	167
<b>12 Basic programming</b>	<b>169</b>
12.1 Creating new functions . . . . .	169
12.2 Conditional statements . . . . .	171
12.2.1 Conditions . . . . .	172
12.2.2 If statement . . . . .	173
12.2.3 If/else statement . . . . .	175
12.2.4 The switch function . . . . .	176
12.3 Loops . . . . .	178
12.3.1 For loop . . . . .	178
12.3.2 While loop . . . . .	179
12.4 Chapter summary: Basic programming . . . . .	181
<b>A Add-on packages</b>	<b>183</b>
<b>B Datasets</b>	<b>187</b>
<b>Bibliography</b>	<b>197</b>

# Preface

---

## About R

R is a statistical analysis and graphics environment and also a programming language. It is command driven and very similar to the commercially-produced S-Plus<sup>®</sup> software. R is known for its professional looking graphics, which allow complete customisation.

R is open source software and free to install under the GNU general public license. It is written and maintained by a group of volunteers known as the R core team.

The base software is supplemented by over three thousand add-on packages developed by R users all over the world, many of whom belong to the academic community. These packages cover a broad range of statistical techniques including some of the most recently developed and niche purpose. Anyone can contribute add-on packages, which are checked for quality before they are added to the collection.

At the time of writing, the current version of R is 2.15.

## The purpose of this book

This book is designed to give straight forward, practical guidance for performing popular statistical methods in R. The programming aspect of R is explored only briefly.

After reading this book you will be able to:

- navigate the R system
- enter and import data
- manipulate datasets
- calculate summary statistics
- create statistical plots and customise their appearance
- perform hypothesis tests such as the t-test and analysis of variance
- build regression models

- create your own functions
- access additional functionality with the use of add-on packages.

## Knowledge assumed

Whilst the book does include some reminders about statistics methods and examples demonstrating their use, it is not intended to teach statistics. Therefore you will require some previous knowledge such that you are able to select the most appropriate statistical method for your purpose and interpret the results. You should also be familiar with common statistical terms and concepts. If you are unsure about any of the methods that you are using, I recommend that you use this book in conjunction with a more detailed book on statistics.

No prior knowledge of R or of programming is assumed, making this book ideal if you are more accustomed to working with point-and-click style packages. Only general computer skills and a familiarity with your operating system are required.

## Conventions used in this book

This book uses the following typographical conventions:

`Fixed width font` is used to distinguish all R commands and output from the main text.

`Normal fixed width font` is used for built-in R function names, argument names, syntax, specific dataset and variable names and any other parts of the commands that can be copied verbatim.

*Slanted fixed width font* is used for generic dataset and variable names and any other parts of the commands that should be replaced with the user's own values.

Often it has not been possible to fit a whole command into the width of the page. In these cases, the command is continued on the following line and indented. Where you see this, the command should still be entered into the console on a single line.

Grey text boxes contain reminders of statistical theory or methods which are separate from the main text.



## Datasets used in this book

A large number of example datasets are included with R, and these are available to use as soon as you open the software. This book makes use of several of these datasets for demonstration purposes.

There are also a number of additional datasets used throughout the book, details of which are given in the Appendix B. They are available from the website:

*[www.InstantR.com](http://www.InstantR.com)*

## Contact the author

Supplementary materials, updates and datasets are available from the website:

*[www.InstantR.com](http://www.InstantR.com)*

To give feedback you can email me at:

*[S.Stowell@InstantR.com](mailto:S.Stowell@InstantR.com)*.

## Acknowledgements

Many thanks to Jemma-Kay Johnstone, Christopher Gilmour, Nina Farrell, Chris Brown and Artur Kyräl, who generously gave their time reviewing the first draft and providing valuable feedback.

I would also like to thank to Andrés Barnett, James Sedgwick and Therese Stukel for providing data for the examples, and to James Sedgwick and Eddie Chung for help with other practical issues.

Thanks to Eddie Chung at *[www.eddiechungdesign.com](http://www.eddiechungdesign.com)* for cover design.



# Chapter 1

## R fundamentals

---

This chapter introduces the R system, beginning with how to download and install R, familiarise yourself with the interface and start giving commands. You will learn about the different types of R files. It also explains all of the important technical terms that will be used throughout the book. If you are new to R I recommend that you read the entire chapter, as it will give you a solid foundation on which to build.

### 1.1 Downloading and installing R

The R software is freely available from the R website. Windows<sup>®</sup> and Mac<sup>®</sup> users should follow the instructions below to download the installation file:

1. Go to the R project website at [www.r-project.org](http://www.r-project.org).
2. Follow the link to 'CRAN' (on the left-hand side).
3. You will be taken to a list of sites that host the R installation files (mirror sites). Select a site close to your location.
4. Select your operating system. There are installation files available for the Windows, Mac and Linux<sup>®</sup> operating systems.
5. If downloading R for Windows, you will be asked to select from the 'base' or 'contrib' distributions. Select the 'base' distribution.
6. Follow the link to download the R installation file and save the file to a suitable location on your machine.

To install R for the Windows and Mac OS environments, open the installation file and follow the instructions given by the set-up wizard as you would for any other piece of Windows-based software. You will be given the option of customising the installation, but if you are new to R I recommend that you use the standard installation settings. If you are installing R on a networked computer, you may need to contact your system administrator to obtain permission before performing the installation.

For Linux users, the simplest way to install R is via the package manager. You can find R by searching for *r-base-core*. Detailed installation instructions are available in the same location as the installation files.

If you have the required technical knowledge then you can also compile the software from the source code. An in-depth guide can be found at:

[www.stats.bris.ac.uk/R/doc/manuals/R-admin.pdf](http://www.stats.bris.ac.uk/R/doc/manuals/R-admin.pdf)

## 1.2 Getting orientated

Once you have installed the software and opened it for the first time, you will see the R interface as shown in Figure 1.1.

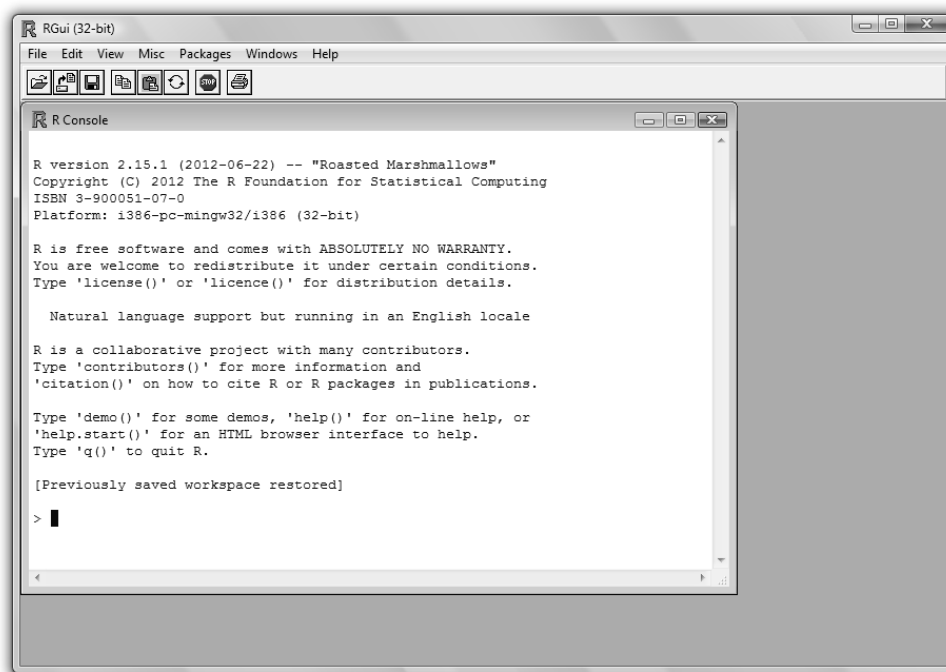


Figure 1.1: The R interface

There are several drop down menus and buttons, but unlike in point-and-click style statistical packages, you will only use these for supporting activities such as opening and saving R files, setting preferences and loading add-on packages. You will perform all of the main tasks (such as importing data, performing statistical analysis and creating graphs) by giving R typed *commands*.

The window named 'R Console' is where you will type your commands. It is also where the output and any error messages are displayed. Later you will use other windows such as the data editor, script editor and graphics device.

## 1.3 The R console and command prompt

Now turn your attention to the R console window. Every time you start R, some blue text relating to copyright and other issues appears in the console window, as shown in Figure 1.1. If you find the text in the console difficult to read, you can adjust it by selecting 'GUI preferences' from the 'Edit' menu. This opens a dialogue window which allows you to change the size and font of the console text, as well as other options.

Below all of the text that appears in the console at startup you will see the *command prompt*, which is coloured red and looks like this:

```
>
```

The command prompt tells you that R is ready to receive your command.

Try typing the following command at the prompt and pressing enter:

```
> 8-2
```

R responds by giving the following output in the next line of the console:

```
[1] 6
>
```

The [1] tells you which component of the output you are looking at, which is not of much interest at this stage as the output has only one component. This is followed by the result of the calculation, which is 6. Notice that all output is shown in blue, to distinguish it from your commands.

The output is followed by another prompt (>) to tell you that it has finished processing your command and is ready for the next one. If you don't see a command prompt after entering a command, it may be because the command you have given is not complete. Try entering the following incomplete command at the command prompt:

```
> 8-
```

R responds with a plus sign:

```
+
```

If you see the plus sign it means you need to type the remainder of the command and press enter. Alternatively you can press the 'Esc' key to cancel the command and return to the command prompt.

Another time that you would not see the command prompt is when R is still working on the task. Usually this time is negligible, but there may be some waiting time for more complex tasks or those involving large datasets. If a command takes much longer than expected to complete, you can cancel it with the 'Esc' key.

From here onwards the command prompt will be omitted when showing output.

Table 1.1 shows the symbols used to represent the basic arithmetic operations.

Operation	Symbol
Addition	+
Subtraction	−
Multiplication	*
Division	/
Exponentiation	^

**Table 1.1:** Arithmetic Operators

If a command is composed of several arithmetic operators, they are evaluated in the usual order of precedence i.e. first the exponentiation (power) symbol, followed by division, then multiplication, and finally addition and subtraction. You can also add parenthesis to control precedence if required. For example the command:

```
> 3^2+6/3+2
```

gives the result:

```
[1] 13
```

while the command:

```
> (3^2+6) / (3+2)
```

gives the result:

```
[1] 3
```

If you want to repeat a command, you can use the up and down arrow keys on your keyboard to scroll through previously entered commands. You will be able to edit the command before pressing enter. This means that you don't have to retype a whole command just to correct a minor mistake, which you will find useful as you begin to use longer and more complex commands.

## 1.4 Functions

In order to do anything more than basic arithmetic calculations, you will need to use *functions*. A function is a set of commands which have been given a name and together perform a specific task producing some kind of output. Usually a function also requires some kind of data as input.

R has many built-in functions for performing a variety of tasks from simple things like rounding numbers, to importing files and performing complex statistical analysis. You will make use of these throughout this book. You can also create your own functions, which is covered briefly in Chapter 12.

Whenever you use a function, you will type the function name followed by round brackets. Any input required by the function is placed between the brackets.

An example of a function that does not require any input is the `date` function, which gives the current date and time from your computer's clock.

```
> date()  
[1] "Sun Jun 10 15:19:27 2012"
```

An example of a simple function that requires input is the `round` function, which rounds numbers. The input required is the number you want to round. A single piece of input is known as an *argument*.

```
> round(3.141593)  
[1] 3
```

As you can see, the `round` function rounds a given number to the nearest whole number, but you can also use it to round a number to a different level of accuracy. The command below rounds the same number to two decimal places.

```
> round(3.141593, digits=2)  
[1] 3.14
```

We were able to change the behaviour of the `round` function by adding an additional argument giving the number of decimal places required. When you provide more than one argument to a function, they must be separated with commas. Each argument has a name. In this case, the argument giving the number of decimal places is called `digits`. Often you don't need to give the names of the arguments, because R is able to identify them by their values and the order in which they are arranged. So for the `round` function, the following command is also acceptable.

```
> round(3.141593, 2)
```

Some arguments are optional and some must be provided for the function to work. For the `round` function, the number to be rounded (in this example 3.141593) is a required argument and the function won't work without it. The `digits` argument is optional. If you don't supply it, R assumes a default value of zero.

For every function included with R, there is a help file which you can view by entering the command:

```
> help(functionname)
```

The help file gives details of all of the arguments for the function, whether they are required or optional and what their default values are.

Table 1.2 shows some useful mathematical functions.

Purpose	Function
Exponential	exp
Natural logarithm	log
Log base 10	log10
Square root	sqrt
Cosine	cos
Sine	sin
Tangent	tan
Arc cosine	acos
Arc sine	asin
Arc tangent	atan
Round	round
Absolute value	abs
Factorial	factorial

**Table 1.2:** Useful mathematical functions

## 1.5 Objects

In R, an *object* is some data which has been given a name and stored in the memory. The data could be anything from a single number to a whole table of data of mixed types.

You can create objects with the *assignment operator*, which looks like this:

```
<-
```

For example, to create an object named `object1` that holds the value 5.234, use the command;

```
> object1<-5.234
```

When creating new objects, you must choose an object name that:

1. consists only of upper and lower case letters, numbers, underscores (`_`) and dots (`.`)



2. begins with an upper or lower case letter or a dot (.)
3. is not one of R's reserved words. Enter `help(reserved)` to see a list of these.

R is case sensitive, so `object1`, `OBJECT1` and `Object1` are all distinct object names.

If you choose an object name which is already in use, you will overwrite the old object with the new one. R does not give any warning if you do this.

To view the contents of an object you have already created, enter the object name.

```
> object1  
[1] 5.234
```

Once you have created an object, you can use it in place of the information it contains. For example as input to a function:

```
> log(object1)  
[1] 1.655176
```

As well as creating objects with specific values, you can save the output of a function or calculation directly to a new object:

```
> object2<-log(object1)+0.6
```

Notice that when you assign the output from a function or calculation to an object, R does not display the output. To see it, you must view the contents of the object by entering the object name.

To change the contents of an object, simply overwrite it with a new value.

```
> object1<-6.214
```

Objects like those created above are called *numeric* objects because they contains numbers. You can also create other types of objects such as *character* objects, which contain a combination of any keyboard characters known as a *character string*. When creating a character object, enclose the character string in quotation marks as shown below.

```
> object3<-"Hello!"
```

You can use either double or single quotation marks to enclose a character string, as long they are both of the same type. To include quotation marks within a character string, place a backslash before the quotes (known as an *escape sequence*), as shown.

```
> object4<-"I said \"Hello!\""
```

So far we have only discussed simple objects which contain a single data value, but you can also create more complex types of objects. Two important types are *vectors* and *data frames*. A vector is an object that contains several data values of the same type. A data frame is an object that holds an entire dataset. Vectors and data frames are discussed in more detail in the following sections.

## 1.6 Vectors

A vector is an object that holds several data values of the same type arranged in a particular order. You can create vectors with a special function which is named `c`, as shown below.

```
> vector1<-c(3, 3.76, -0.35, 1.2, -5)
```

You can view the contents of a vector by entering its name, as you would for any other object.

```
> vector1
[1] 3.00 3.76 -0.35 1.20 -5.00
```

The number of values a vector holds is called its *length*. You can check the length of a vector with the `length` function.

```
> length(vector1)
[1] 5
```

Each data value in the vector has a position within the vector, which you can refer to using square brackets. This is known as bracket notation. For example, you can view the third member of `vector1` with the command:

```
> vector1[3]
[1] -0.35
```

If you have a large vector (such that when displayed, the values of the vector fill several lines of the console window), the indices at the side tell you which member of the vector each line begins with. For example, the vector below contains twenty-seven values. The indices at the side show that the second line begins with the eleventh member and the third line begins with the twenty-first member. This helps you to determine the position of each value within the vector.

---

```
[1] 0.077 0.489 1.603 2.110 2.625 1.019 1.100 1.729 2.469 -0.125
[11] 1.931 0.155 0.572 1.160 -1.405 2.868 0.632 -1.714 2.615 0.714
[21] 0.979 1.768 1.429 -0.119 0.459 1.083 -0.270
```

---

If you give a vector as input to a function intended for use with a single number (such as the `exp` function), R applies the function to each member of the vector individually and gives another vector as output.

```
> exp(vector1)
[1] 20.085536923 42.948425979 0.704688090 3.320116923 0.006737947
```

Some functions are designed specifically for use with vectors and use all members of the vector together to create a single value as output. An example is the `mean` function, which calculates the mean of all the values in the vector.

```
> mean(vector1)
[1] 0.522
```

The `mean` function and other statistical summary functions are discussed in more detail in Section 5.1.

Like basic objects, vectors can hold different types of data values such as numbers or character strings. However all members of the vector must be of the same type. If you attempt to create a vector containing both numbers and characters, R will convert any numeric values into *numerals*. Numerals are character representation of numbers which look like numbers but are treated as text and cannot be used in calculations.

## 1.7 Data frames

A data frame is a type of object which is suitable for holding a dataset. A data frame is composed of several vectors of the same length, displayed vertically and arranged side by side. This forms a rectangular grid in which each column has a name and contains one vector. Whilst all the values in one column of a data frame must be of the same type, different columns can hold different types of data (such as numbers or character strings). This makes them ideal for storing datasets, with each column holding a variable and each row an observation.

In Chapter 2 you will learn how to create new data frames to hold your own datasets. For now, there are some datasets included with R that you can experiment with. One of these is called `Puromycin`, which we will use here to demonstrate the idea of a data frame. You can view the contents of the `Puromycin` dataset in the same way as for any other object, by entering its name at the command prompt.

```
> Puromycin
```

R outputs the contents of the data frame:

---

	conc	rate	state
1	0.02	76	treated
2	0.02	47	treated
3	0.06	97	treated
4	0.06	107	treated
5	0.11	123	treated
6	0.11	139	treated
7	0.22	159	treated
8	0.22	152	treated
9	0.56	191	treated
10	0.56	201	treated
11	1.10	207	treated
12	1.10	200	treated
13	0.02	67	untreated
14	0.02	51	untreated
15	0.06	84	untreated

```

16 0.06 86 untreated
17 0.11 98 untreated
18 0.11 115 untreated
19 0.22 131 untreated
20 0.22 124 untreated
21 0.56 144 untreated
22 0.56 158 untreated
23 1.10 160 untreated

```

---

The dataset has three variables named `conc`, `rate` and `state`, and it has 23 observations.

It is important to know how to refer to the different components of a data frame. To refer to a particular variable within a dataset by name, use the dollar symbol (\$) as shown below.

```
> Puromycin$rate
```

```

[1] 76 47 97 107 123 139 159 152 191 201 207 200 67 51 84 86 98 115 131
[20] 124 144 158 160

```

---

This is useful because it allows you to apply functions to the variable, e.g. to calculate the mean value of the `rate` variable.

```
> mean(Puromycin$rate)
[1] 126.8261
```

As well as selecting variables by name with the dollar symbol, you can refer to sections of the data frame using *bracket notation*. Bracket notation can be thought of as a coordinate system for the data frame. You provide the row number and column number between square brackets.

```
> dataset[r,c]
```

For example, to select the value in the sixth row of the second column of the `Puromycin` dataset using bracket notation, use the command:

```
> Puromycin[6,2]
[1] 139
```

You can select a whole row by leaving the column number blank. For example to select the sixth row of the `Puromycin` dataset:

```
> Puromycin[6,]
```

```

  conc rate  state
6 0.11 139 treated

```

---

Similarly to select a whole column, leave the row number blank. For example to select the second column of the `Puromycin` dataset:

```
> Puromycin[,2]
```

---

```
[1] 76 47 97 107 123 139 159 152 191 201 207 200 67 51 84 86 98 115 131
[20] 124 144 158 160
```

---

When selecting whole columns, you can also leave out the comma entirely and just give the column number.

```
> Puromycin[2]
```

You can use the minus sign to exclude a part of the data frame instead of selecting it. For example to exclude the first column:

```
> Puromycin[-1]
```

---

```
      rate      state
1      76    treated
2      47    treated
3      97    treated
4     107    treated
5     123    treated
6     139    treated
7     159    treated
8     152    treated
9     191    treated
10    201    treated
11    207    treated
12    200    treated
13     67 untreated
14     51 untreated
15     84 untreated
16     86 untreated
17     98 untreated
18    115 untreated
19    131 untreated
20    124 untreated
21    144 untreated
22    158 untreated
23    160 untreated
```

---

You can use the colon (:) to select a range of rows or columns. For example to select row numbers six to ten:

```
> Puromycin[6:10,]
```

---

```
      conc rate      state
6 0.11 139 treated
7 0.22 159 treated
8 0.22 152 treated
9 0.56 191 treated
10 0.56 201 treated
```

---

To select non-consecutive rows or columns, use the `c` function inside the brackets. For example to select columns one and three:

```
> Puromycin[,c(1,3)]
```

---

```
      conc      state
1  0.02    treated
2  0.02    treated
3  0.06    treated
4  0.06    treated
5  0.11    treated
6  0.11    treated
7  0.22    treated
8  0.22    treated
9  0.56    treated
10 0.56    treated
11 1.10    treated
12 1.10    treated
13 0.02 untreated
14 0.02 untreated
15 0.06 untreated
16 0.06 untreated
17 0.11 untreated
18 0.11 untreated
19 0.22 untreated
20 0.22 untreated
21 0.56 untreated
22 0.56 untreated
23 1.10 untreated
```

---

You can also use object names in place of numbers:

```
> rownum<-c(6,8,14)
> colnum<-2
> Puromycin[rownum,colnum]
[1] 139 152  51
```

Or even functions:

```
> Puromycin[sqrt(25),]
```

---

```
      conc rate      state
5  0.11  123    treated
```

---

Finally, you can refer to specific entries using a combination of the variable name and bracket notation. For example to select the tenth observation for the `rate` variable:

```
> Puromycin$rate[10]

[1] 201
```

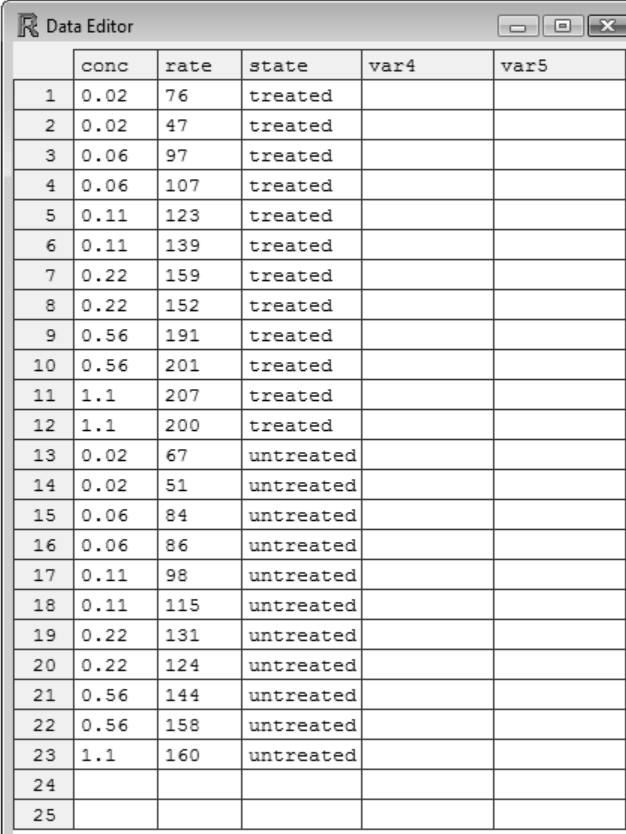
You can view more information about the `Puromycin` dataset (or any of the other dataset included with R) with the `help` function.

```
> help(Puromycin)
```

## 1.8 The data editor

As an alternative to viewing datasets in the command window, R has a spreadsheet style viewer called the *data editor* which allows you to view and edit data frames. To open the `Puromycin` dataset in the data editor window, use the command:

```
> fix(Puromycin)
```



	conc	rate	state	var4	var5
1	0.02	76	treated		
2	0.02	47	treated		
3	0.06	97	treated		
4	0.06	107	treated		
5	0.11	123	treated		
6	0.11	139	treated		
7	0.22	159	treated		
8	0.22	152	treated		
9	0.56	191	treated		
10	0.56	201	treated		
11	1.1	207	treated		
12	1.1	200	treated		
13	0.02	67	untreated		
14	0.02	51	untreated		
15	0.06	84	untreated		
16	0.06	86	untreated		
17	0.11	98	untreated		
18	0.11	115	untreated		
19	0.22	131	untreated		
20	0.22	124	untreated		
21	0.56	144	untreated		
22	0.56	158	untreated		
23	1.1	160	untreated		
24					
25					

Figure 1.2: The data editor window

Alternatively you can select 'Data editor' from the 'Edit' menu and enter the name of the dataset that you want to view when prompted. The dataset opens in the data editor window, as shown in Figure 1.2. Here you can make changes to the data. When you have finished, close the editor window to apply them.

Whilst the data editor can be useful for making minor changes, there are usually more efficient ways of manipulating a dataset. These are covered in Chapter 3.

## 1.9 Workspaces

The *workspace* is the virtual area containing all of the objects you have created in the session. To see a list of all the objects in the workspace, use the `objects` function:

```
> objects()
```

You can delete objects from the workspace with the `rm` function.

```
> rm(object1, object2, object3)
```

To delete all of the objects in the workspace, use the command:<sup>36</sup>

```
> rm(list=objects())
```

You can save the contents of the workspace to a file, which allows you to resume working with them at another time. To save the workspace, select 'File' then 'Save Workspace' from the drop-down menus, then name and save the file in the usual way. Ensure that the file name has the *.RData* file name extension, as it will not be added automatically.

R automatically loads the most recently-saved workspace at the beginning of each new session. You can also open a previously saved workspace by selecting 'File' then 'Open Workspace' from the drop-down menus and selecting the file in the usual way. Once you have opened a workspace, all of the objects within it are available for you to use.

Mac users can find options for saving and loading the workspace from the 'Workspace' menu.

Linux users can save the workspace by entering the command:

```
> save.image("/home/Username/folder/filename.RData")
```

The file path can be either absolute or relative to the home directory.

To load a workspace, use the command:

```
> load("/home/Username/folder/filename.RData")
```



## 1.10 Error messages

Sometimes R will encounter a problem while trying to complete one of your commands. When this happens, a message is displayed in the console window to inform you of the problem. These messages come in two varieties known as *error messages* and *warning messages*.

Error messages begin with the text 'Error:' and are displayed when R is not able to perform the command at all. One of most common causes of error messages is giving a command that is not a valid R command because it contains a symbol that R does not understand, or because a symbol is missing or in the wrong place. This is known as a *syntax error*. In the following example, the error is caused by an extra closing bracket at the end of the command.

```
> round(3.141592))  
Error: unexpected ')' in "round(3.141592))"
```

Another common cause of errors is mistyping an object name so that you are referring to an object that does not exist. Remember that object names are case sensitive.

```
> log(object5)  
Error: object 'object5' not found
```

The same applies to function names, which are also case sensitive.

```
> Log(3.141592)  
Error: could not find function "Log"
```

A third common cause of errors is giving the wrong type of input to a function, such as a data frame where a vector is expected, or a character string where a number is expected.

```
> log("Hello!")  
Error in log("Hello!") : Non-numeric argument to mathematical  
function
```

Warning messages begin with the text 'Warning:' and tell you about issues that have not prevented the command from being completed, but that you should be aware of.

```
> log(-2)  
[1] NaN  
Warning message:  
In log(-2) : NaNs produced
```

## 1.11 Script files

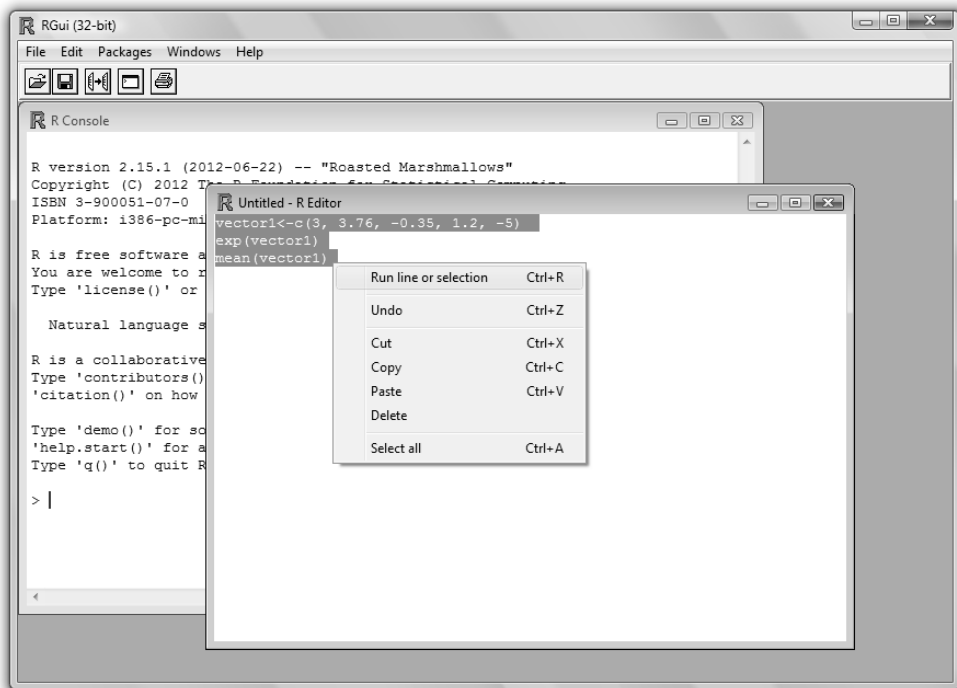
A script file is a type of text file that allows you to save your commands so that they can be easily reviewed, edited and repeated.

To create a new script file, select 'New script' from the 'File' menu. Mac users should select 'New Document' from the 'File' menu. This opens a new 'R Editor' window where you can type commands.

To run a command from the script editor, place the cursor on the line that you want to run, then right-click and select 'Run line or selection'. You can also use the shortcut Ctrl+R. Alternatively you can click the run button, which looks like this:



To run several commands, highlight a selection of commands then right-click and select 'Run line or selection', as shown in Figure 1.3.



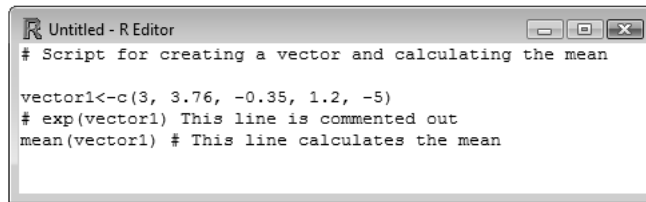
**Figure 1.3:** Running commands from a script file

Mac users can run the current line or a selection of commands by pressing Cmd+Return.

The selected commands are submitted to the command window and executed one after the other.

If your script file is going to be used by someone else or if you are likely to return to it after a long time, it is helpful to add some *comments*. Comments are additional text which are not part of the commands themselves but are used to make notes and explain the commands.

Add comments to your script file by typing the hash sign (#) before the comment. R ignores any text following a hash sign, for the remainder of the line. This means that if you run a section of commands that has comments in, the comments will not cause errors. Figure 1.4 shows a script file with comments.



**Figure 1.4:** Script file with comments

You can save a script file using by selecting 'File' then 'Save'. If the 'Save' option is not shown in the File menu, it is because you don't have focus on the script editor window and need to select it. The file is given the .R file name extension. Similarly, you can open a previously saved script file by selecting 'File' then 'Open script' and selecting the file in the usual manner.

Mac users can save a script file by selecting the icon in the top left-hand corner of the script editor window. They can open a previously saved script file by selecting 'Open Document' from the 'File' menu.

The Linux version of R does not include a script editor, however a number of external editors are available. To see a list of these, go to:

[www.sciviews.org/\\_rgui/projects/Editors.html](http://www.sciviews.org/_rgui/projects/Editors.html)

## Chapter summary: R fundamentals

The purpose of this chapter is to familiarise you with the R interface and the programming terms which will be used throughout the book. Make sure you understand the following terms before proceeding:

**R Console** The window into which you type your commands and in which output and any error or warning messages are displayed.

**Command** A typed instruction to R.

**Command prompt** The symbol used by R to indicate that it is ready to receive your command, which looks like this: >

**Function** A set of commands which have been given a name and together perform a specific task.

**Argument** A value or piece of data supplied to a function as input.

**Object** A piece of data or information which has been stored and given a name.

**Vector** An object that contains several data values of the same type arranged in a particular order.

**Data frame** A type of object which is suitable for holding a dataset.

**Workspace** The virtual area containing all of the objects created in the session, which can be saved to a file with the *.RData* file name extension.

**Script file** A file with the *.R* extension, which is used to save commands.

## Chapter 2

# Working with data files

---

This chapter covers the first step in performing any statistical analysis, which is to get your data in to R.

For very small datasets, it may be easiest to enter your data by typing the values in directly. This is explained in Section 2.1.

Section 2.2 explains how you can import plain text files, including the comma-separated value (CSV), tab-delimited and data interchange format (DIF) file formats.

Section 2.3 explains how you can import Excel<sup>®</sup> files by first converting them to the CSV format.

If you have a dataset stored in a file type specific to another software package such as an SPSS<sup>®</sup> or Stata<sup>®</sup> data file, Section 2.4 discusses how you can deal with it.

In addition to explaining data entry and import, the chapter covers other topics relevant to working with data files. Section 2.5 explains how to work with relative file paths. Section 2.6 explains how you can export a dataset to a CSV or tab-delimited file.

### 2.1 Entering data directly

If you have a small dataset which is not already recorded in electronic form, you may want to input your data into R directly.

Consider the dataset shown in Table 2.1, which gives some data for four UK supermarkets chains. It is representative of a typical dataset where the columns represent variables and each row holds one observation.

To enter a dataset in to R, the first step is to create a vector of data values for each variable using the `c` function, as explained in Section 1.6. So for the supermarkets data, input the four variables as shown.

```
> Chain<-c("Morrisons", "Asda", "Tesco", "Sainsburys")
> Stores<-c(439, NA, 2715, 934)
> Sales.Area<-c(12261, NA, 36722, 19108)
> Market.Share<-c(12.3, 16.9, 30.3, 16.5)
```

Chain	Stores	Sales Area (1,000 sq ft)	Market Share (%)
Morrisons	439	12261	12.3
Asda			16.9
Tesco	2715	36722	30.3
Sainsburys	934	19108	16.5

**Table 2.1:** Data for the UK's four largest supermarket chains (2011). See Appendix B for more details.

The vectors should all have the same length, meaning that they should contain the same number of values. Where a data value is missing, enter the characters `NA` in its place. Remember to put quotation marks around non-numeric values, as shown for the `Chain` variable.

Once you have created vectors for each of the variables, use the `data.frame` function to combine them to form a data frame.

```
> supermarkets<-data.frame(Chain, Stores, Sales.Area, Market.Share)
```

You can check the dataset has been entered correctly by entering its name.

```
> supermarkets
```

---

	Chain	Stores	Sales.Area	Market.Share
1	Morrisons	439	12261	12.3
2	Asda	NA	NA	16.9
3	Tesco	2715	36722	30.3
4	Sainsburys	934	19108	16.5

---

After you have created the data frame, the individual vectors still exist in the workspace as separate objects from the data frame. To avoid any confusion, you can delete them with the `rm` function.

```
> rm(Chain, Stores, Sales.Area, Market.Share)
```

## 2.2 Importing plain text files

The simplest way to transfer data to R is in a plain text file (sometimes called a *flat text file*). These are files that consist of plain text with no additional formatting and can be read by plain text editors such as Microsoft Notepad, TextEdit (for Mac users) or gedit (for Linux users). There are several standard formats for storing spreadsheet data in text files, which use symbols to indicate the layout of the data. These include:

- Comma-separated values or comma-delimited (.csv) files
- Tab-delimited (.txt) files
- Data interchange format (.dif) files.

These file formats are useful because they can be read by all of the popular statistical and database software packages, allowing easy transfer of datasets between them.

The following subsections explain how you can import datasets from these standard file formats into R.

### 2.2.1 CSV and tab-delimited files

Comma-separated values (CSV) files are the most popular way of storing spreadsheet data in a plain text file. In a CSV file, the data values are arranged with one observation per line and commas are used to separate data values within each line (hence the name). The tab-delimited file format is very similar to the CSV format except that the data values are separated with horizontal tabs instead of commas. Figure 2.1 shows how the supermarkets data looks in the CSV and tab-delimited formats.

You can import a CSV file with the `read.csv` function, as shown below:

```
> dataset1<-read.csv("C:/folder/filename.csv")
```

This command imports data from the file location `C:/folder/filename.csv` and saves it to a data frame called `dataset1`.

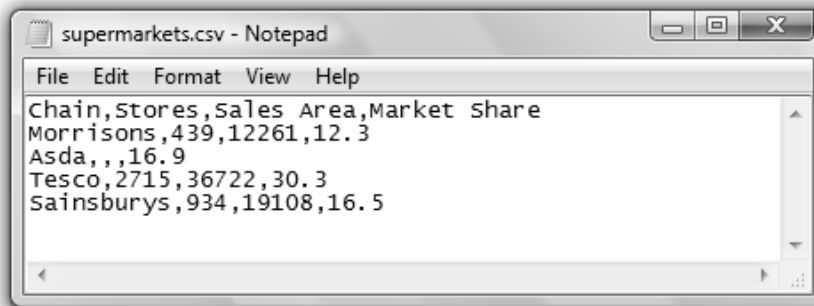
Remember that when choosing a dataset name, the usual object naming rules apply (see Section 1.5). It is a good idea to choose a short but meaningful name that describes the data in the file.

The file path `C:/folder/filename.csv` is the location of the CSV file on your hard drive, network drive or storage device. Notice that you must use forward slashes to indicate directories instead of the back slashes used by Windows, and that you must enclose the file path between quotation marks. For Mac and Linux users, the file path will begin with a forward slash e.g. `/Users/username/folder/filename.csv`.

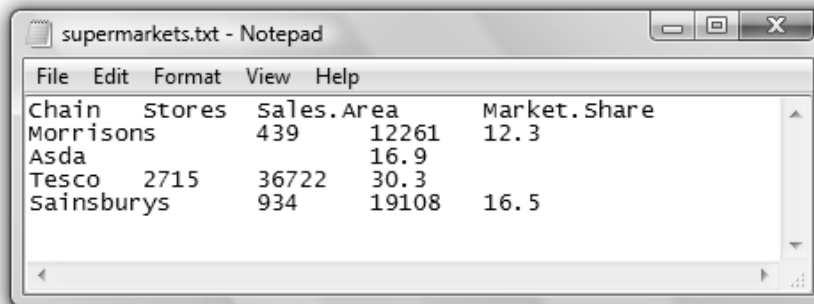
If the dataset has been successfully imported, there is no output and R displays the command prompt once it has finished importing the file. Otherwise you will see an error message explaining why the import failed. Assuming there are no issues, your data will be stored in the data frame named `dataset1`. You can view and check the data by typing the dataset name at the command prompt, or by opening it with the data editor.

For importing tab-delimited files, there is a similar function called `read.delim`.

```
> dataset1<-read.delim("C:/folder/filename.txt")
```



(a) CSV



(b) Tab-delimited

**Figure 2.1:** The supermarkets dataset saved in the CSV and tab-delimited file formats

When you use the `read.csv` or `read.delim` functions to import a file, R assumes that the entries in the first line of the file are the variable names for the dataset. Sometimes R will adjust the variable names so that they follow the naming rules (see Section 1.5) and are unique within the dataset.

If your file does not contain any variable names, set the `header` argument to `F` (for false) as shown below. This prevents R from using the first line of your data as the variable names.

```
> dataset1<-read.csv("C:/folder/filename.csv", header=F)
```

When you set the `header` argument to `F`, R assigns generic variable names of `V1`, `V2`, etc. Alternatively you can supply your own names with the `col.names` argument.

```
> dataset1<-read.csv("C:/folder/filename.csv", header=F,
  col.names=c("Name1", "Name2", "Name3"))
```

When using the `col.names` argument, make sure you give the same number of names as there



are variables in the file. Otherwise you will either see an error message or get an undesirable results as R attempts to match the variable names with the variables.

In a CSV and tab-delimited file, missing data is usually represented by an empty field. However some files may use a symbol or character such as a decimal point, the number 9999 or the word `NULL` as a place holder. If so, use the `na.strings` argument to tell R which characters to interpret as missing data.

```
> dataset1<-read.csv("C:/folder/filename.csv", na.strings=".")
```

Note that R automatically interprets the letters `NA` as indicating missing data. Blank spaces found in numeric variables (but not character variables) are also recognised as missing data.

In some parts of the world, a comma is used instead of a dot to denote the decimal point in numbers. R has two special functions for dealing with data in this form called `read.csv2` and `read.delim2`, which you can use in place of the `read.csv` and `read.delim` functions.

### **2.2.2 DIF files**

To import a data interchange format (DIF) file with the `.dif` file name extension, use the `read.DIF` function as shown.

```
> dataset1<-read.DIF("C:/folder/filename.dif")
```

By default, the `read.DIF` function assumes that there are no variable names in the file (unlike the `read.csv` and `read.delim` functions). If the file does contain variable names, set the `header` argument to `T` (for true) as shown below. Otherwise R will treat the variable names as part of the data values.

```
> dataset1<-read.DIF("C:/folder/filename.dif", header=T)
```

As when importing CSV and tab-delimited files, you can use the `na.strings` argument to tell R about any values that it should interpret as missing data.

```
> dataset1<-read.DIF("C:/folder/filename.dif", na.string="NULL")
```

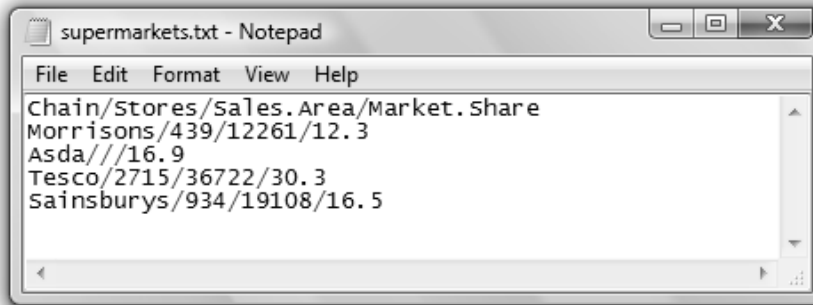
Note that if you are importing a DIF file that was created by Microsoft Excel, you may see the error message `More rows than specified in header; maybe use 'transpose=TRUE'` when you try to import the file. This error message appears because there are some differences in the file format used by Microsoft Excel. You can resolve the problem by setting the `transpose` argument to `T` as shown below.

```
> dataset1<-read.DIF("C:/folder/filename.dif", transpose=T)
```

### 2.2.3 Other plain text files

As well as all of the functions available for importing specific file formats, R also has a generic function for importing data from plain text files called `read.table`. It allows you to import any plain text file in which the data is arranged with one observation per line.

Consider the file shown in Figure 2.2, which has data arranged with one observation per line and data values separated by the forward slash symbol (/).



**Figure 2.2:** Supermarkets data in a non-standard file type

You could import the file with the command below.

```
> dataset1<-read.table("C:/folder/supermarkets.txt", sep="/",  
  header=T)
```

By default, the `read.table` function assumes that there are no variable names in the first row of the file (unlike the `read.csv` and `read.delim` functions). If the file has variable names in the first row (as in this example), set the `header` argument to `T`.

As with the other import functions, you can use the `na.strings` argument to tell R of any values to interpret as missing data.

```
> dataset1<-read.table("C:/folder/filename.txt", sep="/", header=T,  
  na.strings="NULL")
```

## 2.3 Importing Excel files

The simplest way to import a Microsoft Excel file is to save your Excel file as a CSV file which you can then import as explained in Section 2.2.1.

First open your file in Excel and ensure that the data is arranged correctly within the spreadsheet, with one variable per column and one observation per row. If the dataset includes variable names then these should be placed in the first row of the spreadsheet. Otherwise, the data values should begin on the first row. Figure 2.3 shows how the supermarkets data looks when correctly arranged in an Excel file.

	A	B	C	D	E	F
1	Chain	Stores	Sales Area	Market Share		
2	Morrisons	439	12261	12.3		
3	Asda			16.9		
4	Tesco	2715	36722	30.3		
5	Sainsburys	934	19108	16.5		
6						
7						
8						

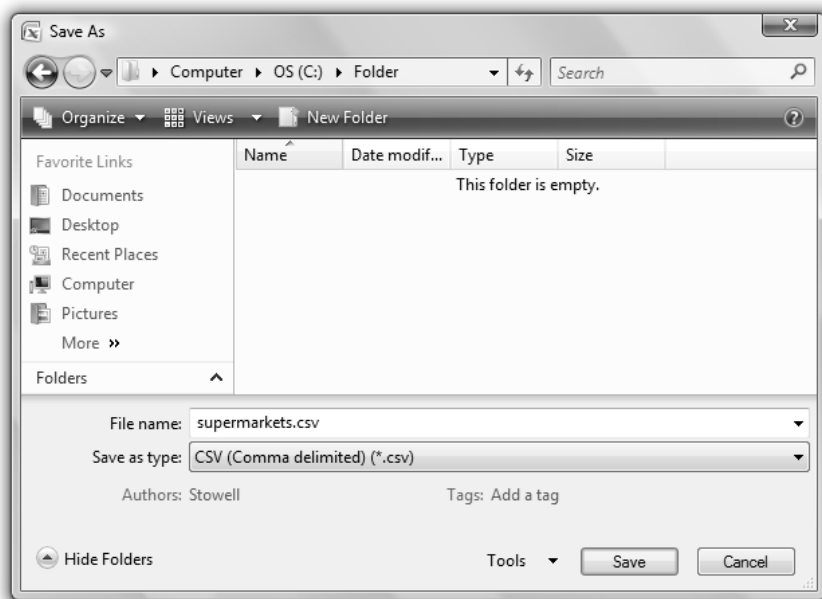
**Figure 2.3:** The correct way to arrange a dataset in an Excel spreadsheet, to facilitate easy conversion to the CSV file format

To ensure a smooth file conversion, check the following:

- There are no empty cells above or to the left of the data grid.
- There are no merged cells.
- There is no more than one row of column headers.
- Where data values are missing, the cell is left empty.
- There are no commas in large numbers (e.g. 1324157 is acceptable but 1,324,157 is not).
- If exponential (scientific) notation is used, the format is correct (e.g. 0.00312 can be expressed as 3.12e-3 or 3.12E-3).
- There are no currency, unit or percent symbols in numeric variables (symbols in categorical variables or in the variable names are fine).

- The minus sign is used to indicate negative numbers (e.g. -5) and not brackets (parenthesis) or red text.
- The workbook has only one worksheet.

Once the data is prepared, save the spreadsheet as a CSV file by selecting 'Save as' from the 'File' menu. When the save dialogue box appears, select the .csv file type from the 'Save as type' field, as shown in Figure 2.4. Then save the file as usual. Excel will give you a warning that all formatting will be lost, which you can accept.



**Figure 2.4:** Saving an Excel file as a CSV file

The CSV file is now ready for you to import with the `read.csv` function, as explained in Section 2.2.1.

If you do not have access to Excel, you can use an add-on package such as `xlsx` or `xlsReadWrite` to import Excel files directly. See Appendix A for more details on using add-on packages.

## 2.4 Importing files from other software

Sometimes you may need to import a dataset that is saved in a file format specific to another statistical package, such as an SPSS or Stata data file.

If you have access to the software, the simplest solution is to open the file using the software and convert the file to the CSV file format using the 'Save as' or 'Export' option, which is usually found in the 'File' menu. Once the file is in CSV format you can import it with the `read.csv` function, as explained in Section 2.2.1.

If you are not able to convert the file then you can use an add-on package called `foreign`, which allows you to directly import data from files types produced by some of the popular statistical software packages.

Add-on packages are covered in greater detail in Appendix A. For now, you just need to know that an add-on package contains additional functions which are not part of the standard R installation. To use the functions within the `foreign` package, you must first load the package.

To load the `foreign` package, select 'Load Package' from the 'Packages' menu. When the list of packages appears, select 'foreign' and press OK. Once the package has loaded, all of the functions within it will be available for you to use for the duration of the session.

Table 2.2 lists some of the functions available for importing foreign file types.

File type	Extension	Function
Database format file	.dbf	<code>read.dbf</code>
Stata v5.11 data file	.dta	<code>read.dta</code>
Minitab portable worksheet file	.mtp	<code>read.mtp</code>
SPSS data file	.sav	<code>read.spss</code>
SAS transfer format	.xport	<code>read.xport</code>
Epi Info data file	.rec	<code>read.epiinfo</code>
Octave text data file		<code>read.octave</code>
Attribute-relation file	.arff	<code>read.arff</code>
Systat file	.sys, .syd	<code>read.systat</code>

**Table 2.2:** Some of the functions available in the `foreign` add-on package<sup>28</sup>

For example to import a Stata data file, use the command:

```
> dataset1<-read.dta("C:/folder/filename.dta")
```

You may need to use additional arguments to ensure the file is imported correctly. For further information on using the functions in the `foreign` package, use the `help` function or refer to the package documentation available from the R project website at:

[cran.r-project.org/web/packages/foreign/foreign.pdf](http://cran.r-project.org/web/packages/foreign/foreign.pdf)

## 2.5 Using relative file paths

So far we have only used *absolute* file paths to describe the location of a data file. An absolute file path gives the full address of the file, which in the Windows environment begins with a drive name such as `C:/.`

You can also use *relative* file paths, which describe the location of the file in relation to the *working directory*. The working directory is the directory which R is set to look in when given relative file paths. This is useful if you need to import or export a large number of files and don't want to type the full file path each time. To see which is the current working directory, use the command:

```
> getwd()
```

If you are using a fresh installation of R for Windows, the working directory will be your 'My Documents' folder, and R will output something like this:

```
[1] "C:/Users/Username/Documents"
```

For Mac and Linux users, the default working directory will be your home directory and will look something like this:

```
[1] "/Users/Username"
```

So to import a CSV file which has the absolute file path:

```
C:/Users/Username/Documents/Data/filename.csv
```

you could use the command:

```
> read.csv("Data/filename.csv")
```

You can change the working directory to wherever you prefer to store your data files, use the `setwd` function as shown below. The change is applied for the remainder of the session.

```
> setwd("C:/folder/subfolder")
```

## 2.6 Exporting datasets

R allows you to export datasets from the R workspace to the CSV and tab-delimited files formats.

To export a data frame named `dataset` to a CSV file, use the `write.csv` function.

```
> write.csv(dataset, "filename.csv")
```

For example, to export the `Puromycin` dataset to a file named `puromycin_data.csv`, use the command:

```
> write.csv(Puromycin, "puromycin_data.csv")
```

This command creates the file and saves it to your working directory (see Section 2.5 for how to find and set the working directory). To save the file somewhere other than in the working directory, enter the full path for the file as shown.

```
> write.csv(dataset, "C:/folder/filename.csv")
```

If a file with your chosen name already exists in the specified location, R overwrites the original file without giving a warning. You should check the files in the destination folder beforehand to make sure you are not overwriting anything important.

The `write.table` function allows you to export data to a wider range of file formats, including tab-delimited files. Use the `sep` argument to specify which character should be used to separate the values. To export a dataset to a tab-delimited file, set the `sep` argument to `"\t"` (which denotes the tab symbol), as shown below.

```
> write.table(dataset, "filename.txt", sep="\t")
```

By default, the `write.csv` and `write.table` functions create an extra column in the file containing the observation numbers. To prevent this, set the `row.names` argument to `F`.

```
> write.csv(dataset, "filename.csv", row.names=F)
```

With the `read.table` function, you can also prevent variable names being placed in the first line of the file with the `col.names` argument.

```
> write.table(dataset, "filename.txt", col.names=F)
```

**Chapter summary: Working with data files**

Task	Command
Create data frame	<code>dataset&lt;-data.frame(vector1, vector2, vector3)</code>
Import CSV file	<code>dataset&lt;-read.csv("filepath")</code>
Import tab-delimited file	<code>dataset&lt;-read.delim("filepath")</code>
Import DIF file	<code>dataset&lt;-read.DIF("filepath")</code>
Import other text file	<code>dataset&lt;-read.table("filepath", sep="?")</code>
Display working directory	<code>getwd()</code>
Change working directory	<code>setwd("C:/folder/subfolder")</code>
Export dataset to CSV file	<code>write.csv(dataset, "filename.csv")</code>
Export dataset to tab-delimited file	<code>write.table(dataset, "filename.txt", sep="\t")</code>



## Chapter 3

# Preparing and manipulating your data

Once you have imported your dataset, it is likely that you will need to make some changes before beginning any statistical analysis. This could be tidying the dataset by renaming variables, removing irrelevant data and sorting the observations. Alternatively you may need to adjust the data type for some of the variables, or create new variables from the existing ones. This chapter explains how you can make these types of changes to a dataset. More complex changes, such as combining two datasets or changing the structure of the data, are covered in Chapter 4.

This chapter uses the `people` dataset shown in Figure 3.1 for demonstration purposes. This dataset gives the eye colour (brown, blue or green), height in centimetres, hand span in millimetres, sex (1 for male, 2 for female) and handedness (L for left-handed, R for right-handed) of 16 people.

	Subject	Eye.Colour	Height	Hand.Span	Sex	Handedness
1	1	Brown	186	210	1	R
2	2	Green	182	220	1	R
3	3	Brown	147	167	2	
4	4	Green	157	180	2	L
5	5	Brown	170	193	1	R
6	6	Blue	169	190	2	L
7	7	brown	174	217	1	R
8	8	Blue	173	211	1	R
9	9	Blue	166	193	2	R
10	10	Blue	166	178	2	R
11	11	Brown	163	223	1	R
12	12	Blue	184	225	1	R
13	13	Blue	176	214	1	
14	14	Blue	183	218	1	R
15	15	Green	160	190	2	
16	16	Brown	173	196	1	R

**Figure 3.1:** The `people` dataset. See Appendix B for more details.

This chapter also uses the `pulserates`, `fruit`, `flights`, `customers` and `coffeeshop` datasets, which are all available from the website in CSV format or in an R workspace file. For more information about these datasets, see Appendix B.

## 3.1 Rearranging and removing variables

You can rearrange or remove the variables in a dataset with the `subset` function. Use the `select` argument to choose which variables to *keep* and in which order. Remove unwanted variables by excluding them from the list.

For example, the command below removes the `Subject` and `Height` variables from the `people` dataset, and rearranges the remaining variables so that `Hand.Span` is first, followed by `Sex` then `Eye.Colour`.

```
> people1<-subset(people, select=c(Hand.Span, Sex, Eye.Colour))
```

Figure 3.2 shows how the new dataset looks after the changes have been applied.

	Hand.Span	Sex	Eye.Colour
1	210	1	Brown
2	220	1	Green
3	167	2	Brown
4	180	2	Green
5	193	1	Brown
6	190	2	Blue
7	217	1	brown
8	211	1	Blue
9	193	2	Blue
10	178	2	Blue
11	223	1	Brown
12	225	1	Blue
13	214	1	Blue
14	218	1	Blue
15	190	2	Green
16	196	1	Brown

**Figure 3.2:** The `people1` dataset, created by removing variables from the `people` dataset with the `subset` function.

Notice that the command creates a new dataset called `people1` which is a modified version of the original, and leaves the original dataset unchanged. Alternatively you can overwrite the original dataset with the modified version as shown below.

```
> people<-subset(people, select=c(Hand.Span, Sex, Eye.Colour))
```

The `subset` function does more than remove and rearrange variables. You can also use it to select a subset of observations from a dataset, which is explained in Section 3.11.

Another way of removing variables from a dataset is with bracket notation. This is particularly useful

if you have a dataset with a large number of variables and you only want to remove a few. For example, to remove the first, third and sixth variables from the `people` dataset, use the command:

```
> people1<-people[-c(1,3,6)]
```

Similarly to retain the second, fourth and first variables and reorder them, use the command:

```
> people1<-people[c(2,4,1)]
```

See Section 1.7 for more details on using bracket notation.

## 3.2 Renaming variables

The `names` function displays a list of the variable names for a dataset.

```
> names(people)
```

---

```
[1] "Subject"      "Eye.Colour"  "Height"      "Hand.Span"   "Sex"         "Handedness"
```

---

You can also use the `names` function to rename variables. The command below renames the fifth variable in the `people` dataset.

```
> names(people)[5]<-"Gender"
```

Similarly to rename the second, fourth and fifth variables:

```
> names(people)[c(2,4,5)]<-c("Eyes", "Span.mm", "Gender")
```

Alternatively you can rename all of the variables in the dataset simultaneously as shown.

```
> names(people)<-c("Subject", "Eyes", "Height.cm", "Span.mm",
  "Gender", "Hand")
```

Make sure that you provide the same number of variable names as there are variables in the dataset.

## 3.3 Variable classes

Each of the variables in a dataset has a *class*, which describes the type of data the variable contains. You can view the class of a variable with the `class` function.

```
> class(dataset$variable)
```

To check the class of all the variables simultaneously, use the command:

```
> sapply(dataset, class)
```

A variable's class determines how R will treat the variable when you use it in statistical analysis and plots. There are many possible variable classes in R, but only a few that you are likely to use:

**numeric** variables contain *real* numbers, meaning positive or negative numbers with or without a decimal point. They can also contain the missing data symbol (NA).

**integer** variables contain positive or negative numbers without a decimal point. This class behaves in much the same way as the numeric class. An integer variable is automatically converted to a numeric variable if a value with a fractional part is included.

**factor** variables are suitable for categorical data. Factor variables generally have a small number of unique values, known as *levels*. The actual values can be either numbers or character strings.

**date & POSIXlt** variables contain dates or date-times in a special format which is convenient to work with.

**character** variables contain character strings. A character string is any combination of unicode characters including letters, numbers and symbols. This class is suitable for any data which does not belong to one of the other classes, such as reference numbers, labels and text giving additional comments or information.

When you import a data file using a function such as `read.csv`, R automatically assigns each variable a class based on its contents. If a variable contains only numbers, R assigns the `numeric` or `integer` class. If a variable contains any non-numeric values, it assigns the `factor` class.

Since R does not know how you intend to use the data contained in each variable, the classes that it assigns to them may not always be appropriate. To illustrate, consider the `Sex` variable in the `people` dataset. Since the variable contains whole numbers, R automatically assigns the `integer` class when the data is imported. But the `factor` class would be more appropriate, as the values represent categories rather than counts or measurements.

You can change the class of a variable to `factor` with the `as.factor` function.

```
> dataset$variable<-as.factor(dataset$variable)
```

If you have a variable containing numeric values which for some reason has been assigned another class, you can change it using the `as.numeric` function. Any non-numeric values are treated as missing data and replaced with the missing data code (NA).

```
> dataset$variable<-as.numeric(dataset$variable)
```

If R has not automatically recognised a variable as numeric when importing a dataset, then it is because the variable contains at least one non-numeric value. It is wise to determine the cause, as

it may be that a value has been entered incorrectly or that a symbol used to represent missing data has not been recognised.

You can change the class of a variable to `character` using the `as.character` function.

```
> dataset$variable<-as.character(dataset$variable)
```

There is also an `as.Date` function for creating date variables, which you will learn more about in Section 3.8.

## 3.4 Calculating new numeric variables

You can create a new variable within a dataset in the same way you would create any other new object, using the assignment operator. So to create a new variable named `var2` that is a copy of an existing variable named `var1`, use the command:

```
> dataset$var2<-dataset$var1
```

You can create new numeric variables from combinations of existing numeric variables and arithmetic operators and functions. For example, the command below adds a new variable called `Height.Inches` to the `people` dataset, which gives the subject's heights in inches rounded to the nearest inch.

```
> people$Height.Inches<-round(people$Height/2.54)
```

You can use bracket notation to make conditional changes to a variable. For example, to set all values of `Height` less than 150 cm to missing, use the command:

```
> people$Height[people$Height<150]<-NA
```

You will learn more about using conditions in Sections 3.11 and 12.2.1.

You may want to create a new variable which is a statistical summary of several of the existing variables. The `apply` function allows you to do this.

Consider the `pulserates` dataset shown in Figure 3.3 which gives pulse rate data for four patients. The patients' pulse rates are measured in triplicate and stored in variables `Pulse1`, `Pulse2` and `Pulse3`.

Suppose that you want to calculate a new variable giving the mean pulse for each patient. You can create the new variable with the command:

```
> pulserates$Mean.Pulse<-apply(pulserates[2:4], 1, mean)
```

	Patient	Pulse1	Pulse2	Pulse3	
1	3051	70	77	73	Mean.Pulse
2	3052	65	66	56	73.33333
3	3053	64	58	60	62.33333
4	3054	53	58	56	60.66667
					55.66667

(a) Original dataset

(b) New variable

**Figure 3.3:** `pulserates` dataset giving the pulse rates of four patients, measured in triplicate. See Appendix B for more details.

Notice that bracket notation is used to select column numbers 2 to 4. See Section 1.7 for more details on using bracket notation.

The second argument allows you to specify whether the summary should be calculated for each row (by setting it to 1) or each column (by setting it to 2). To create a new variable, set it to 1.

You can substitute the `mean` function with any univariate statistical summary function that gives a single value as output, such as `sd` or `max`. Table 5.1 on page 60 gives a list of these (use only those marked with an asterisk).

### 3.5 Dividing a continuous variable into categories

Sometimes you may want to create a new categorical variable by classifying the observations according to the value of a continuous variable.

For example, consider the `people` dataset shown on page 31. Suppose that you want to create a new variable called `Height.Cat` which classifies the people as 'Short', 'Medium' and 'Tall' according to their height. People less than 160 cm tall are classified as 'Short', people between 160 cm and 180 cm tall are classified as 'Medium' and people greater than 180 cm tall are classified as 'Tall'.

You can create the new variable with the `cut` function, as shown.

```
> people$Height.Cat<-cut (people$Height, c(150, 160, 180, 200),
  c("Short", "Medium", "Tall"))
```

Figure 3.4 shows the `people` dataset with the new `Height.Cat` variable.

When using the `cut` function, the numbers of group boundaries (in this example four) must be one more than the number of group names (in this example three). If a data value is equal to one of the boundaries, it is placed in the category below. Make sure your categories cover the whole range of the data values, otherwise the new variable will have missing values. In this example, there is one observation (subject 3) that does not fall in to any of the categories that have been defined, so has a missing value for the `Height.Cat` variable.

	Subject	Eye.Colour	Height	Hand.Span	Sex	Handedness	Height.Cat
1	1	Brown	186	210	1	R	Tall
2	2	Green	182	220	1	R	Tall
3	3	Brown	147	167	2		
4	4	Green	157	180	2	L	Short
5	5	Brown	170	193	1	R	Medium
6	6	Blue	169	190	2	L	Medium
7	7	brown	174	217	1	R	Medium
8	8	Blue	173	211	1	R	Medium
9	9	Blue	166	193	2	R	Medium
10	10	Blue	166	178	2	R	Medium
11	11	Brown	163	223	1	R	Medium
12	12	Blue	184	225	1	R	Tall
13	13	Blue	176	214	1		Medium
14	14	Blue	183	218	1	R	Tall
15	15	Green	160	190	2		Short
16	16	Brown	173	196	1	R	Medium

Figure 3.4: The people dataset with the new Height.cat variable.

If you prefer, you can specify the number of categories and let R determine where the boundaries should be. R divides the range of the variable to create evenly sized categories. For example, the command below shows how you would split the Height variable into three evenly sized categories.

```
> people$Height.Cat<-cut (people$Height, 3, c("Short", "Medium",
      "Tall"))
```

Any variables you create with the cut function are automatically assigned the factor class.

## 3.6 Working with factor variables

As explained in Section 3.3, factor variables are suitable for holding categorical data. To change the class of a variable to factor, use the as.factor function.

```
> people$Sex<-as.factor (people$Sex)
```

A factor variable has a number of levels, which are all of the unique values that the variable takes (i.e. all of the possible categories). To view the levels of a factor variable, use the levels function:

```
> levels (people$Sex)
[1] "1" "2"
```

Since the level names will appear on any plots and statistical output that you create based on the variable, it is helpful if they are meaningful and attractive. You can change the names of the levels as shown.

```
> levels(people$Sex) <- c("Male", "Female")
```

You must give the same number of names as there are levels of the factor, and enter the new names in corresponding order.

You can also combine factor levels by renaming them. Consider the `Eye.Colour` variable in the `people` dataset. Using the `levels` function, you can see that there is an extra level resulting from a spelling variation.

```
> levels(people$Eye.Colour)
[1] "Blue" "brown" "Brown" "Green"
```

To rename the second factor level so that it has the correct spelling, use the command:

```
> levels(people$Eye.Colour)[2] <- "Brown"
```

When the factor levels are viewed again, you can see that the two levels have been combined.

```
> levels(people$Eye.Colour)
[1] "Blue" "Brown" "Green"
```

You can change the order of the levels with the `relevel` function. For example, to make `Brown` the first level of the `Eye.Colour` variable, use the command:

```
> people$Eye.colour <- relevel(people$Eye.Colour, "Brown")
```

The order of the factor levels is important because if you include the factor in a statistical model, R uses the first level of the factor as the *reference level*. You will learn more about this in Chapter 11.

## 3.7 Manipulating character variables

R has a number of functions for manipulating character strings. Three of the most useful are `paste` (for concatenating strings), `substring` (for extracting a substring) and `grep` (for searching a string). These are demonstrated in the following subsections.

### 3.7.1 Concatenating character strings

The `paste` function allows you to create new character variables by pasting together existing variables (of any class) and other characters.



	Product	Price	Unit	Label
1	Apricots	10	per kg	Apricots: £10.00 per kg
2	Baby Watermelon	2	each	Baby Watermelon: £2.00 each
3	Bananas	0.68	per kg	Bananas: £0.68 per kg
4	Blush Pears	2.1	per kg	Blush Pears: £2.10 per kg
5	Braeburn Apples	1.65	per kg	Braeburn Apples: £1.65 per kg
6	Bramley Cooking Apples	1.55	per kg	Bramley Cooking Apples: £1.55 per kg
7	Cantaloupe Melon	2	each	Cantaloupe Melon: £2.00 each

(a) Original dataset

(b) New variable

**Figure 3.5:** fruit dataset giving UK fruit prices for August 2012. See Appendix B for more details.

Consider the `fruit` dataset shown in Figure 3.5a, which gives prices for a selection of fruit. Suppose you want to create a new variable giving a price label for each of the fruit. The label should have the product description, price with pound sign and sale unit.

You can create the new variable (shown in Figure 3.5b) with the command:

```
> fruit$Label<-paste(fruit$Product, ": £", format(fruit$Price,
  trim=T, digits=2), " ", fruit$Unit, sep="")
```

By default, the `paste` function inserts a space between each of the components being pasted together. In this example, `sep=""` has been added to prevent this, so that spaces are not inserted in unwanted places such as between the pound sign and the price. Instead spaces have been inserted manually where required, placed between quotation marks. You can also use the `sep` argument to specify another keyboard symbol to use as a separator.

Notice that the `format` function has been used to ensure that the fruit prices are always displayed to two decimal places.

### 3.7.2 Extracting a substring

The `substring` function allow you to create a new variable by extracting a section of characters from an existing variable.

Consider the `flights` dataset shown in Figure 3.6a. The `Flight.Number` variable gives flight numbers that begin with a two-letter prefix indicating which airline the flight is operated by. Suppose that you wish to create two new variables, one named `Airline` giving the two-letter airline prefix and another named `Ref` giving the number component, as shown in Figure 3.6b.

When using the `substring` function, give the positions within the character string of the first and last characters you want to extract. So to extract the first two characters from the `Flight.Number` variable to create the `Airline` variable, use the command:

```
> flights$Airline<-substring(flights$Flight.Number, 1, 2)
```

	Date	Time	Flight.Number	Destination	Airline	Ref
1	12/01/2012	20:30	BE898	GLASGOW	BE	898
2	12/01/2012	20:35	BE775	EDINBURGH	BE	775
3	13/01/2012	06:50	BE382	DUBLIN	BE	382
4	13/01/2012	07:00	BE861	MANCHESTER	BE	861
5	13/01/2012	07:05	BE1011	AMSTERDAM	BE	1011
6	13/01/2012	08:00	AF6541	RENNES	AF	6541
7	13/01/2012	08:00	BE3025	RENNES	BE	3025

(a) Original dataset

(b) New variables

**Figure 3.6:** `flights` dataset giving details of flights from Southampton Airport. See Appendix B for more details.

You can also give a starting position only, and the remainder of the string is extracted. So to create the `Ref` variable, use the command:

```
> flights$Ref<-substring(flights$Flight.Number, 3)
```

Note that although the new `Ref` variable contains numbers, it still has the `character` class because it was created with the `substring` function. If you wanted to use it as a numeric variable, you can convert it with the `as.numeric` function as described in Section 3.3.

### 3.7.3 Searching a character variable

The `grep` function allows you to search a character string for a search term.

Consider the `customers` dataset shown in Figure 3.7. Suppose that you want to identify all of the customer that live in the city of Reading.

	Name	Address
1	Mr A. Jackson	17 Bridge Street, Reading, RG3 2QN
2	Ms D. Phillips	112 park avenue, reading, berkshire, RG21NY
3	Mrs S. O'Neill	Flat 1, 72 Norfolk Road, Maidenhead, SL6 7AZ
4	Mr D. Singh	373 Castle Lane, READING, RG5 2LL
5	Mr A. Rojas	2 Green Lane, Wokingham, RG40 2NA

**Figure 3.7:** The `customers` dataset. See Appendix B for more details.

The command below searches the `Address` variable for the term 'reading'.

```
> grep("reading", customers$Address)
[1] 2
```

R outputs the number 2 to indicate that observation number 2 contains the term 'reading'.

Notice that R has only returned one result because the search is case sensitive, so that 'Reading' and 'READING' are not considered matches to the search term 'reading'. To change this, set the `ignore.case` argument to `T`.

```
> grep("reading", customers$Address, ignore.case=T)
[1] 1 2 4
```

Instead of displaying the observation numbers, you can save them to an object. This allows you to create a new dataset containing only the observations that matched the search term, as shown.

```
> matches<-grep("reading", customers$Address, ignore.case=T)
> reading.customers<-customers[matches,]
```

## 3.8 Working with dates and times

R has special date and date-time variable classes that make this type of data easier to work with. When you import a dataset, R does not automatically recognise date variables. Instead they are assigned one of the other classes according to their contents. You can convert these variables with the `as.Date` and `strptime` functions.

For variables containing just dates (without times), use the `as.Date` function to convert the variable to the `date` class. The command takes the form:

```
> dataset$variable<-as.Date(dataset$variable, "format")
```

where `"format"` tells R how to read the dates. R uses a special notation for specifying the format of a date, shown in Table 3.1.

Consider the `Date` variable in the `coffeeshop` dataset, shown in Figure 3.8. The variable has dates in the format `dd/mmm/yyyy`.

	Date	Sales
1	21/NOV/2011	2144.88
2	22/NOV/2011	1702.99
3	23/NOV/2011	2731.45
4	24/NOV/2011	1943.04
5	25/NOV/2011	1862.09

**Figure 3.8:** `coffeeshop` dataset. See Appendix B for more details.

To convert the variable to the `date` class, use the command:

```
> coffeeshop$Date<-as.Date(coffeeshop$Date, "%d/%b/%Y")
```

Symbol	Meaning	Possible values
%d	Day of the month	01 to 31, or 1 to 31
%m	Month number	01 to 12
%b	Three letter abbreviated month name	Jan, Feb, Mar, Apr, etc
%B	Full month name	January, February, March, April, etc
%y	Two digit year	00 to 99, e.g. 10 for 2010
%Y	Four digit year	e.g. 2004
%H	Hour in 24-hour format	00 to 23, e.g. 19 for 7pm
%M	Minute past the hour	00 to 59
%S	Seconds past the hour	00 to 59
%I	Hour in 12-hour format	01 to 12
%p	AM or PM	AM or PM

**Table 3.1:** The most commonly used symbols for date-time formats.<sup>32</sup> Enter `help(strptime)` to view a complete list.

The format `%d/%b/%Y` tells R to expect a day (`%d`), three letter month name (`%b`), and four digit year (`%Y`), seperated by forward slashes. The format must be enclosed in quotation marks.

There are two date formats which R recognises without you needing to specify them, which are `yyyy-mm-dd` and `yyyy/mm/dd`. If your dates are in either of these formats, then you don't need to give a format when using the `as.Date` function.

For variables containing dates with time information, use the `strptime` function to convert the variable to the `POSIXlt` class.

For example, suppose you want to create a new date-time variable from the `Date` and `Time` variables in the `flights` dataset (see Figure 3.6 on page 40). Before you can create a date-time variable, you will need to combine the two variables to create a single character variable using the `paste` function (see Section 3.7.1).

```
> flights$DateTime<-paste(flights$Date, flights$Time)
```

Once the date and time are together in the same character string, you can use the `strptime` function to convert the variable class. The `strptime` function is used in the same way as the `as.Date` function.

```
> flights$DateTime<-strptime(flights$DateTime, "%d/%m/%Y %H:%M")
```

Once your variable has the `date` or `POSIXlt` class, you can perform a number of date related operations using functions designed for these variable classes.

For example, to find the length of the time interval between two dates or date-times, use the `difftime` function as shown below.

```
> dataset$duration<-difftime(dataset$enddate, dataset$startdate,
  units="hours")
```

Options for the `units` argument are `secs`, `mins`, `hours`, `days` (the default) and `weeks`.

To compare a date variable with the current date (e.g. to calculate an age), use the `Sys.Date` function.

```
> dataset$age<-difftime(Sys.Date(), dataset$dob)
```

You can use arithmetic operators to add or subtract days (for date variables) or seconds (for POSIXlt variables). For example, to find the date one week before a given date.

```
> dataset$newdatevar<-dataset$datevar-7
```

To find which day of the week a date falls on, use the `weekdays` function. There are also similar functions called `months` and `quarters`.

```
> coffeeshop$Day<-weekdays(coffeeshop$Date)
```

The `round` function can also be used with date-time variables. Specify one of the time units `secs`, `mins`, `hours` or `days` as shown.

```
> round(flights$DateTime, units="hours")
```

You can create a character variable from a date variable with the `format` function. Specify how you want R to display the date using the format symbols given in Table 3.1.

```
> dataset$charvar<-format(dataset$datevar, format="%d.%m.%Y")
```

## 3.9 Adding and removing observations

The simplest way to add additional observations to a dataset is with the data editor, which you can open with the command:

```
> fix(dataset)
```

When the editor window opens, type the values for the new observation into the first empty row beneath the existing data. If any of the values are missing, leave the relevant cell empty. When you have finished adding new values, close the data editor to apply the changes.

The simplest way to remove individual observations from a dataset is using bracket notation (see Section 1.7). For example, to remove observation numbers 2, 4 and 7 use the command:

```
> dataset<-dataset[-c(2,4,7),]
```

Be careful to include the comma before the closing bracket, otherwise you will remove columns rather than rows.

Remember that you can also use the colon symbol (:) to select a range of consecutive observations. For example, to remove observation numbers 2 to 10 use the command:

```
> dataset<-dataset[-c(2:10),]
```

To remove all of the observations which match a specified criteria or belong to a particular group, use the `subset` function as explained in Section 3.11.

### 3.10 Removing duplicate observations

To remove duplicate observations from a dataset, use the `unique` function.

```
> dataset<-unique(dataset)
```

To save the duplicates to a separate dataset before removing them, use the `duplicated` function as shown below.

```
> dups<-dataset[duplicated(dataset),]
```

### 3.11 Selecting a subset of the data

Sometimes you may need to select a subset of a dataset containing only those observations that match certain criteria, such as belonging to a particular category or where the value of one of the numeric variables falls within a given range. You can do this with the `subset` function. The command takes the general form:

```
> subset(dataset, condition)
```

For example, to select all of the observations from the `people` dataset where the value of the `Eye.Colour` variable is Brown, use the command:

```
> subset(people, Eye.Colour=="Brown")
```

---

	Subject	Eye.Colour	Height	Hand.Span	Sex	Handedness
1	1	Brown	186	210	1	R
3	3	Brown	147	167	2	
5	5	Brown	170	193	1	R
11	11	Brown	163	223	1	R
16	16	Brown	173	196	1	R

---

Notice that you must use two equals signs rather than one.

To save the selected observations to a new dataset, assign the output to a new dataset name as shown.

```
> browneyes<-subset (people, Eye.Colour=="Brown")
```

To select all the observations for which a variable takes any one of a list of values, use the `%in%` operator. For example, to select all observations where `Eye.Colour` is either `Brown` or `Green`, use the command:

```
> subset (people, Eye.Colour %in% c("Brown", "Green"))
```

---

	Subject	Eye.Colour	Height	Hand.Span	Sex	Handedness
1	1	Brown	186	210	1	R
2	2	Green	182	220	1	R
3	3	Brown	147	167	2	
4	4	Green	157	180	2	L
5	5	Brown	170	193	1	R
11	11	Brown	163	223	1	R
15	15	Green	160	190	2	
16	16	Brown	173	196	1	R

---

To select observations to exclude instead of to include, replace `==` with `!=` (which mean 'not equal to'). For example, to exclude all observations where the value of `Eye.Colour` is equal to `"Blue"`, use the command:

```
> subset (people, Eye.Colour!="Blue")
```

You can also select observations according to the value of a numeric variable. For example to select all observations from the `people` dataset where the `Height` variable is equal to 169, use the command:

```
> subset (people, Height==169)
```

---

	Subject	Eye.Colour	Height	Hand.Span	Sex	Handedness
6	6	Blue	169	190	2	L

---

Notice that quotation marks are not required for numeric values.

With numeric variables you can also use *relational operators* to select observations. For example, to select all observations for which the value of the `Height` variable is less than 165, use the command:

```
> subset (people, Height<165)
```

---

	Subject	Eye.Colour	Height	Hand.Span	Sex	Handedness
3	3	Brown	147	167	2	
4	4	Green	157	180	2	L
11	11	Brown	163	223	1	R
15	15	Green	160	190	2	

---

Other relational operators you could use are `>` (greater than), `>=` (greater than or equal to) and `<=` (less than or equal to).

You can also combine two or more conditions using the *AND* operator (denoted `&`) and the *OR* operator (denoted `|`). When two criteria are joined with the AND operator, R selects only those observations which meet both conditions. When they are joined with the OR operator, R selects the observations which meet either one of the conditions, or both.

For example, to select observations where `Eye.Colour` is Brown *and* `Height` is less than 165, use the command:

```
> subset (people, Eye.Colour=="Brown" & Height<165)
```

---

	Subject	Eye.Colour	Height	Hand.Span	Sex	Handedness
3	3	Brown	147	167	2	
11	11	Brown	163	223	1	R

---

As well as selecting a subset of observations from the dataset, you can also use the `select` argument to select which variables to keep.

```
> subset (people, Height<165, select=c (Hand.Span, Height))
```

---

	Hand.Span	Height
3	167	147
4	180	157
11	223	163
15	190	160

---

You can also subset a dataset using bracket notation. For example, the command below selects only those people with brown eyes.

```
> people[people$Eye.Colour=="Brown",]
```

Note that it is not always necessary to subset a dataset before performing analysis. Many analysis functions have a `subset` argument within the function. This allows you to perform the analysis for a subset of the data. For example, the command below creates a scatter plot of height against hand span, showing only males (i.e. where `Sex` is equal to 2).

```
> plot (Height~Hand.Span, people, subset=Sex==2)
```

You will learn more about scatter plots in Section 8.7.



## 3.12 Selecting a random sample from a dataset

To select a random sample of observations from a dataset, use the `sample` function. For example, the following command selects a random sample of 50 observations from a dataset named `dataset` and saves them to new dataset named `sampladata`.

```
> sampladata<-dataset[sample(1:nrow(dataset), 50),]
```

By default, the `sample` function samples without replacement, so that no observation can be selected more than once. For this reason, the sample size must be less than the number of observations in the dataset. To sample *with* replacement, set the `replace` argument to `T` as shown.

```
> sampladata<-dataset[sample(1:nrow(dataset), 50, replace=T),]
```

## 3.13 Sorting a dataset

You can use the `order` function to sort a dataset. For example, to sort the `people` dataset by the `Hand.Span` variable, use the command:

```
> people<-people[order(people$Hand.Span),]
```

---

	Subject	Eye.Colour	Height	Hand.Span	Sex	Handedness
3	3	Brown	147	167	2	
10	10	Blue	166	178	2	R
4	4	Green	157	180	2	L
6	6	Blue	169	190	2	L
15	15	Green	160	190	2	
5	5	Brown	170	193	1	R
9	9	Blue	166	193	2	R
16	16	Brown	173	196	1	R
1	1	Brown	186	210	1	R
8	8	Blue	173	211	1	R
13	13	Blue	176	214	1	
7	7	brown	174	217	1	R
14	14	Blue	183	218	1	R
2	2	Green	182	220	1	R
11	11	Brown	163	223	1	R
12	12	Blue	184	225	1	R

---

To sort in decreasing instead of ascending order, set the `decreasing` argument to `T` as shown.

```
> people<-people[order(people$Hand.Span, decreasing=T),]
```

You can also sort by more than one variable. To sort the dataset first by `Sex` and then by `Height`, use the command:

```
> people<-people[order(people$Sex, people$Height),]
```

**Chapter summary: Preparing and manipulating your data**

Task	Command
Rename variable	<code>names(dataset)[n] &lt;- "Newname"</code>
View variable class	<code>class(dataset\$variable)</code>
Change variable class to numeric	<code>dataset\$var1 &lt;- as.numeric(dataset\$var1)</code>
Change variable class to factor	<code>dataset\$var1 &lt;- as.factor(dataset\$var1)</code>
Change variable class to character	<code>dataset\$var1 &lt;- as.character(dataset\$var1)</code>
Change variable class to date	<code>dataset\$var1 &lt;- as.Date(dataset\$var1, "format")</code>
Copy variable	<code>dataset\$var2 &lt;- dataset\$var1</code>
Divide variable into categories	<code>dataset\$factor1 &lt;- cut(dataset\$var1, c(1,2,3,4), c("Name1", "Name2", "Name3"))</code>
Rename factor level	<code>levels(dataset\$variable)[n] &lt;- "Newname"</code>
Reorder factor levels	<code>relevel(dataset\$variable, "Level1")</code>
Join two character strings	<code>dataset\$var3 &lt;- paste(dataset\$var1, dataset\$var2)</code>
Extract a substring	<code>dataset\$var2 &lt;- substring(dataset\$var1, first, last)</code>
Search character variable	<code>grep("search term", dataset\$variable)</code>
Remove cases	<code>dataset &lt;- dataset[-c(2,4,7),]</code>
Remove duplicates	<code>dataset &lt;- unique(dataset)</code>
Select subset	<code>subset(dataset, variable=="value")</code>
Select random sample	<code>newdataset &lt;- dataset[sample(1:nrow(dataset), samplesize),]</code>
Sort dataset	<code>dataset &lt;- dataset[order(dataset\$variable),]</code>

## Chapter 4

# Combining and restructuring datasets

---

This chapter explains how to combine two or more datasets to create one large dataset, and how to change the structure of a dataset.

There are several ways of combining datasets in R. Section 4.1 explains how to attach one dataset to the bottom of another, which is known as *appending* or *concatenating*. Section 4.2 explains how to attach one dataset to the side of another. Section 4.3 explains how you can merge two datasets by using a common variable to match the observations.

Restructuring a dataset means to change its shape by combining several variables into one long variable, or creating several shorter variables from one long one. Stacking and unstacking datasets is covered in Sections 4.4 and 4.5. Reshaping (also known as *rotating* or *transposing*) is covered in Section 4.6.

This chapter uses the `CIAdatal`, `CIAdata2`, `WHOdata`, `CPIData`, `bigcats`, `endangered`, `grades1`, `resistance` and `vitalsigns` datasets, which are available from the website. For more details about these datasets, see Appendix B. It also uses the `iris` dataset, which is included with R. To view more information about this dataset, enter `help(iris)`.

### 4.1 Appending rows

This `rbind` function allows you to attach one dataset on to the bottom of the other, which is known as *appending* or *concatenating* the datasets. This is useful when you want to combine two datasets that contain different observations for the same variables, such as the `CIAdatal` and `CIAdata2` datasets shown in Figure 4.1a and 4.1b.

Before using the `rbind` function, make sure that each dataset contains the same number of variables and that all of the variable names match. You may need to remove or rename some variables first, as explained in Sections 3.1 and 3.2. The variables do not need to be arranged in the same order within the datasets, as the `rbind` function automatically matches them by name.

Once the datasets are prepared, append them with the `rbind` function as shown below for the `CIAdatal` and `CIAdata2` datasets.

	country	lifeExp	urban	pcGDP
1	Finland	79.41	85	36700
2	Slovakia	76.03	55	23600
3	UK	80.17	80	36600
4	Ukraine	68.74	69	7300
5	Spain	81.27	77	31000

(a) CIAdata1

	country	pcGDP	lifeExp	urban
1	Italy	30900	81.86	68
2	Croatia	18400	75.99	58
3	Slovakia	23600	76.03	55

(b) CIAdata2

	country	lifeExp	urban	pcGDP
1	Finland	79.41	85	36700
2	Slovakia	76.03	55	23600
3	UK	80.17	80	36600
4	Ukraine	68.74	69	7300
5	Spain	81.27	77	31000
6	Italy	81.86	68	30900
7	Croatia	75.99	58	18400
8	Slovakia	76.03	55	23600

(c) CIAdata

**Figure 4.1:** Data from the CIA World Factbook. See Appendix B for more details.

```
> CIAdata<-rbind(CIAdata1, CIAdata2)
```

The new `CIAdata` dataset is shown in Figure 4.1c. Notice that the new dataset contains all of the original data in the original order, including two copies of the data for Slovakia. The `rbind` function does not identify duplicates or sort the data. You can do this with the `unique` and `order` functions as explained in Sections 3.10 and 3.13.

You can append three or more datasets in a similar manner.

```
> newdataset<-rbind(dataset1, dataset2, dataset3)
```

## 4.2 Appending columns

The `cbind` function pastes one dataset on to the side of another. This is useful if the data from corresponding rows of each dataset belong to the same observation, as is the case for the `CIAdata1` and `WHodata` datasets shown in Figure 4.2a and 4.2b.

	country	lifeExp	urban	pcGDP		alcohol	mortality
1	Finland	79.41	85	36700	1	12.52	91
2	Slovakia	76.03	55	23600	2	13.33	130
3	UK	80.17	80	36600	3	13.37	77
4	Ukraine	68.74	69	7300	4	15.6	274
5	Spain	81.27	77	31000	5	11.62	68

(a) CIAdata1

	alcohol	mortality
1	12.52	91
2	13.33	130
3	13.37	77
4	15.6	274
5	11.62	68

(b) WHodata

	country	lifeExp	urban	pcGDP	alcohol	mortality
1	Finland	79.41	85	36700	12.52	91
2	Slovakia	76.03	55	23600	13.33	130
3	UK	80.17	80	36600	13.37	77
4	Ukraine	68.74	69	7300	15.6	274
5	Spain	81.27	77	31000	11.62	68

(c) CIAWHodata

**Figure 4.2:** Data from the CIA World Factbook and from the World Health Organisation. See Appendix B for more details.

You can only use the `cbind` function to combine datasets which have the same number of rows. If your datasets have a common variable or variables that can be used to match the observations, use the `merge` function to combine the datasets as explained in Section 4.3.

The command below combines the `CIAdata1` and `WHodata` datasets to create a new dataset called `CIAWHodata`. The result is shown in Figure 4.2c.

```
> CIAWHodata<-cbind(CIAdata1, WHodata)
```

You can combine three or more datasets in a similar way.

```
> newdataset<-cbind(dataset1, dataset2, dataset3)
```

## 4.3 Merging datasets by common variables

The `merge` function allows you to combine two datasets by matching the observations according to the values of common variables.

Consider the `CIAdata1` and `CPIdata` datasets shown in Figure 4.3a and 4.3b. The datasets have a common variable called `country`, which can be used to match corresponding observations.

The following command shows how you would combine the `CIAdata1` and `CPIdata` datasets. The result is shown in Figure 4.3c.

	country	lifeExp	urban	pcGDP
1	Finland	79.41	85	36700
2	Slovakia	76.03	55	23600
3	UK	80.17	80	36600
4	Ukraine	68.74	69	7300
5	Spain	81.27	77	31000

(a) CIAdat1

	country	CPI
1	Spain	80.24
2	UK	100.13
3	Croatia	67.54
4	Italy	94.82
5	Ukraine	51.1
6	Finland	99.69
7	Spain	80.24

(b) CPIdata

	country	lifeExp	urban	pcGDP	CPI
1	Finland	79.41	85	36700	99.69
2	Spain	81.27	77	31000	80.24
3	Spain	81.27	77	31000	80.24
4	UK	80.17	80	36600	100.13
5	Ukraine	68.74	69	7300	51.1

(c) CIACPIdata

	country	lifeExp	urban	pcGDP	CPI
1	Finland	79.41	85	36700	99.69
2	Slovakia	76.03	55	23600	NA
3	Spain	81.27	77	31000	80.24
4	Spain	81.27	77	31000	80.24
5	UK	80.17	80	36600	100.13
6	Ukraine	68.74	69	7300	51.1
7	Croatia	NA	NA	NA	67.54
8	Italy	NA	NA	NA	94.82

(d) allCIACPIdata

**Figure 4.3:** Data from the CIA World Factbook and Numbeo. See Appendix B for more details.

```
> CIACPIdata<-merge(CIAdat1, CPIdata)
```

The `merge` function identifies variables with the same name and uses them to match up the observations. In this example both datasets contain a variable named `country`, so R automatically uses this variable to match the observations. If your datasets have more than one common variable, R matches the observations by the unique combinations of all of the common variables.

If the names of the common variables are not identical in the two datasets, the simplest solution is to rename the variables as explained in Section 3.2. Alternatively you can use the `by.x` and `by.y` arguments to specify which variables to use for matching the observations. For example, to merge two datasets where the common variable is named `var1` in the first dataset and `VAR1` in the second dataset, use the command:

```
> newdataset<-merge(dataset1, dataset2, by.x="var1", by.y="VAR1")
```

When you combine two datasets with the `merge` function, R automatically excludes any unmatched observations that appear in only one of the datasets. In Figure 4.3c, you can see that Slovakia (which appears only in `CIAdatal`) and Italy and Croatia (which appear only in `CPIdata`) are all excluded from the merged dataset.

The `all`, `all.x` and `all.y` arguments allow you to control how R deals with any unmatched observations. To keep all unmatched observations, set the `all` argument to `T`.

```
> allCIACPIdata<-merge(CIAdatal, CPIdata, all=T)
```

The results are shown in Table 4.3d. Notice that Slovakia, Italy and Croatia have now been included and where corresponding data is missing, the missing data symbol `NA` has been substituted. Alternatively, you can use the `all.x` and `all.y` arguments to include unmatched cases from just one of the datasets instead of both. So the command:

```
> allCIAGPIdata<-merge(CIAdatal, CPIdata, all.x=T)
```

includes the data for Slovakia but not Italy and Croatia, while the command:

```
> allCIAGPIdata<-merge(CIAdatal, CPIdata, all.y=T)
```

includes the data for Italy and Croatia but not Slovakia.

When there are multiple matches for an observation, R creates an observation in the final dataset for every possible match. This is known as the *Cartesian product*. Consider the `bigcats` and

	Name	Weight		Name	Status
1	Tiger	203	1	Lion	Vulnerable
2	Leopard	64	2	Leopard	Least Concern
3	Leopard	58	3	Tiger	Endangered
4	Lion	200	4	Leopard	Endangered

(a) `bigcats`                      (b) `endangered`

	Name	Weight	Status
1	Leopard	64	Least Concern
2	Leopard	64	Endangered
3	Leopard	58	Least Concern
4	Leopard	58	Endangered
5	Lion	200	Vulnerable
6	Tiger	203	Endangered

(c) Merged dataset

**Figure 4.4:** Datasets giving information about four big cat species. See Appendix B for more details.

endangered datasets shown in Figure 4.4a and 4.4b. Both datasets have two observations with a name of 'Leopard'. When these two datasets are merged by the `Name` variable, the final dataset (shown in Figure 4.4c) contains four rows for the Leopard data.

To ensure that you get the desired result from a merge, make sure that you are using a common variable or variables that allow R to correctly identify matches. You may also need to remove duplicate observations beforehand, as described in Section 3.10.

R automatically sorts the merged dataset by the same variables that were used to match the observations. If you want to prevent this, set the `sort` argument to `F`.

```
> newdataset<-merge(dataset1, dataset2, sort=F)
```

## 4.4 Stacking data

*Stacking* a dataset means to convert it from *unstacked* form to *stacked* form.

To illustrate the difference between the two forms, consider the `grades1` and `grades2` datasets shown in Figure 4.5. The `grades1` dataset is in unstacked form. It gives the test results of fifteen students, arranged in separate columns according to which class they belong to. The `grades2` dataset gives the same data in stacked form. Here the data is arranged in two columns, the first giving the test result and the second identifying which class the student belongs to.

	ClassA	ClassB	ClassC
1	87	83	80
2	64	97	86
3	89	97	80
4	82	99	80
5	59	92	84

(a) `grades1` (unstacked form)

	values	ind
1	87	ClassA
2	64	ClassA
3	89	ClassA
4	82	ClassA
5	59	ClassA
6	83	ClassB
7	97	ClassB
8	97	ClassB
9	99	ClassB
10	92	ClassB
11	80	ClassC
12	86	ClassC
13	80	ClassC
14	80	ClassC
15	84	ClassC

(b) `grades2` (stacked form)

**Figure 4.5:** The same data in stacked and unstacked form

To convert a dataset from unstacked to stacked form, use the `stack` function.



```
> grades2<-stack(grades1)
```

To stack only some of the columns in your dataset, use the `select` argument. For example, to stack only the `ClassA` and `ClassC` variables, use the command:

```
> newdataset<-stack(grades1, select=c("ClassA", "ClassC"))
```

Note that you can only stack numeric variables.

The `stack` function automatically names the new variables 'values' and 'ind', but you can change the names to something more informative with the `names` function.

```
> names(grades2) <-c("Result", "Class")
```

## 4.5 Unstacking data

*Unstacking* a dataset means to convert it from stacked form to unstacked form. This is the opposite procedure to stacking a dataset.

You can unstack a dataset with the `unstack` function. If your dataset has only two variables, the first of which gives the data values and the second of which identifies which group the observation belongs to (like the `grades2` dataset), use the function as shown.

```
> grades1<-unstack(grades2)
```

If your dataset has more than two variables or the variables are in reverse order then you must specify which variables you want to unstack, as shown below.

```
> unstackeddata<-unstack(stackeddata, values~groups)
```

Here `values` is the names of the variable containing the data values and `groups` is the name of the variable that identifies which group the observation belongs to.

For example, to unstack the `Sepal.Width` column in the `iris` dataset (included with R) to create a new dataset with a column for each iris species, use the command:

```
> sepalwidths<-unstack(iris, Sepal.Width~Species)
```

If you unstack a variable that does not have an equal number of values in each group, R cannot arrange the values to create a new data frame. In this case, R creates a *list* object instead of a data frame. However, you can still access each group of values in the new object by using the dollar notation as shown.

```
> unstackeddata$groupA
```

## 4.6 Reshaping a dataset

Reshaping a dataset is also known as rotating or transforming a dataset. It usually applies to datasets where repeat measurements have been taken, but it is useful in other situations too. Reshaping is the process of changing between long form (with repeat measurements in separate columns) and wide form (with repeat measurements in the same column).

To illustrate the difference between the long and wide forms, consider the dataset shown in Figure 4.6a, which gives cubic resistance measurements for four concrete formulations taken at three, seven and fourteen days after setting. This dataset is in wide form. Figure 4.6b shows the same dataset in long form, with all the measurements in one variable and another variable giving the day that the measurement was taken.

	Formula	Day3	Day7	Day14
1	A	10.8	18.5	41.3
2	C	20.1	29.2	37
3	B	3.7	17.5	28.9
4	D	18.1	27.2	37.6

(a) resistance (wide form)

	row.names	Formula	Day	Resistance
1	A.3	A	3	10.8
2	B.3	B	3	20.1
3	C.3	C	3	3.7
4	D.3	D	3	18.1
5	A.7	A	7	18.5
6	B.7	B	7	29.2
7	C.7	C	7	17.5
8	D.7	D	7	27.2
9	A.14	A	14	41.3
10	B.14	B	14	37
11	C.14	C	14	28.9
12	D.14	D	14	37.6

(b) resistance2 (long form)

**Figure 4.6:** The resistance dataset in long and wide forms. See Appendix B for more details.

The `reshape` functions allows you to convert a dataset from wide to long form. For example, to convert the `resistance` dataset to long form, use the command:

```
> resistance2<-reshape(resistance, direction="long",
  varying=list(c("Day3", "Day7", "Day14")), times=c(3, 7, 14),
  idvar="Formula", v.names="Resistance", timevar="Day")
```

Use the `varying` argument to specify the variables that you want to combine into one column. Use the `times` argument to give the new time values or replicate numbers for each of these variables. Note that the `times` can be character values instead of numeric. Use the `idvar` argument to specify the variable that will group the records together. The `v.names` and `timevar` arguments are optional and give the names for the new variables.

The new dataset is shown in Figure 4.6b.

You can also use the `reshape` function to perform the opposite procedure, i.e. converting a dataset from long to wide form.

Consider the `vitalsigns` dataset shown in Figure 4.7a, which gives measurements of systolic blood pressure, diastolic blood pressure and pulse for four patients. All of the measurements are held in the `result` variable, while the `test` variable identifies the parameter.

	subject	test	result
1	1733	Pulse	120
2	1733	DiaBP	79
3	1733	SysBP	79
4	1734	Pulse	121
5	1734	DiaBP	86
6	1734	SysBP	72
7	1735	Pulse	130
8	1735	DiaBP	94
9	1735	SysBP	74
10	1736	Pulse	142
11	1736	DiaBP	99
12	1736	SysBP	87

(a) `vitalsigns` (long form)

	subject	result.Pulse	result.DiaBP	result.SysBP
1	1733	120	79	79
2	1734	121	86	72
3	1735	130	94	74
4	1736	142	99	87

(b) `vitalsigns2` (wide form)

**Figure 4.7:** The `vitalsigns` dataset in long and wide forms. See Appendix B for more details.

To split the different types of measurements in to separate columns, use the command:

```
> vitalsigns2<-reshape(vitalsigns, direction="wide",
  v.names="result", timevar="test", idvar="subject")
```

Use the `v.names` argument to specify the variable that you want to separate into different columns. Use the `timevar` argument to specify the variable that indicates which column the value belongs

to (i.e. the variable giving the time point, replicate number or category for the record). Use the `idvar` argument to specify which variable is used to group the records together.

Figure 4.7b shows the new wide form dataset. Notice that R automatically generates names for the new variables. Alternatively you can specify the new names with the `varying` argument, as shown below.

```
> vitalsigns2<-reshape(vitalsigns, direction="wide",
  v.names="result", timevar="test", idvar="subject",
  varying=list(c("SysBP", "DiaBP", "Pulse")))
```

### Chapter Summary: Combining and restructuring datasets

Task	Command
Append datasets vertically	<code>rbind(dataset1, dataset2)</code>
Append datasets horizontally	<code>cbind(dataset1, dataset2)</code>
Merge datasets	<code>merge(dataset1, dataset2)</code>
Stack dataset	<code>stack(dataset, select=c("var1", "var2", "var3"))</code>
Unstack dataset	<code>unstack(dataset, values~groups)</code>
Reshape (wide to long)	<code>reshape(dataset, direction="long", varying=list(c("var1", "var2", "var3")), times=c(t1,t2,t3), idvar="identifier")</code>
Reshape (long to wide)	<code>reshape(dataset, direction="wide", v.names="values", timevar="groups", idvar="identifier")</code>

## Chapter 5

# Summary statistics for continuous variables

---

This chapter explains how to calculate basic summary statistics for continuous variables.

Section 5.1 covers *univariate* statistics, meaning statistics that are calculated from a single variable. This includes measures of location such as the mean and median, and measures of dispersion such as the variance, standard deviation and range. Section 5.2 shows how you can calculate these summaries for different groups of observations.

Section 5.3 explains how to calculate measures of the association between two or more continuous variables. This includes the covariance, correlation and Spearman's correlation. It also explains how to perform a hypothesis test to check whether a correlation is statistically significant.

Sections 5.4 and 5.5 show how to compare a sample of continuous values with a theoretical distribution using the Shapiro-Wilk and Kolmogorov-Smirnov tests.

Finally, Section 5.6 shows how to calculate confidence and prediction intervals.

This chapter uses the `trees`, `iris`, `warpbreaks` and `PlantGrowth` datasets, which are included with R, and the `bottles` dataset, which is available from the website. It is helpful to become familiar with them before beginning the chapter. For the datasets included with R, you can view additional information about them by entering `help(datasetname)`. For more information about the `bottles` dataset, see Appendix B.

## 5.1 Univariate statistics

To produce a summary of all the variables in a dataset, use the `summary` function. The function summarises each variable in a manner suitable for its class. For numeric variables it gives the mean, median, range and interquartile range. For factor variables it gives the number in each category. If a variable has any missing values, it will tell you how many.

```
> summary(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
Min. :4.300	Min. :2.000	Min. :1.000	Min. :0.100	setosa :50
1st Qu.:5.100	1st Qu.:2.800	1st Qu.:1.600	1st Qu.:0.300	versicolor:50
Median :5.800	Median :3.000	Median :4.350	Median :1.300	virginica :50
Mean :5.843	Mean :3.057	Mean :3.758	Mean :1.199	
3rd Qu.:6.400	3rd Qu.:3.300	3rd Qu.:5.100	3rd Qu.:1.800	
Max. :7.900	Max. :4.400	Max. :6.900	Max. :2.500	

To calculate a particular statistic for a single variable, use the relevant function from Table 5.1.

Statistic	Function
Mean*	mean
Median*	median
Standard deviation*	sd
Median absolute deviate*	mad
Variance*	var
Maximum value*	max
Minimum value*	min
Interquartile range*	IQR
Range	range
Quantiles	quantile
Tukey five-number summary	fivenum
Sum*	sum
Product*	prod
Number of observations*	length

**Table 5.1:** Functions for summarising continuous variables. Those marked with an asterisk give a single value as output.

For example, to calculate the mean tree height, use the command:

```
> mean(trees$Height)
[1] 76
```

If the variable has any missing data values, set the `na.rm` argument to `T` as shown below. This tells R to ignore any missing values when calculating the statistic. Otherwise the result will be either another missing value or an error message, depending on the function.

```
> mean(dataset$variable, na.rm=T)
```

To calculate a particular statistic for each of the variables in a dataset simultaneously, use the `sapply` function with any of the statistics in Table 5.1.

```
> sapply(trees, mean)
```

---

```
Girth   Height   Volume
13.24839 76.00000 30.17097
```

---

Again, if the dataset has any missing values then set the `na.rm` argument to `T`.

```
> sapply(dataset, mean, na.rm=T)
```

If any of the variables in your dataset are non-numeric, the `sapply` function behaves inconsistently. For example, the command below attempts to calculate the maximum value for each of the variables in the `iris` dataset. R returns an error message because the fifth variable in the dataset is a factor variable.

```
> sapply(iris, max)
Error in Summary.factor(c(1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L,
  1L,  :
  max not meaningful for factors
```

To avoid this problem, exclude any non-numeric variables from the dataset by using bracket notation or the `subset` function, as described in Sections 1.7 and 3.11.

```
> sapply(iris[-5], max)
```

---

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
           7.9           4.4           6.9           2.5
```

---

## 5.2 Statistics by group

You may want to group the values of a numeric variable according to the levels of a factor and calculate a statistic for each group. There are two functions that allow you to do this, called `tapply` and `aggregate`.

You can use the `tapply` function with any of the statistics in Table 5.1. For example, to calculate the mean sepal width for each species for the `iris` dataset:

```
> tapply(iris$Sepal.Width, iris$Species, mean)
```

---

```
setosa versicolor virginica
 3.428    2.770    2.974
```

---

If the numeric variable has any missing data, set the `na.rm` argument to `T`.

```
> tapply(dataset$variable, dataset$factor1, mean, na.rm=T)
```

You can also group the data by more than one factor, by nesting the factor variables inside the `list` function. R calculates the statistic separately for each combination of factor levels. For example, to display the median number of breaks for each combination of tension and wool type for the `warpbreaks` dataset:

```
> tapply(warpbreaks$breaks, list(warpbreaks$wool,
                                warpbreaks$tension), median)
```

---

```
      L   M   H
A 51 21 24
B 29 28 17
```

---

When using more than one grouping variable, you can only use statistical functions that give a single value as output (i.e. those marked with an asterisk in Table 5.1).

Alternatively you can also use the `aggregate` function to summarise variables by groups. Using the `aggregate` function has the advantage that you can summarise several continuous variables simultaneously. It can also be used with statistical functions that give more than one value as output (such as `range` and `quantile`). However the results are displayed a little differently, so it is a matter of personal preference whether to use `tapply` or `aggregate`.

To calculate the mean sepal width for each species, use the `aggregate` function as shown.

```
> aggregate(Sepal.Width~Species, iris, mean)
```

---

```
      Species Sepal.Width
1    setosa      3.428
2 versicolor      2.770
3 virginica      2.974
```

---

Again you can also use more than one grouping variable. For example, to calculate the median number of breaks for each combination of wool and tension for the `warpbreaks` dataset:

```
> aggregate(breaks~wool+tension, warpbreaks, median)
```

---

```
      wool tension breaks
1      A         L     51
2      B         L     29
3      A         M     21
4      B         M     28
5      A         H     24
6      B         H     17
```

---

To summarise two or more continuous variables simultaneously, nest them inside the `cbind` function as shown.

```
> aggregate(cbind(Sepal.Width, Sepal.Length)~Species, iris, mean)
```



---

	Species	Sepal.Width	Sepal.Length
1	setosa	3.428	5.006
2	versicolor	2.770	5.936
3	virginica	2.974	6.588

---

You can save the output to a new data frame, as shown below. This allows you to use the results for further analysis.

```
> sepalmeans<-aggregate(cbind(Sepal.Width, Sepal.Length)~Species,
  iris, mean)
```

Remember to set the `na.rm` argument to `T` if any of the continuous variables have missing values.

## 5.3 Measures of association

### Association

The *association* between two variables is a relationship between them, such that if you know the value of one variable, it tells you something about the value of the other.<sup>50</sup> Positive association means that as the value of one variable increases, the value of the other also tends to increase. Negative association means that as the value of one variable increases, the value of the other tends to decrease. The most commonly used measures of association are:

**Covariance** A measure of the linear association between two continuous variables. Covariance is scale dependent, meaning that the value depends on the units of measurements used for the variables. For this reason, it is difficult to directly interpret the covariance value. The higher the absolute covariance between two variables, the greater the association. Positive values indicate positive association and negative values indicate negative association.<sup>18</sup>

**Pearson's correlation coefficient** (denoted  $r$ ) A scale independent measure of association, meaning that the value is not affected by the unit of measurement. The correlation can take values between -1 and 1, where -1 indicates perfect negative correlation, 0 indicates no correlation and 1 indicates perfect positive correlation. The correlation coefficient only measures *linear* relationships, so it is important to check for non-linear relationships with a scatter plot (see Section 8.7).<sup>6</sup>

**Spearman's rank correlation coefficient** A non-parametric alternative to the Pearson's correlation coefficient, which measures non-linear as well as linear relationships. It also takes values between -1 (perfect negative correlation) and 1 (perfect positive correlation), with a value of 0 indicating no correlation.<sup>27,53</sup> Spearman's correlation can be calculated for ranked as well as continuous data.

### 5.3.1 Covariance

To calculate the covariance between two variables, use the `cov` function.

```
> cov(trees$Height, trees$Volume)
[1] 62.66
```

You can also create a covariance matrix for a whole dataset, which shows the covariance for each pair of variables.

```
> cov(trees)
```

---

	Girth	Height	Volume
Girth	9.847914	10.38333	49.88812
Height	10.383333	40.60000	62.66000
Volume	49.888118	62.66000	270.20280

---

From the output you can see that the covariance between tree girth and tree height is 10.38. The values along the diagonal of the matrix give the variance of the variables. For example, the tree volumes have a variance of 270.2.

If your dataset has any non-numeric variables, remember to exclude them using bracket notation or the `subset` function. Otherwise R will display an error message.

```
> cov(iris[-5])
```

If any of the variables have missing values, set the `use` argument to "pairwise" as shown below. Otherwise the covariance matrix will also have missing values.

```
> cov(dataset, use="pairwise")
```

Another useful option for the `use` argument is "complete". This option completely excludes observations which have missing values for any of the variables, while "pairwise" excludes only those observations which have missing values for either of the variables in a given pair.<sup>31</sup> Enter `help(cov)` to view more details about these options.

### 5.3.2 Pearson's correlation coefficient

To calculate the Pearson's correlation coefficient between two variables, use the `cor` function.

```
> cor(trees$Girth, trees$Volume)
[1] 0.9671194
```

The value is very close to 1, which indicates a very strong positive correlation between tree girth and tree volume. This means that trees with a larger girth tend to have a larger volume.

You can also create a correlation matrix for a whole dataset. Remember to exclude any non-numeric variables using bracket notation or the `subset` function.

```
> cor(trees)
```

---

	Girth	Height	Volume
Girth	1.0000000	0.5192801	0.9671194
Height	0.5192801	1.0000000	0.5982497
Volume	0.9671194	0.5982497	1.0000000

---

Notice that the values along the diagonal of the matrix are all equal to 1, because a variable always correlates perfectly with itself.

Again if any of the variables have missing values, set the `use` argument to `"pairwise"`.

```
> cor(dataset, use="pairwise")
```

### 5.3.3 Spearman's rank correlation coefficient

The `cor` function can also calculate the Spearman's rank correlation coefficient between two variables. Set the `method` argument to `"spearman"`, as shown below.

```
> cor(trees$Girth, trees$Volume, method="spearman")
[1] 0.9547151
```

You can also create a correlation matrix for a whole dataset. Remember to exclude any non-numeric variables using bracket notation.

```
> cor(trees, method="spearman")
```

---

	Girth	Height	Volume
Girth	1.0000000	0.4408387	0.9547151
Height	0.4408387	1.0000000	0.5787101
Volume	0.9547151	0.5787101	1.0000000

---

If any of the variables have missing values, set the `use` argument to `"pairwise"` as shown below.

```
> cor(dataset$var1, dataset$var2, method="spearman", use="pairwise")
```

### 5.3.4 Hypothesis test of correlation

#### Hypothesis test of correlation

A hypothesis test of correlation determines whether a correlation is statistically significant. The null hypothesis for the test is that the population correlation is equal to zero, meaning that there is no correlation between the variables. The alternative hypothesis is that the population correlation is not equal to zero, meaning that there is some correlation between the variables. You can also perform a one-sided test, where the alternative hypothesis is either that the population correlation is greater than zero (the variables are positively correlated) or that the population correlation is less than zero (the variables are negatively correlated).<sup>51</sup>

See Chapter 10 for more details about hypothesis testing.

You can perform a test of the correlation between two variables with the `cor.test` function.

```
> cor.test(dataset$var1, dataset$var2)
```

By default R performs a test of the Pearson's correlation. If you would prefer to test the Spearman's correlation, set the `method` argument to "spearman" as shown.

```
> cor.test(dataset$var1, dataset$var2, method="spearman")
```

By default R performs a two-sided test, but you can adjust this by setting the `alternative` argument to "less" or "greater" as required.

```
> cor.test(dataset$var1, dataset$var2, alternative="greater")
```

The output includes a 95% confidence interval for the correlation estimate. To adjust the size of this interval, use the `conf.level` argument.

```
> cor.test(dataset$var1, dataset$var2, conf.level=0.99)
```

#### **Example 5.1.** Hypothesis test of correlation using the `trees` dataset

Suppose you want to perform a hypothesis test to help determine whether the correlation between tree girth and tree volume is statistically significant.

To perform a two-sided test of the Pearson's product moment correlation between tree girth and volume at the 5% significance level, use the command:

```
> cor.test(trees$Girth, trees$Volume)
```

The output is shown below.

---

```
Pearson's product-moment correlation

data:  trees$Girth and trees$Volume
t = 20.4783, df = 29, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9322519 0.9841887
sample estimates:
      cor
0.9671194
```

---

The correlation is estimated at 0.967, with a 95% confidence interval of 0.932 to 0.984. This means that as tree girth increases, tree volume tends to increase also.

Since the p-value of 2.2e-16 is much less than the significance level of 0.05, we can reject the null hypothesis that there is no correlation between girth and volume, in favour of the alternative hypothesis that the two are correlated.

## 5.4 Shapiro-Wilk test

### Shapiro-Wilk test

The Shapiro-Wilk test is a hypothesis test which can help to determine whether a sample has been drawn from a normal distribution. The null hypothesis for the test is that the sample is drawn from a normal distribution and the alternative hypothesis is that it is not.<sup>25, 42</sup>

You can perform a Shapiro-Wilk test with the `shapiro.test` function.

```
> shapiro.test(dataset$variable)
```

### Example 5.2. Shapiro-Wilk test using the `trees` dataset

Suppose you want to perform a Shapiro-Wilk test to help determine whether the tree heights follow a normal distribution. You will use a 5% significance level. To perform the test, use the command:

```
> shapiro.test(trees$Height)
```

---

```
Shapiro-Wilk normality test

data:  trees$Height
W = 0.9655, p-value = 0.4034
```

---

From the output we can see that the p-value for the test is 0.4034. Since this is not less than our significance level of 0.05, we cannot reject the null hypothesis. This means there is no evidence that the tree heights do not follow a normal distribution.

## 5.5 Kolmogorov-Smirnov test

### Kolmogorov-Smirnov test

A one-sample Kolmogorov-Smirnov test helps to determine whether a sample is drawn from a particular theoretical distribution. It has the null hypothesis that the sample is drawn from the distribution and the alternative hypothesis that it is not.

A two-sample Kolmogorov-Smirnov test helps to determine whether two samples are drawn from the same distribution. It has the null hypothesis that they are drawn from the same distribution and the alternative hypothesis that they are not.<sup>43</sup>

You can perform a Kolmogorov-Smirnov test with the `ks.test` function. To perform a one-sample test with the null hypothesis that the sample is drawn from a normal distribution with a mean of 100 and a standard deviation of 10, use the command:

```
> ks.test(dataset$variable, "pnorm", 100, 10)
```

To test a sample against another theoretical distribution, replace `"pnorm"` with the relevant cumulative distribution function. A list of functions for standard probability distributions is given in Table 7.1 on page 84. You must also substitute the mean and standard deviation with any parameters relevant to the distribution. Use the `help` function to check the parameters for the distribution of interest.

To perform a two-sample test to determine whether two samples are drawn from the same distribution, use the command:

```
> ks.test(dataset$sample1, dataset$sample2)
```

If your data is in stacked form (with the values for both samples in one variable), you must first unstack the dataset as explained in Section 4.5.

### Example 5.3. One-sample Kolmogorov-Smirnov test using `bottles` data

Consider the `bottles` dataset, which is available from the website. The dataset gives data for a sample of 20 bottles of soft drink taken from a filling line. The dataset contains one variable named `Volume`, which gives the volume of liquid in millilitres for each of the bottles.

The bottle filling volume is believed to follow a normal distribution with a mean of 500 ml and a standard deviation of 25 ml. Suppose you wish to use a one-sample Kolmogorov-Smirnov test to

determine whether the data is consistent with this theory. The test has the null hypothesis that the bottles volumes are drawn from the described distribution, and the alternative hypothesis that they are not. A significance level of 0.05 will be used for the test.

To perform the test, use the command:

```
> ks.test(bottles$Volume, "pnorm", 500, 25)
```

This gives the following output:

---

```
One-sample Kolmogorov-Smirnov test

data:  bottles$Volume
D = 0.2288, p-value = 0.2108
alternative hypothesis: two-sided
```

---

From the output we can see that the p-value for the test is 0.2108. As this is not less than the significance level of 0.05, we cannot reject the null hypothesis. This means that there is no evidence that the bottle volumes are not drawn from the described normal distribution.

**Example 5.4.** Kolmogorov-Smirnov two-sample test using the `PlantGrowth` data

Consider the `PlantGrowth` dataset (included with R), which gives the dried weight of thirty batches of plants, each of which received one of three different treatment. The `weight` variable gives the weight of the batch and the `groups` variable gives the treatment received (`ctrl`, `trt1` or `trt2`).

Suppose you want to use a two-sample Kolmogorov-Smirnov test to determine whether batch weight has the same distribution for the treatment groups `trt1` and `trt2`.

First you need to unstack the data with the `unstack` function, as described in Section 4.5.

```
> PlantGrowth2<-unstack(PlantGrowth)
```

Once the data is in unstacked form, perform the test as shown.

```
> ks.test(PlantGrowth2$trt1, PlantGrowth2$trt2)
```

This gives the following output:

---

```
Two-sample Kolmogorov-Smirnov test

data:  PlantGrowth2$trt1 and PlantGrowth2$trt2
D = 0.8, p-value = 0.002057
alternative hypothesis: two-sided
```

---

The p-value of 0.002057 is less than the significance level of 0.05. This means that there is evidence to reject the null hypothesis that the batch weight distribution is the same for both treatment groups, in favour of the alternative hypothesis that the batch weight distribution is different for the two treatment groups.

## 5.6 Confidence intervals and prediction intervals

### Confidence and prediction intervals

A confidence interval for the population mean gives an indication of how accurately the sample mean estimates the population mean. A 95% confidence interval is defined as an interval calculated in such a way that if a large number of samples were drawn from a population and the interval calculated for each of these samples, 95% of the intervals will contain the true population mean value.<sup>40</sup>

A prediction interval gives an indication of how accurately the sample mean predicts the value of a further observation drawn from the population.<sup>41</sup>

The simplest way to obtain a confidence interval for a sample mean is with the `t.test` function, which provides one with the output. Section 10.1 discusses the function in more detail. If you are only interested in obtaining a confidence interval, use the command:

```
> t.test(trees$Height)
```

---

```
One Sample t-test

data:  trees$Height
t = 66.4097, df = 30, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 73.6628 78.3372
sample estimates:
mean of x
      76
```

---

From the results you can see that the mean tree height is 76 ft, with a 95% confidence interval of 73.7 to 78.3 ft.

By default R calculates a 95% interval. For a different size confidence interval such as 99%, adjust the `conf.level` argument as shown.

```
> t.test(trees$Height, conf.level=0.99)
```

You can also calculate one-sided confidence intervals. For an upper confidence interval, set the `alternative` argument to "less". For a lower confidence interval, set it to "greater".

```
> t.test(dataset$variable, alternative="greater")
```

To calculate a prediction interval for the tree heights, use the command:



```
> predict(lm(trees$Height~1), interval="prediction")[1,]
```

---

```
      fit      lwr      upr  
76.0000 62.7788 89.2212
```

```
Warning message:
```

```
In predict.lm(lm(trees$Height ~ 1), interval = "prediction") :  
  Predictions on current data refer to _future_ responses
```

---

From the output you can see the the prediction interval for the tree heights is 62.8 to 89.2 ft.

Again you can adjust the confidence level with the `level` argument as shown.

```
> predict(lm(dataset$variable~1), interval="prediction",  
          level=0.99)[1,]
```

The commands for creating prediction intervals use the `lm` and `predict` functions. You will learn more about these functions in Chapter 11, which will help you to understand what these complex commands are doing. For now, you can just use them by substituting the appropriate dataset and variable names.

**Chapter summary: Summary statistics for continuous variables**

Task	Command
Statistic for each variable	<code>sapply(dataset, statistic)</code>
Statistic by group	<code>tapply(dataset\$var1, dataset\$factor1, statistic)</code>
Statistic by group	<code>aggregate(variable~factor, dataset, statistic)</code>
Covariance	<code>cov(dataset\$var1, dataset\$var2)</code>
Covariance matrix	<code>cov(dataset)</code>
Pearson's correlation coefficient	<code>cor(dataset\$var1, dataset\$var2)</code>
Correlation matrix	<code>cor(dataset)</code>
Spearman's rank correlation coefficient	<code>cor(dataset\$var1, dataset\$var2, method="spearman")</code>
Spearman's correlation matrix	<code>cor(dataset, method="spearman")</code>
Hypothesis test of correlation	<code>cor.test(dataset\$var1, dataset\$var2)</code>
Shapiro-Wilk test	<code>shapiro.test(dataset\$variable)</code>
One-sample Kolmogorov-Smirnov test	<code>ks.test(dataset\$sample1, "pnorm", mean, sd)</code>
Two-sample Kolmogorov-Smirnov test	<code>ks.test(dataset\$sample1, dataset\$sample2)</code>
Confidence interval	<code>t.test(dataset\$variable)</code>
Prediction interval	<code>predict(lm(dataset\$variable ~ 1), interval="prediction")[1,]</code>

# Chapter 6

## Tabular data

---

This chapter explains how to summarise and calculate statistics for categorical variables.

Section 6.1 explains how to create frequency and contingency tables to summarise categorical data, and how to store the results as a table object.

Section 6.2 explains how to perform a chi-square goodness-of-fit test to compare categorical data with a hypothesised distribution.

Section 6.3 explains how to perform a chi-square test of association to identify a relationship between two or more categorical variables. Section 6.4 explains how to perform a Fisher's exact test of association for a two-by-two contingency table.

This chapter uses the `warpbreaks` and `esoph` datasets and the `Titanic` table object, which are included with R. You can view more information about them by entering `help(datasetname)`. It also uses the `people2` dataset (which is a modified version of the `people` dataset introduced in Chapter 3), and the `apartments` dataset. These are both available from the website and described in Appendix B.

### 6.1 Frequency tables

Frequency tables summarise a categorical variable by displaying the number of observations belonging to each category. Frequency tables for two or more categorical variables (known as contingency tables or cross tabs) summarise the relationship between two or more categorical variables by displaying the number of observations that fall into each combination of categories.

In R, a table is also a type of object that holds tabulated data. There are some example table objects included with R, such as `HairEyeColor` and `Titanic`.

To create a one-dimensional frequency table showing the number of observations for each level of a factor variable, use the `table` function as shown.

```
> table(people2$Eye.Colour)
```

---

```
Blue Brown Green
  7      6      3
```

---

Whilst R allows you to create tables from any type of variable, they are only really meaningful for factor variables with a relatively small number of values. If you want to include a continuous variable, first divide it into categories with the `cut` function, as explained in Section 3.5.

To add an additional column to the table showing the numbers of missing values (if there are any), set the `useNA` argument as shown.

```
> table(dataset$factor1, useNA="ifany")
```

To create a two-dimensional contingency table, give two variables as input.

```
> table(people2$Eye.Colour, people2$Sex)
```

---

	Female	Male
Blue	3	4
Brown	1	5
Green	2	1

---

Similarly you can create contingency tables for three or more variables.

```
> table(people2$Eye.Colour, people2$Height.Cat, people2$Sex)
```

---

```
, , = Female
```

	Medium	Short	Tall
Blue	3	0	0
Brown	0	0	0
Green	0	2	0

```
, , = Male
```

	Medium	Short	Tall
Blue	2	0	2
Brown	4	0	1
Green	0	0	1

---

To save a table as a table object, assign the output of the `table` function to a new object name as shown.

```
> sexeyetable<-table(people2$Eye.Colour, people2$Sex)
```

Once you have created a table object, you can use it with functions such as `pie` or `chisq.test` to create output that is relevant to tabular data. You will learn about some of these functions later in this chapter.

There are also a few functions which allow you to present table objects in different ways. These include `fTable`, `prop.table` and `addmargins`.

The `ftable` function displays your table in a more compact way, which is useful for tables with three or more dimensions.

```
> ftable(Titanic)
```

---

Class	Sex	Age	Survived	No	Yes
1st	Male	Child		0	5
		Adult		118	57
	Female	Child		0	1
		Adult		4	140
2nd	Male	Child		0	11
		Adult		154	14
	Female	Child		0	13
		Adult		13	80
3rd	Male	Child		35	13
		Adult		387	75
	Female	Child		17	14
		Adult		89	76
Crew	Male	Child		0	0
		Adult		670	192
	Female	Child		0	0
		Adult		3	20

---

The `prop.table` function displays the table with each cell count expressed as a proportion of the total count.

```
> prop.table(sexeyetable)
```

---

	Female	Male
Blue	0.1875	0.2500
Brown	0.0625	0.3125
Green	0.1250	0.0625

---

To display the cell counts expressed as a proportion of the row or column totals instead of the grand total, set the `margin` argument to 1 for rows, 2 for columns and 3+ for higher dimensions.

```
> prop.table(sexeyetable, margin=2)
```

---

	Female	Male
Blue	0.5000000	0.4000000
Brown	0.1666667	0.5000000
Green	0.3333333	0.1000000

---

To display percentages, multiply the whole table by 100. You can also use the `round` function to round all of the numbers in the table.

```
> round(prop.table(sexeyetable)*100)
```

---

	Female	Male
Blue	19	25
Brown	6	31
Green	12	6

---

The `addmargins` function displays your table with row and column totals.

```
> addmargins(sexeyetable)
```

---

	Female	Male	Sum
Blue	3	4	7
Brown	1	5	6
Green	2	1	3
Sum	6	10	16

---

To create a table from a data frame containing count data, such as the `warpbreaks` dataset, use the `xtabs` function as shown.

```
> xtabs(breaks~wool+tension, warpbreaks)
```

---

	tension		
wool	L	M	H
A	401	216	221
B	254	259	169

---

If the data frame has more than one column of count data, put them inside the `list` function as shown.

```
> xtabs(list(counts1, counts2)~factor1, dataset)
```

You can create a table object by assigning the output of the `xtabs` function to a new object name.

```
> warpbreakstable<-xtabs(breaks~wool+tension, warpbreaks)
```

To create a data frame of count data from a table object, use the `as.data.frame` function.

```
> as.data.frame(Titanic)
```

---

	Class	Sex	Age	Survived	Freq
1	1st	Male	Child	No	0
2	2nd	Male	Child	No	0
3	3rd	Male	Child	No	35
4	Crew	Male	Child	No	0
5	1st	Female	Child	No	0
6	2nd	Female	Child	No	0
7	3rd	Female	Child	No	17
8	Crew	Female	Child	No	0
9	1st	Male	Adult	No	118

---

10	2nd	Male	Adult	No	154
11	3rd	Male	Adult	No	387
12	Crew	Male	Adult	No	670
13	1st	Female	Adult	No	4
14	2nd	Female	Adult	No	13
15	3rd	Female	Adult	No	89
16	Crew	Female	Adult	No	3
17	1st	Male	Child	Yes	5
18	2nd	Male	Child	Yes	11
19	3rd	Male	Child	Yes	13
20	Crew	Male	Child	Yes	0
21	1st	Female	Child	Yes	1
22	2nd	Female	Child	Yes	13
23	3rd	Female	Child	Yes	14
24	Crew	Female	Child	Yes	0
25	1st	Male	Adult	Yes	57
26	2nd	Male	Adult	Yes	14
27	3rd	Male	Adult	Yes	75
28	Crew	Male	Adult	Yes	192
29	1st	Female	Adult	Yes	140
30	2nd	Female	Adult	Yes	80
31	3rd	Female	Adult	Yes	76
32	Crew	Female	Adult	Yes	20

## 6.2 Chi-square goodness-of-fit test

### Chi-square goodness-of-fit test

The chi-square goodness-of-fit test (also known as the Pearson's chi-squared test or  $\chi^2$  test) allows you to compare categorical data with a theoretical distribution. It has the null hypothesis that the data follows the specified distribution, and the alternative hypothesis that it does not. The test is only suitable if sufficient data is available, which is commonly defined as each category having an expected frequency (under the null hypothesis) of at least five.<sup>7</sup>

The test should not be confused with the chi-square test of association (see Section 6.3), which helps to determine whether two or more categorical variables are associated.

For more details about hypothesis testing, see Chapter 10.

You can perform a chi-square goodness-of-fit test with the `chisq.test` function. If you have a one-dimensional table object, you can test it against the uniform distribution (i.e. against the null hypothesis that all categories are equally likely to occur) as shown.

```
> chisq.test(tableobject)
```

If you are using raw data, nest the `table` function inside the `chisq.test` function as shown.

```
> chisq.test(table(dataset$factor1))
```

To test the data against a different theoretical distribution, use the `p` argument to give a list of expected relative frequencies under the null hypothesis. You must give the same number of relative frequencies as there are level in your table, and they must sum to one.

For example, to test the hypothesis that 10% of the population belong to the first category, 40% to the second category, 40% to the third category and 10% to the fourth category, use the command:

```
> chisq.test(tableobject, p=c(0.1, 0.4, 0.4, 0.1))
```

### Example 6.1. Chi-square goodness-of-fit test using the `apartments` data

Consider the `apartments` dataset (available from the website and described in Appendix B), which give details of 39 one-bedroom apartments advertised for rent in a particular area of the UK in October 2012. The `Price.Cat` variable gives the rental price category for the apartment.

To create a table showing the number of apartments in each price category, use the command:

```
> table(apartments$Price.Cat)
```

---

£500-550	£551-600	£601-650	£651+
7	14	11	7

---

It is believed that 20% of the apartments in this area have a rental price less than £550, 30% have a price between £551 and £600, 30% a price between £601 and £650 and 20% have a rental price greater than £650.

Suppose that you want to use a chi-square goodness-of-fit test to determine whether the data is consistent with the hypothesised price distribution. The test has the null hypothesis that the described price distribution is correct, and the alternative hypothesis that it is not. A significance level of 0.05 is used.

To perform the test, use the command:

```
> chisq.test(table(apartments$Price.Cat), p=c(0.2, 0.3, 0.3, 0.2))
```

This gives the output:

---

```
Chi-squared test for given probabilities

data:  table(apartments$Price.Cat)
X-squared = 0.6581, df = 3, p-value = 0.883
```

---

The p-value of 0.883 is not less than the significance level of 0.05, so we cannot reject the null hypothesis. This mean that the data is consistent with the hypothesised price distribution.



## 6.3 Chi-square test of association

### Chi-square test of association

The chi-square test of association helps to determine whether two or more categorical variables are associated. The test has the null hypothesis that the variables are independent and the alternative hypothesis that they are not independent (i.e at least two of the variables are associated).<sup>22</sup> The test is only suitable if there is sufficient data, which is commonly defined as all table cells having expected counts (under the null hypothesis) of at least five.<sup>7</sup> An alternative test for two-by-two tables is Fisher's exact test, described in Section 6.4.

The `summary` function performs a chi-square test of association when given a table object as input. If you have already created a table object, use the `summary` function directly.

```
> summary(tableobject)
```

If you have raw data, nest the `table` function inside the `summary` function as shown below.

```
> summary(table(dataset$var1, dataset$var2, dataset$var3))
```

### Example 6.2. Chi-square test of association using `people2` data

Suppose you want to use a chi-square test of association to determine whether sex and eye colour are associated, using the `people2` dataset.

If you still have the `sexeyetable` object created in Section 6.1, use the command:

```
> summary(sexeyetable)
```

Alternatively you can perform the test using the raw data, as shown below.

```
> summary(table(people2$Sex, people2$Eye.Colour))
```

---

```
Number of cases in table: 16
Number of factors: 2
Test for independence of all factors:
  Chisq = 2.2857, df = 2, p-value = 0.3189
  Chi-squared approximation may be incorrect
```

---

As the p-value of 0.3189 is not less than the significance level of 0.05, we cannot reject the null hypothesis that sex and eye colour are independent. This means that there is no evidence of an association between the two.

The warning `Chi-squared approximation may be incorrect` tells us that as some cells have expected counts less than five. This means that the results may be unreliable and should be

interpreted with caution. A discussion of Chi-square tests with small expected counts can be found on page 39 of *The Analysis of Contingency Tables*, by B.S. Everitt.<sup>3</sup>

**Example 6.3.** Chi-square test of association using the `esoph` data

Consider the `esoph` dataset, which is included with R. The dataset gives the results of a case-control study of oesophageal cancer. You can view more details by entering `help(esoph)`. The `agegp`, `alcgp` and `tobgp` variables list categories for the subjects' age, alcohol consumption and smoking habits. The variables `ncases` and `ncontrols` give the number of subjects with and without oesophageal cancer that fall into each of these categories.

Suppose you want to use a chi-square test of association to determine where there is any association between smoking habits and oesophageal cancer. The test has the null hypothesis that smoking habits and oesophageal cancer are independent, and the alternative hypothesis that they are associated. A significance level of 0.05 will be used.

As the dataset contains data which is already expressed as counts, you can use the `xtabs` function to create a table giving the number of subjects with and without oesophageal cancer that fall into each category of smoking habits.

```
> tobacco<-xtabs(cbind(ncases, ncontrols)~tobgp, esoph)
> tobacco
```

---

tobgp	ncases	ncontrols
0-9g/day	78	525
10-19	58	236
20-29	33	132
30+	31	82

---

To perform the test, use the command:

```
> summary(tobacco)
```

---

```
Call: xtabs(formula = cbind(ncases, ncontrols) ~ tobgp, data = esoph)
Number of cases in table: 1175
Number of factors: 2
Test for independence of all factors:
    Chisq = 18.363, df = 3, p-value = 0.0003702
```

---

As the p-value of 0.0003702 is less than the significance level of 0.05, we can reject the null hypothesis that smoking category and oesophageal cancer are independent, in favour of the alternative hypothesis that the two are associated. This means that there is evidence of a relationship between smoking habits and oesophageal cancer.

## 6.4 Fisher's exact test

### Fisher's exact test

The Fisher's exact test is used to test for association between two categorical variables which each have two levels. Unlike the chi-square test of association, it can be used even when very little data is available. The test has the null hypothesis that the two variables are independent and the alternative hypothesis that they are not independent.<sup>5,10</sup>

You can perform a Fisher's exact test with the `fisher.test` function. You can use the function with a two-by-two table object as shown.

```
> fisher.test(tableobject)
```

You can also use raw data as shown below.

```
> fisher.test(dataset$var1, dataset$var2)
```

The test results are accompanied by a 95% confidence interval for the odds ratio. You can change the size of the interval with the `conf.level` argument.

```
> fisher.test(dataset$var1, dataset$var2, conf.level=0.99)
```

### Example 6.4. Fisher's exact test using `people2` data

Using the `table` function, we can see there are more left-handed women than left-handed men in the `people2` dataset.

```
> table(people2$Sex, people2$Handedness)
```

---

	Left	Right
Female	2	2
Male	0	9

---

Suppose you want to perform a Fisher's exact test to determine whether there is any statistically significant relationship between sex and handedness. The test has the null hypothesis that sex and handedness are independent, and the alternative hypothesis that they are associated. A significance level of 0.05 is used.

To perform the test, use the command:

```
> fisher.test(people2$Sex, people2$Handedness)
```

Which gives the output:

## Fisher's Exact Test for Count Data

```

data:  people2$Sex and people2$Handedness
p-value = 0.07692
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.4766247      Inf
sample estimates:
odds ratio
      Inf

```

Since the p-value of 0.07692 is not less than the significance level of 0.05, we cannot reject the null hypothesis that sex and handedness are independent. This means that there is no evidence of a relationship between sex and handedness.

### Chapter Summary: Tabular data

Task	Command
Contingency table	<code>table(dataset\$factor1, dataset\$factor2)</code>
Compact table	<code>ftable(tableobject)</code>
Proportions table	<code>prop.table(tableobject)</code>
Table with margins	<code>addmargins(tableobject)</code>
Chi-square goodness-of-fit test	<code>chisq.test(tableobject, p=c(p1, p2, pN))</code> <code>chisq.test(table(dataset\$factor1), p=c(p1, p2, pN))</code>
Chi-square test of association	<code>summary(tableobject)</code> <code>summary(table(dataset\$factor1, dataset\$factor2))</code>
Fisher's exact test	<code>fisher.test(tableobject)</code> <code>fisher.test(dataset\$factor1, dataset\$factor2)</code>

## Chapter 7

# Probability distributions

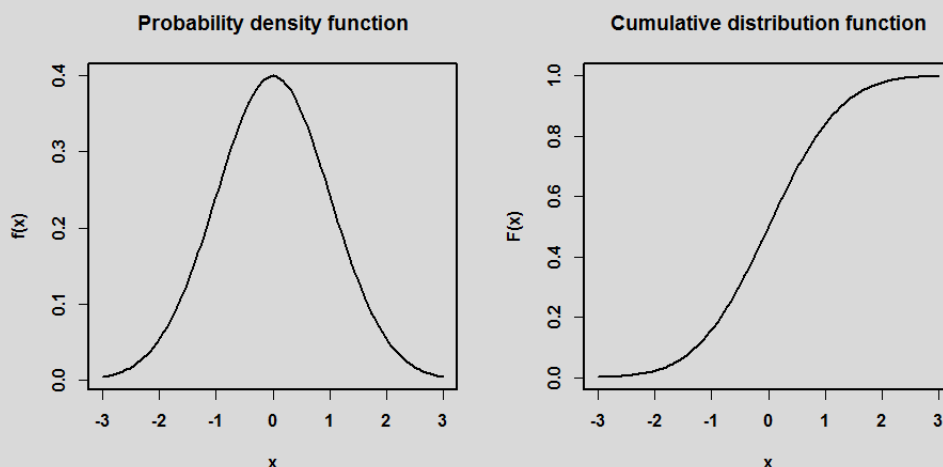
### The probability density function and cumulative distribution function

The probability density function (pdf) and cumulative distribution function (cdf) are two ways of specifying the probability distribution of a random variable.

The pdf is denoted  $f(x)$  and gives the relative likelihood that the value of the random variable will be equal to  $x$ . The total area under the curve is equal to one.<sup>48</sup>

The cdf is denoted  $F(x)$  and gives the probability that the value of a random variable will be less than or equal to  $x$ . The value of the cdf at  $x$  is equal to the area under the curve of the pdf between  $-\infty$  and  $x$ . The cdf takes values between zero and one.<sup>54</sup>

The plots below show the pdf and cdf for the standard normal distribution.



R has functions for all of the well-known standard probability distributions. These allow you to use R in place of a set of statistical tables. For each distribution, there are the following four functions:

**Probability density function or probability mass function** (prefix `d`)

For discrete distributions, you can use the probability mass function (pmf) to answer questions of the type *'What is the probability that the outcome will be equal to  $x$ ?'*. The probability density function and probability mass function are covered in Section 7.1.

**Cumulative distribution function** (prefix `p`)

You can use the cumulative density function (cdf) to answer questions of the type *'If we were to randomly select a member of a given population, what is the probability that it will have a value less than  $x$ , or a value between  $x$  and  $y$ ?'*. The cumulative distribution function is covered in Section 7.2.

**Inverse cumulative distribution function** (prefix `q`)

Use the inverse cdf to answer questions such as *'Which value do  $x\%$  of the population fall below?'*, or *'What range of values do  $x\%$  of the population fall within?'*. The inverse cumulative distribution function is covered in Section 7.3.

**Random number generator** (prefix `r`)

Use the random number generator to simulate a random sample from a given distribution. Functions for generating random numbers are covered in Section 7.4.

Distribution	Function suffix
Beta	beta
Binomial	binom
Cauchy	cauchy
Chi-square	chisq
Exponential	exp
F	f
Gamma	gamma
Geometric	geom
Hypergeometric	hyper
Logistic	logis
Log normal	lnorm
Multinomial	multinom
Negative binomial	nbinom
Normal	norm
Poisson	pois
Student's t distribution	t
Uniform	unif
Weibull	weibull

**Table 7.1:** Standard probability distributions included with R.<sup>33</sup> Combine the distribution suffix with prefix: `d` for the pdf; `p` for the cdf; `q` for the inverse cdf; and `r` for the random number generator.

The function names are formed from a prefix (either `d`, `p`, `q` or `r`) and a suffix for the relevant distribution, taken from Table 7.1. For example, the cumulative distribution function for the normal distribution is called `pnorm`, and the random number generator for the exponential distribution is called `rexp`.

## 7.1 Probability density functions and probability mass functions

For discrete distributions, use the pmf to find the probability that an outcome or observation is equal to  $x$ . The name of the function is formed by combining the prefix `d` with the relevant suffix from Table 7.1 on page 7.1.

When using the pmf, the first argument that the function requires is always the possible outcome of interest. This is followed by any parameters relevant to the distribution. You can check these with the `help` function. The examples below demonstrate how to use the pmfs of the binomial and Poisson distributions.

For continuous distributions, the pdf gives the relative likelihood of  $x$ , which has no direct interpretation. However you can still find the value of the pdf in the same way as for discrete distributions.

In Section 8.10 you will learn how you can use the `curve` function to plot the pdf of a statistical distribution.

### Example 7.1. Probability mass function for the Binomial distribution

Suppose that a fair die is rolled 10 times. What is the probability of throwing exactly 2 sixes?

You can answer the question using the `dbinom` function, as shown.

```
> dbinom(2, 10, 1/6)
[1] 0.29071
```

The probability of throwing 2 sixes is approximately 0.29 or 29%.

### Example 7.2. Probability mass function for the Poisson distribution

The number of lobster ordered in a restaurant on a given day is known to follow a Poisson distribution with a mean of 20. What is the probability that exactly 18 lobsters will be ordered tomorrow?

You can answer the question with the `dpois` function as shown.

```
> dpois(18, 20)
[1] 0.08439355
```

The probability that exactly 18 lobsters are ordered is 8.4%.

**Example 7.3.** Probability density function for the normal distribution

To find the value of the pdf at  $x = 2.5$  for a normal distribution with a mean of 5 and a standard deviation of 2, use the command:

```
> dnorm(2.5, mean=5, sd=2)
[1] 0.09132454
```

The value of the pdf at  $x = 2.5$  is 0.091.

## 7.2 Finding probabilities

To find the probability that a randomly selected member of a population will have a value less than or equal to  $x$ , use the cdf for the appropriate distribution. The name of the function is formed by combining the prefix `p` with the relevant suffix from Table 7.1 on page 7.1.

For the normal distribution, use the `pnorm` function. For a standard normal distribution (with a mean of zero and a standard deviation of one) the function does not require any additional arguments. For example to find the probability that a randomly selected value will be less than or equal to 2.5, use the command:

```
> pnorm(2.5)
[1] 0.9937903
```

From the output, you can see that the probability is 0.9937903, or approximately 99%.

To find a probability for a non-standard normal distribution, add the `mean` and `sd` arguments. For example, if a random variable is known to be normally distributed with a mean of 5 and a standard deviation of 2 and you wish to find the probability that a randomly selected member will be no more than 6, use the command:

```
> pnorm(6, mean=5, sd=2)
[1] 0.6914625
```

To find the complementary probability that the value will be *greater* than 6, set the `lower.tail` argument to `F`.

```
> pnorm(6, 5, 2, lower.tail=F)
[1] 0.3085375
```

You can find probabilities for other distributions by substituting `norm` for the relevant distribution suffix from Table 7.1. You will also need to change the `mean` and `sd` arguments for the parameters relevant to the distribution. Use the `help` function to check the parameters for the function that you are using.



**Example 7.4.** Finding probabilities for the normal distribution

Suppose the height of men in the UK is known to be normally distributed with a mean of 177 cm and a standard deviation of 10 cm. If you were to select a man from the UK population at random, what is the probability that he would be more than 200 cm tall?

To answer the question, use the command:

```
> pnorm(200, 177, 10, lower.tail=F)
[1] 0.01072411
```

From the output you can see that the probability is approximately 0.011, or 1.1%.

What is the probability that he would be less than 150 cm tall?

```
> pnorm(150, 177, 10)
[1] 0.003466974
```

The probability is 0.0035, or 0.35%.

**Example 7.5.** Finding probabilities for the binomial distribution

If you were to role a fair six-sided die 100 times, what is the probability of rolling a six no more than 10 times?

The number of sixes in 100 dice roles follows a binomial distribution, so you can answer the question with the `pbinom` function as shown.

```
> pbinom(10, 100, 1/6)
[1] 0.04269568
```

From the output you can see that the probability of rolling no more than 10 sixes is 0.043 (4.3%).

What is the probability of rolling a six more than 20 times?

```
> pbinom(20, 100, 1/6, lower.tail=F)
[1] 0.1518878
```

The probability of rolling more than 20 sixes is approximately 0.15, or 15%.

**Example 7.6.** Finding probabilities for the exponential distribution

Malfunctions in a particular type of electronic device are known to follow an exponential distribution with a mean time of 24 months until the device malfunctions. What is the probability that a randomly selected device will malfunction within the first six months?

You can answer the question using the `pexp` function, as shown below:

```
> pexp(6, 1/24)
[1] 0.2211992
```

The probability of malfunction within six months is 0.22 (22%).

What is the probability that a randomly selected device will last more than 5 years (60 months) without malfunction?

```
> pexp(60, 1/24, lower.tail=F)
[1] 0.082085
```

The probability that it will last more than 5 years is approximately 0.08, or 8%.

## 7.3 Finding quantiles

To find the value that a given percentage of a population falls above or below, or the range of values within which a given percentage of a population lies, use the inverse cdf for the appropriate distribution. The name of the function is formed by combining the prefix `q` with the relevant suffix from Table 7.1 on page 7.1.

To find quantiles for the normal distribution, use the `qnorm` function. For a standard normal distribution, use the function without any additional arguments. For example to find the value below which 95% of values fall, use the command:

```
> qnorm(0.95)
[1] 1.644854
```

For non-standard normal distributions, use the `mean` and `sd` arguments to specify the parameters for the distribution. For example, suppose that a variable is known to be normally distributed with a mean of 5 and standard deviation of 2. To find the value below which 95% of the population falls, use the command:

```
> qnorm(0.95, mean=5, sd=2)
[1] 8.289707
```

To find the value *above* which 95% of the population falls, set the `lower.tail` argument to `F` as shown.

```
> qnorm(0.95, 5, 2, lower.tail=F)
[1] 1.710293
```

For other standard probability distributions, you must replace the `mean` and `sd` arguments with other parameters relevant to the distribution that you are using. Use the `help` function to see what these are.

**Example 7.7.** Finding quantiles for the normal distribution

A manufacturer of a special type of one-size glove wants to design the glove to fit at least 99% of the population. Hand span is known to be normally distributed with a mean of 195 mm and a standard deviation of 17 mm. What range of hand spans must the glove accomodate?

To find the value below which 0.5% of the population falls, use the command:

```
> qnorm(0.005, 195, 17)
[1] 151.2109
```

Similarly to find the value above which 0.5% of the population falls, use the command:

```
> qnorm(0.005, 195, 17, lower.tail=F)
[1] 238.7891
```

The remaining 99% of the population falls between these two values. So to accomodate 99% of the population, the gloves must be designed to fit hands with a span between 151 and 239 mm.

**Example 7.8.** Finding quantiles for the exponential distribution

Malfunctions in a particular type of electronic device are known to follow an exponential distribution with a mean time of 24 months until the device malfunctions. After how many months will 40% of the devices already have malfunctioned?

To find the length of time within which 40% of devices will have malfunctioned, use the command:

```
> qexp(0.4, 1/24)
[1] 12.25981
```

So 40% of devices will malfunction within 12.3 months.

**Example 7.9.** Finding quantiles for the Poisson distribution

The number of lobster ordered in a restaurant on a given day is known to follow a Poisson distribution with a mean of 20. If the manager wants to be able to satisfy all requests for lobster on at least 80% of days, how many lobster should they order each day?

To find the number of lobster requests that will not be exceeded on 80% of days, use the command:

```
> qpois(0.8, 20)
[1] 24
```

By ordering 24 lobster per day, the restaurant will be able to satisfy all requests for lobster on at least 80% of days.

## 7.4 Generating random numbers

R has a set of functions that allow you to generate random numbers from any of the standard probability distributions. This is useful for simulating data.

The functions names are formed from the prefix `r` and the relevant suffix from Table 7.1 on page 7.1. The first argument required by the functions is the quantity of random numbers to generate. This is followed by any parameters relevant to the distribution, which you can check using the `help` function.

For example, the command below generates 100 random numbers from a standard normal distribution and saves them in an object named `vector1`.

```
> vector1<-rnorm(100)
```

To generate random numbers from a non-standard normal distribution, add the `mean` and `sd` arguments. For example, to generate 100 random numbers from a normal distribution with a mean of 27.3 and a standard deviation of 4.1, use the command:

```
> vector2<-rnorm(100, 27.3, 4.1)
```

To generate a simple random sample from a range of numbers, you can use the `sample` function. For example, to select 20 random numbers between 1 and 100 without replacement, use the command:

```
> sample(1:100, 20)
```

---

```
[1] 58 76 87 14 95 68 98  2 47 96 12 49 44 21 23 34 84  6 29  5
```

---

To sample with replacement, set the `replace` argument to `T`.

```
> sample(1:100, 20, replace=T)
```

**Example 7.10.** Generating random numbers from a normal distribution.

Hand span in a particular population is known to be normally distributed with a mean of 195 mm and a standard deviation of 17 mm. To simulate the hand spans of three randomly selected people, use the command:

```
> rnorm(3, 195, 17)
```

---

```
[1] 186.376 172.164 195.504
```

---

**Example 7.11.** Generating random numbers from a binomial distribution

To simulate the number of sixes thrown in 10 rolls of a fair die, use the command:

```
> rbinom(1, 10, 1/6)
```

---

```
[1] 3
```

---

**Example 7.12.** Generating random numbers from a Poisson distribution

The number of lobsters ordered on any given day in a restaurant follows a Poisson distribution with a mean of 20. To simulate the number of lobsters ordered over a seven day period, use the command:

```
> rpois(7, 20)
```

```
[1] 19 10 13 23 21 13 25
```

**Example 7.13.** Generating random numbers from an exponential distribution

Malfunctions in a particular type of electronic device are known to follow an exponential distribution with a mean time of 24 months until the device malfunctions.

To simulate the time to malfunction for ten randomly selected devices, use the command:

```
> rexp(10, 1/24)
```

```
[1] 7.7949626 1.4280596 63.6650676 33.3343910 0.5911718 46.2797640
[7] 16.4355239 38.1404491 12.2637182 10.9453535
```

### Chapter Summary: Probability distributions

Task	Command
Find the value of the density function at value $x$	<code>dnorm(x, mean, sd)</code>
Obtain $P(X \leq x)$	<code>pnorm(x, mean, sd)</code>
Obtain $x$ where $P(X \leq x) = p$	<code>qnorm(p, mean, sd)</code>
Generate $n$ random numbers from a normal distribution	<code>rnorm(n, mean, sd)</code>



# Chapter 8

## Creating plots

---

One of the strong points of R is its ability to easily produce excellent quality, fully customisable plots and statistical graphics. It is possible to produce just about any type of statistical plot with R. This chapter explains how to create the most popular types.

This chapter concentrates on creating plots using the default settings. In Chapter 9 you will learn how to make your plots more presentable by changing titles, labels, colours and other aspects of the plot's appearance.

This chapter uses the `trees` and `iris` datasets (which are included with R), the `people2` dataset (available from the website) and the `sexeyetable` table object (created in Section 6.1).

### 8.1 Simple plots

To create a basic plot of a continuous variable against the observation number, use the `plot` function. For example to plot the `Height` variable from the `trees` dataset, use the command:

```
> plot(trees$Height)
```

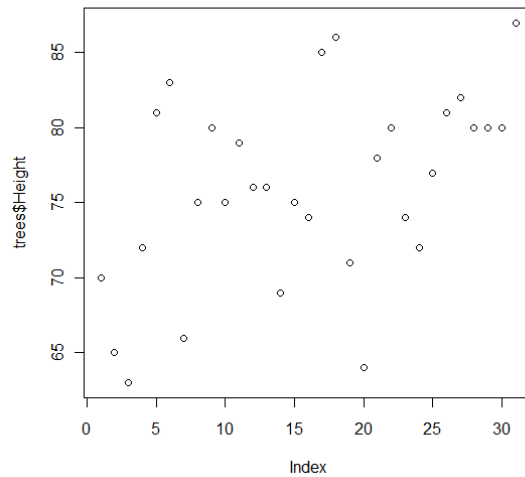
When you create a plot, it is displayed in a new window called the graphics device (or for Mac users, the Quartz device). Figure 8.1a shows how the plot looks.

The `plot` function does not just create basic one-dimensional plots. The type of plot created depends on the type and number of variables you give as input. You will see it used in different parts of this book to create other types of plots, including bar charts and scatter plots.

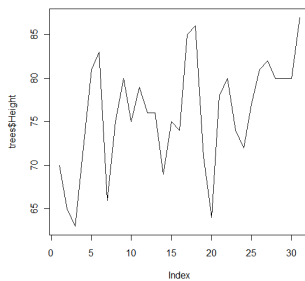
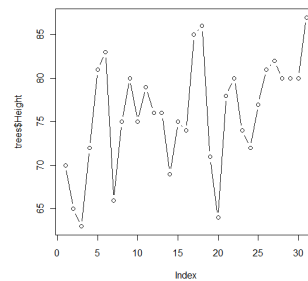
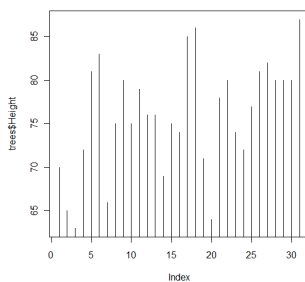
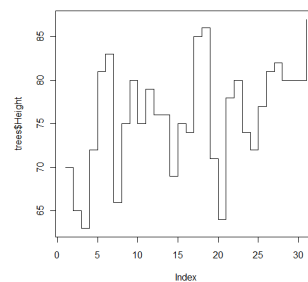
By default, R uses symbols to plot the data values. To use lines instead of symbols, set the `type` argument to `"l"` as shown below.

```
> plot(trees$Height, type="l")
```

Other possible values for the `type` argument include: `"b"` for both lines and symbols; `"h"` for vertical lines; and `"s"` for steps.<sup>34</sup> Figures 8.1b-e shows how the plot looks when these values are used. Depending on the nature of your data, some of these options will be more suitable than others.



(a) Default style

(b) `type="l"`(c) `type="b"`(d) `type="h"`(e) `type="s"`**Figure 8.1:** Simple plots created with the `plot` function



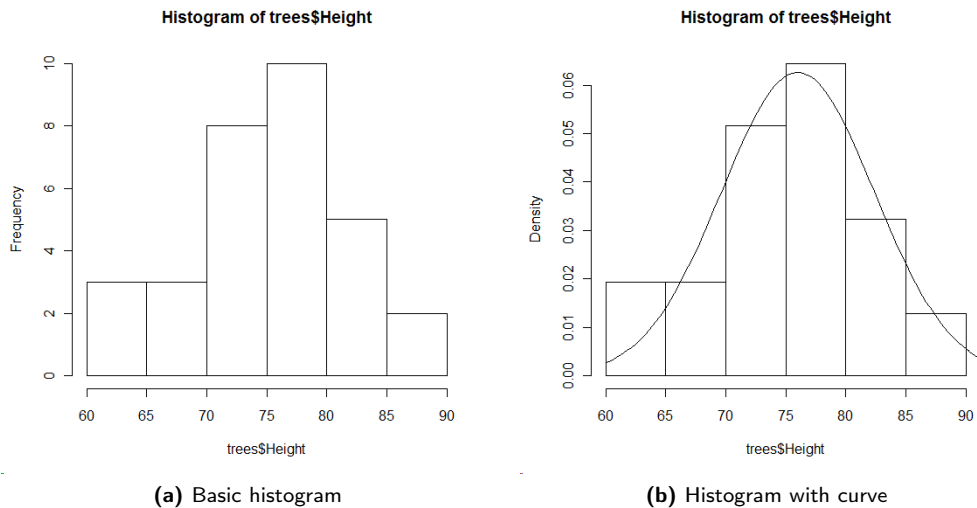
## 8.2 Histograms

### Histogram

A histogram is a plot for a continuous variable that allows you to assess its probability distribution.

You can create a histogram with the `hist` function, as shown below for the `Height` variable. Figure 8.2a shows the result.

```
> hist(trees$Height)
```



**Figure 8.2:** Histograms of the `Height` variable from the `trees` dataset

R automatically selects a suitable number of bars for the histogram. If you prefer, you can specify the number of bars with the `breaks` argument.

```
> hist(dataset$variable, breaks=15)
```

By default, R creates a histogram of frequencies. To create a histogram of densities (so that the total area of the histogram is equal to one), set the `freq` argument to `F`.

```
> hist(dataset$variable, freq=F)
```

You can use the `curve` function to fit a normal distribution curve to the data. This allows you to see how well the data fits the normal distribution. Use the `curve` function directly after the `hist`

function, while the histogram is still displayed in the graphics device. Adding a density curve is only appropriate for a histogram of densities, so remember to set the `freq` argument to `F`.

```
> hist(trees$Height, freq=F)
> curve(dnorm(x, mean(trees$Height), sd(trees$Height)), add=T)
```

The result is shown in Figure 8.2b.

If the variable has any missing data values, remember to set the `na.rm` argument to `T` for the `mean` and `sd` functions as shown.

```
> hist(dataset$variable, freq=F)
> curve(dnorm(x, mean(dataset$variable, na.rm=T),
  sd(dataset$variable, na.rm=T)), add=T)
```

The `curve` function is discussed in more detail in Sections 8.10 and 9.7.2, and the `dnorm` function is covered in Section 7.1.

## 8.3 Normal probability plots

### Normal probability plot

A normal probability plot is a plot for a continuous variable that helps to determine whether a sample is drawn from a normal distribution. If the data is drawn from a normal distribution, the points will fall approximately in a straight line. If the data points deviate from a straight line in any systematic way, it suggests that the data is not drawn from a normal distribution.<sup>2</sup>

You can create a normal probability plot using the `qqnorm` function, as shown for the `Height` variable.

```
> qqnorm(trees$Height)
```

You can also add a reference line to the plot, which makes it easier to determine whether the data points are falling into a straight line. To add a reference line, use the `qqline` function directly after the `qqnorm` function as shown.

```
> qqnorm(trees$Height)
> qqline(trees$Height)
```

The result is shown in Figure 8.3.

There is also a function called `qqplot` which allows you to create quantile plots for comparing data with other standard probability distributions besides the normal distribution. Enter `help(qqplot)` for more details.

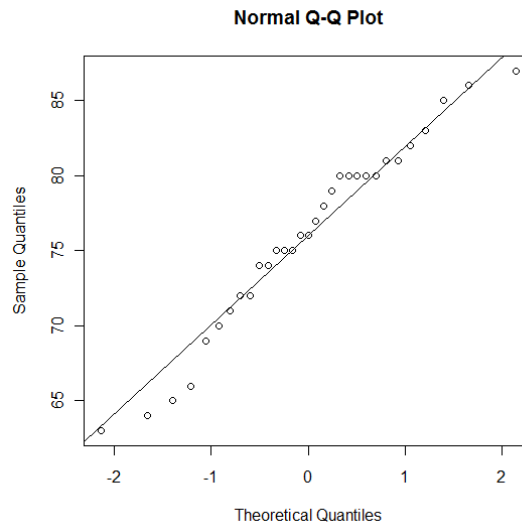


Figure 8.3: Normal probability plot of the `Height` variable

## 8.4 Stem-and-leaf plots

### Stem-and-leaf plot

The stem-and-leaf plot (or stemplot) is another popular plot for a continuous variable. It is similar to a histogram, but all of the data values can be read from the plot.

You can create a stem-and-leaf plot with the `stem` function, as shown below for the `Volume` variable.

```
> stem(trees$Volume)
```

Unlike other plots, the output is displayed in the command window rather than the graphics device.

```
The decimal point is 1 digit(s) to the right of the |
```

```
1 | 00066899
2 | 00111234567
3 | 24568
4 | 3
5 | 12568
6 |
7 | 7
```

## 8.5 Bar charts

### Bar chart

A bar chart is a plot for summarising categorical data. A simple bar chart summarises one categorical variable by displaying the number of observations in each category. Grouped and stacked bar charts summarise two categorical variables by displaying the number of observations for each combination of categories.

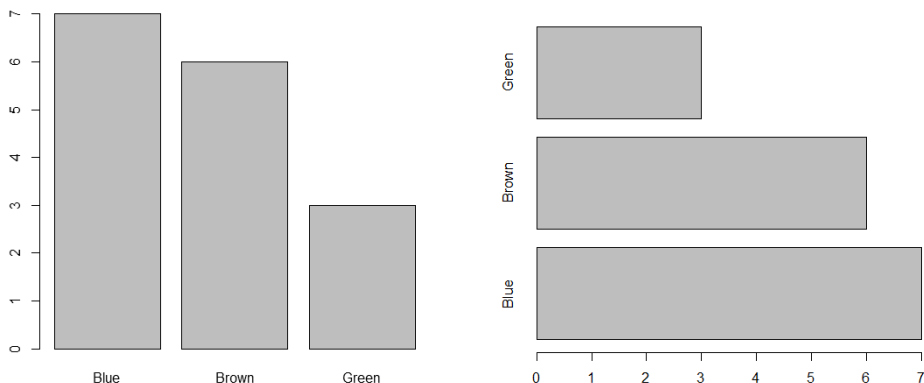
You can create bar charts with the `plot` and `barplot` functions. If you have raw data such as the `Eye.Colour` variable in the `people2` dataset, use the `plot` function as shown below. The result is given in Figure 8.4a.

```
> plot(people2$Eye.Colour)
```

When creating a bar chart from raw data, the variable must have the `factor` class. Section 3.3 explains how to check the class of a variable and change it if necessary.

To create a bar chart from a previously saved one-dimensional table object (see Section 6.1), use the `barplot` function.

```
> barplot(tableobject)
```



(a) Default style

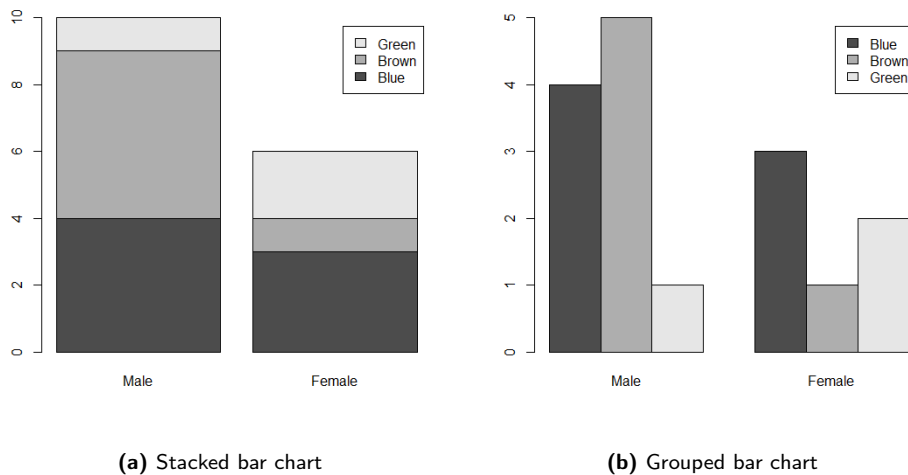
(b) with `horiz=T`

**Figure 8.4:** Bar charts of the `Eye.colour` variable from the `people2` dataset

For a horizontal bar chart like the one shown in Figure 8.4b, set the `horiz` argument to `T`. This works with both the `plot` and `barplot` functions.

```
> plot(people2$Eye.Colour, horiz=T)
```

The `barplot` function can also create a bar chart for two categorical variables, known as a multiple bar chart. There are two styles of multiple bar chart, as shown in Figure 8.5.



**Figure 8.5:** Two styles of multiple bar chart of `Eye.Colour` by Sex

The first is the stacked style, which you can create from a two-dimensional table object as shown.

```
> barplot(sexeyetable, legend.text=T)
```

The second style is a grouped bar chart, which displays the categories side-by-side. Create this style by setting the `beside` argument to `T`.

```
> barplot(sexeyetable, beside=T, legend.text=T)
```

To create multiple bar charts from raw data, first create a two-dimensional table object as explained in Section 6.1. Alternatively you can nest the `table` function inside the `barplot` function as shown below.

```
> barplot(table(people2$Eye.Colour, people2$Sex), legend.text=T)
```

## 8.6 Pie charts

### Pie chart

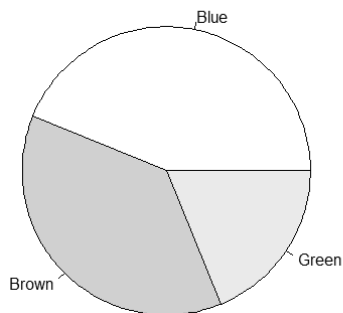
The pie chart is a plot for a single categorical variable and is an alternative to the bar chart. It displays the number of observations in each category as a portion of the total number of observations.

You can create a pie chart with the `pie` function. If you have previously created a one-dimensional table object (see Section 6.1), you can use the function directly.

```
> pie(tableobject)
```

To create a pie chart from raw data, nest the `table` function inside the `pie` function as shown below. The result is given in Figure 8.6.

```
> pie(table(people2$Eye.Colour))
```



**Figure 8.6:** Pie chart of the `Eye.Colour` variable

If your variable has missing data and you want this to appear as an additional section in the pie chart, set the `useNA` argument to `"ifany"`.

```
> pie(table(dataset$variable, useNA="ifany"))
```

## 8.7 Scatter plots

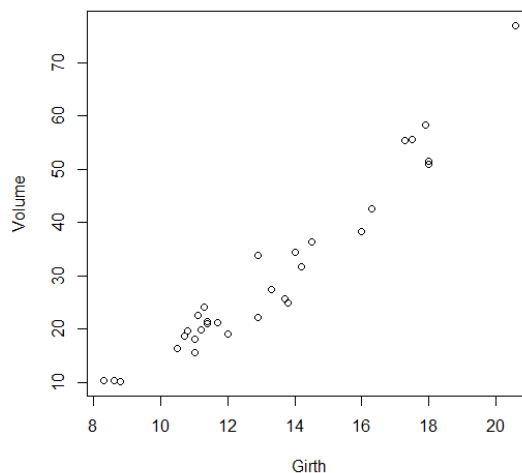
### Scatter plot

A scatter plot is a plot for two continuous variables, which allows you to examine the relationship between them.

You can create a scatter plot with the `plot` function, by giving two numeric variables as input. The first variable is displayed on the vertical axis and the second variable on the horizontal axis. For example, to plot `Volume` against `Girth` for the `trees` dataset, use the command:

```
> plot(Volume~Girth, trees)
```

The output is shown in Figure 8.7.



**Figure 8.7:** Scatter plot of `Volume` against `Girth`, for the `trees` dataset

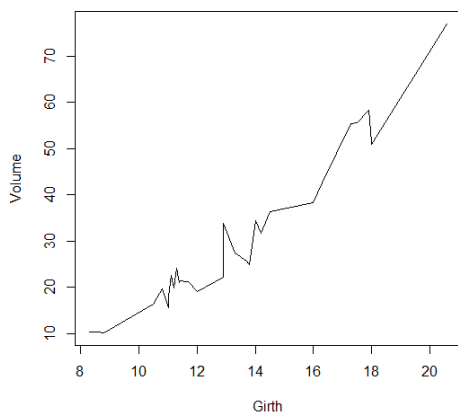
To add a line of best fit (linear regression line), use the `abline` function directly after the `plot` function as shown.

```
> plot(Volume~Girth, trees)
> abline(coef(lm(Volume~Girth, trees)))
```

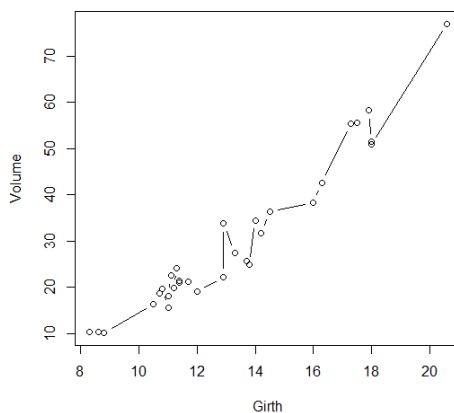
You will learn more about the `abline` function in Section 9.7.1, and the `lm` and `coef` functions in Chapter 11.

If it is meaningful for your data, you can join the data points with lines by setting the `type` argument to `"l"`. To create a plot with both symbols and lines, set it to `"b"`. The effect is shown in Figure 8.8.

```
> plot(Volume~Girth, trees, type="l")
```



(a) `type="l"`



(b) `type="b"`

Figure 8.8: Scatter plots with (a) lines and (b) both lines and symbols

## 8.8 Scatterplot matrices

### Scatterplot matrix

A scatterplot matrix is a collection of scatter plots showing the relationship between each pair in a set of variables. It allows you to examine the correlation structure of a dataset.

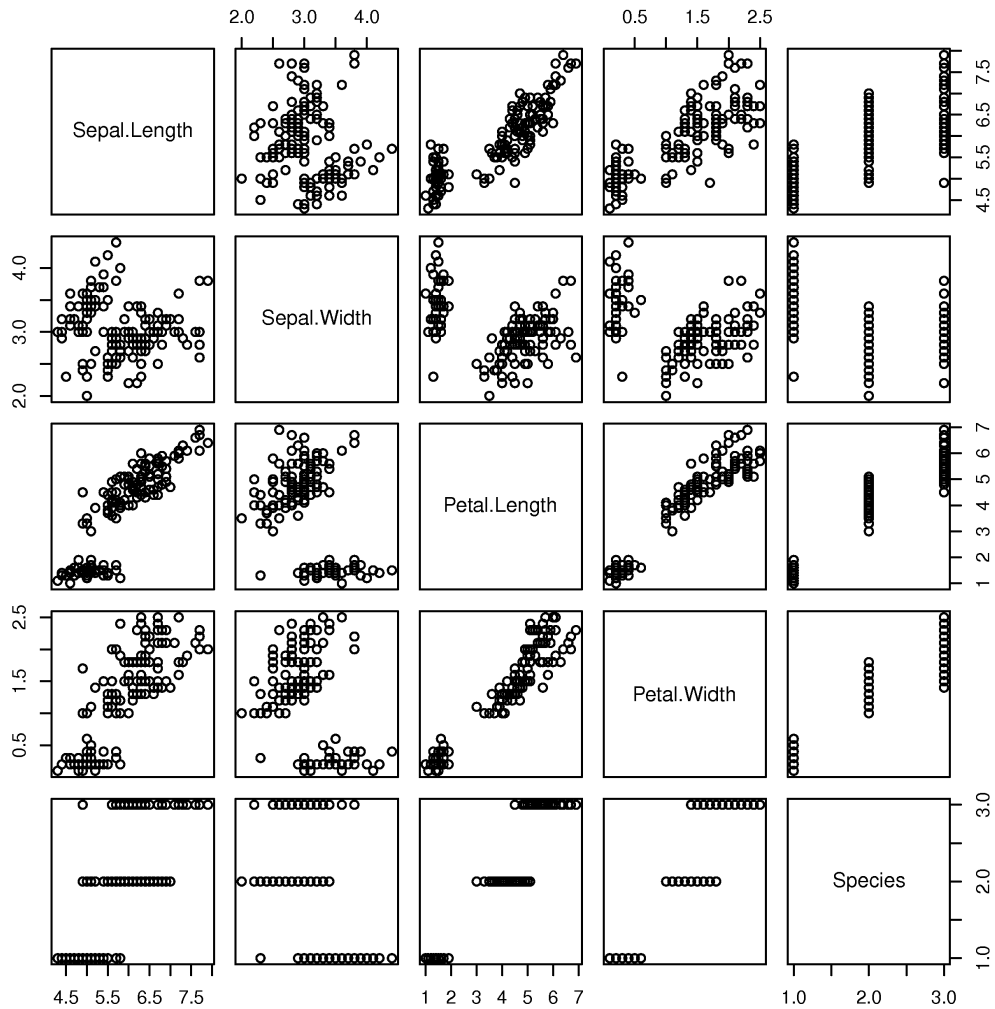
To create a scatterplot matrix of the variables in a dataset, use the `pairs` function. For example, to create a scatterplot matrix for the `iris` dataset use the command:

```
> pairs(iris)
```

The output is shown in Figure 8.9.

You can also select a subset of variables to include in the matrix. For example to include just the `Sepal.Length`, `Sepal.Width` and `Petal.Length` variables, use the command:



Figure 8.9: Scatterplot matrix for the `iris` dataset

```
> pairs(~Sepal.Length+Sepal.Width+Petal.Length, iris)
```

Alternatively you can use bracket notation or the `subset` function to select or exclude variables from a dataset, as explained in Sections 1.7 and 3.11 respectively.

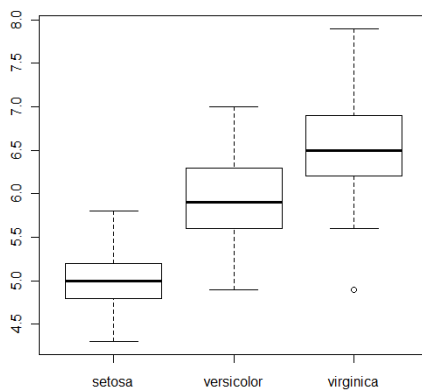
## 8.9 Box plots

### Box plot

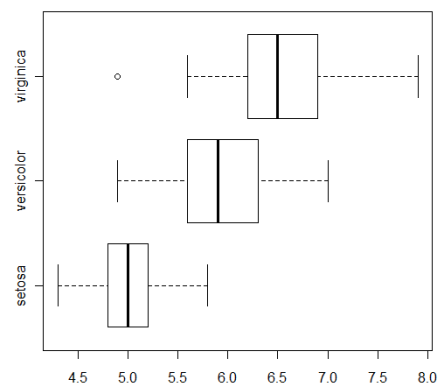
A box plot (or box-and-whisker plot) presents summary statistics for a continuous variable in a graphical form. Usually a categorical variable is used to group the observations, so that the plot summarises the distribution for each category. This helps you to understand the relationship between a categorical and a continuous variable.

You can create a box plot with the `boxplot` function. For example, the following command creates a box plot of `Sepal.Length` grouped by `Species` for the `iris` dataset. Figure 8.10a shows the result.

```
> boxplot(Sepal.Length~Species, iris)
```



(a) Default style



(b) with `horizontal=T`

**Figure 8.10:** Box plot of `Sepal.Length` grouped by `Species`, for the `iris` dataset

To create a horizontal box plot as shown in Figure 8.10b, set the `horizontal` argument to `T`.

```
> boxplot(Sepal.Length~Species, iris, horizontal=T)
```

You can also create a single box plot for a continuous variable (without any grouping), as shown below.

```
> boxplot(iris$Sepal.Length)
```

By default, the whiskers extend to a maximum of 1.5 times the interquartile range of the data, with any values beyond this is shown as outliers. If you want the whiskers to extend to the minimum and maximum values, set the `range` argument to 0.

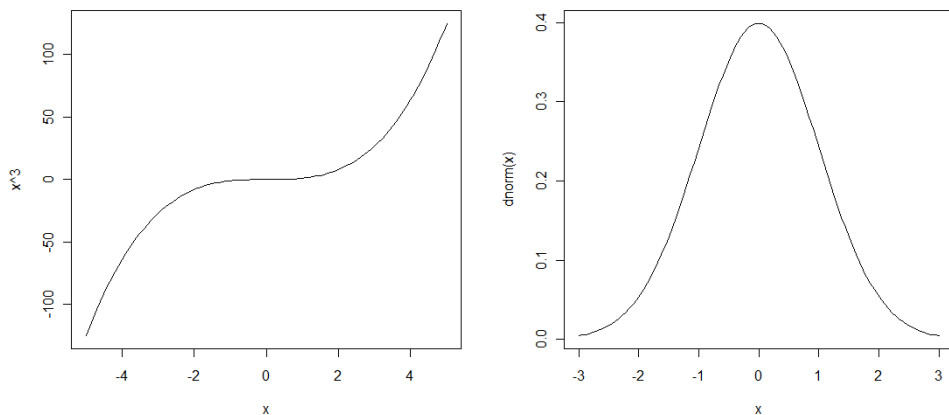
```
> boxplot(Sepal.Length~Species, iris, range=0)
```

## 8.10 Plotting a function

You can plot a mathematical function such as  $y = t^3$  with the `curve` function. Express the mathematical function in terms of `x`, as shown below. Figure 8.11a shows the result.

```
> curve(x^3)
```

To display the most interesting part of the curve, you may need to adjust the axes as explained in Section 9.2.



(a)  $y = x^3$

(b) Density function for the normal distribution

**Figure 8.11:** Plots created with the `curve` function

To plot a curve of the probability density function (pdf) for a standard probability distribution, use the relevant density function from Table 7.1 on page 84. For example, to plot the pdf of the standard normal distribution, use the command:

```
> curve(dnorm(x))
```

The results are shown in Figure 8.11b.

## 8.11 Exporting and saving plots

Before exporting or saving a plot, resize the graphics device window so that the plot has the required dimensions.

The simplest way to export a plot is to right-click on the plot and select 'Copy as bitmap'. The image is copied to the clipboard, so that you can paste it into a Microsoft Word<sup>®</sup> or Powerpoint<sup>®</sup> file or a graphics package.

Alternatively, R allows you to save the image to a variety of file types including png, jpeg, bmp and pdf. With the graphics device as the active window, select 'Save as' from the 'File' menu. You will be given a number of options for saving the image.

Mac users can copy the image to the clipboard by selecting 'Edit' then 'Copy', or save the image as a pdf file by selecting 'File' then 'Save'.

Linux users can save the current plot by entering the command:

```
> savePlot("/home/Username/folder/filename.png", type="png")
```

Other possible file types are `bmp`, `jpeg` and `tiff`.

**Chapter summary: Creating plots**

Plot type	Command
Basic plot	<code>plot(dataset\$variable)</code>
Line plot	<code>plot(dataset\$variable, type="l")</code>
Histogram	<code>hist(dataset\$variable)</code>
Normal probability plot	<code>qqnorm(dataset\$variable)</code>
Stem-and-leaf plot	<code>stem(dataset\$variable)</code>
Bar chart	<code>plot(dataset\$factor1)</code> <code>barplot(tableobject1D)</code>
Stacked bar chart	<code>barplot(tableobject2D)</code>
Grouped bar chart	<code>barplot(tableobject2D, beside=T)</code>
Pie chart	<code>pie(tableobject1D)</code> <code>pie(table(dataset\$factor1))</code>
Scatter plot	<code>plot(yvar~xvar, dataset)</code>
Scatterplot matrix	<code>pairs(dataset)</code>
Single box plot	<code>boxplot(dataset\$variable)</code>
Grouped box plot	<code>boxplot(variable~factor1, dataset)</code>
Function plot	<code>curve(f(x))</code>



## Chapter 9

# Customising your plots

---

The previous chapter explained how to create the most popular types of plots. This chapter explains how to customise the appearance of your plots to create a more polished look. This includes adding titles and labels, adjusting the font style of text, changing the colour of the various elements of the plot, changing the style of lines and symbols, and adding additional items.

Most changes to the appearance of a plot are made by giving additional arguments to the plotting function. In this chapter, the `plot` function is used to illustrate. However most of the arguments can also be used with other plotting functions such as `hist` and `boxplot`.

### 9.1 Titles and labels

Whenever you create a plot, R add default axis labels and sometimes a default title too. You can overwrite these with your own titles and axis labels.

You can add a title to your plot with the `main` argument as shown.

```
> plot(dataset$variable, main="This is the title")
```

To add a subtitle to be displayed underneath the plot, use the `sub` argument.

```
> plot(dataset$variable, sub="This is the subtitle")
```

If you have a very long title or subtitle that should be displayed over two or more lines, use `\n` to tell R where the line break should be.

```
> plot(dataset$variable, main="This is the first line of the title\nand this is the second line")
```

To change the x and y axis labels, use the `xlab` and `ylab` arguments.

```
> plot(dataset$variable, xlab="The x axis label", ylab="The y axis label")
```

You can also add additional arguments to control the appearance of the text in the title, subtitle, axis labels and axis units. Table 9.1 gives a list of these.

Aspect		Argument	Possible values
Font family	All text	<code>family="serif"</code>	<code>sans</code> , <code>serif</code> , <code>mono</code>
Font type	Title	<code>font.main=2</code>	1 (normal); 2 (bold); 3 (italic); 4 (bold & italic).
	Subtitle	<code>font.sub=2</code>	
	Axis labels	<code>font.lab=2</code>	
	Axis units	<code>font.axis=2</code>	
Font size	Title	<code>cex.main=2</code>	Relative to default, e.g. 2 is twice normal size
	Subtitle	<code>cex.sub=2</code>	
	Axis labels	<code>cex.lab=2</code>	
	Axis units	<code>cex.axis=2</code>	
Font colour	Title	<code>col.main="red"</code>	See Section 9.3 for details of how to specify colours
	Subtitle	<code>col.sub="red"</code>	
	Axis labels	<code>col.lab="red"</code>	
	Axis units	<code>col.axis="red"</code>	

**Table 9.1:** Additional arguments for customising the font of titles and labels.

For example, the command below add a title, sets the font family to `serif` and the title font to be grey, italic and three times the usual size.

```
> plot(trees$Height, main="This is the title", family="serif",
      col.main="grey80", cex.main=3, font.main=3)
```

For plots that include a categorical variable such as the bar chart, pie chart and box plot, you can also customise the category labels. If you plan to create several plots then it is easier to change the level names for the factor variables in your dataset, as explained in Section 3.6. This means you won't have to modify the labels every time you plot the variable. However it is also possible to modify the labels at the time of plotting.

For pie charts, use the `labels` argument.

```
> pie(table(dataset$variable), labels=c("Label1", "Label2",
    "Label3"))
```

For bar charts, use `names.arg`:

```
> plot(dataset$variable, names.arg=c("Label1", "Label2", "Label3"))
```



For box plots, use `names`:

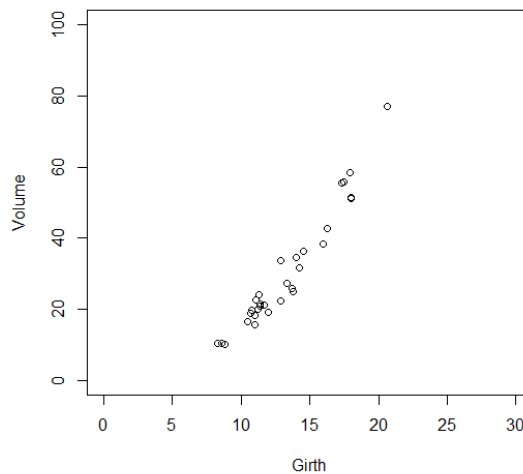
```
> boxplot(variable~factor, dataset, names=c("Label1", "Label2",  
      "Label3"))
```

## 9.2 Axes

When plotting continuous variables, R automatically select sensible axis limits for your data. However if you want to change them, you can do so with the `xlim` and `ylim` arguments. For example, to change the axis limits to 0-30 for the horizontal axis and 0-100 for the vertical axis, use the command:

```
> plot(Volume~Girth, trees, xlim=c(0, 30), ylim=c(0, 100))
```

Figure 9.1 shows the results.



**Figure 9.1:** Plot with adjusted axis ranges

To rotate the axis numbers for the vertical axis so that they are in upright rather than in line with the vertical axis, set the `las` argument to 1.

```
> plot(dataset$variable, las=1)
```

## 9.3 Colours

R allows you to change the colour of every component of a plot. The most important argument is the `col` argument, which allows you to change the plotting colour. This is the colour of the lines or symbols that are used to represent the data values. For histograms, bar charts, pie charts and box plots it is the colour of the area enclosed by the shapes.

There are three ways to specify colours. The first is to give the name of the colour:

```
> plot(dataset$variable, col="red")
```

The `colours` function displays a list of recognised colour names (the American-English spelling `colors` also works).

```
> colours()
```

The second way to specify colours is with a number between 1 and 8, which correspond to the set of basic colours shown in Table 9.2.

Value	Name
1	black
2	red
3	green3
4	blue
5	cyan
6	magneta
7	yellow
8	gray

**Table 9.2:** Basic plotting colours. Note that these may vary for some platforms. Enter `palette()` to view the colours for your platform.

For example to specify red as the plotting colour, use the command:

```
> plot(dataset$variable, col=2)
```

Finally, if you require more precise control over the appearance of the plot then you can specify the colour using the hexadecimal RGB format. The command below changes the plotting colour to red using the hexadecimal colour specification.

```
> plot(dataset$variable, col="#FF0000")
```

As well as specifying a single plotting colour, you can give a list of colours for R to use. This is particularly useful for pie charts and bar charts, where each category is given a different colour from the list.

```
> plot(dataset$variable, col=c("red", "blue", "green"))
```

The numeric notation is useful for this purpose.

```
> plot(dataset$variable, col=1:8)
```

There are also functions such as `rainbow`, which help to easily create a visually appealing set of colours. The command below creates five colours, evenly spaced along the spectrum.

```
> plot(dataset$variable, col=rainbow(5))
```

Other colour scheme functions include `heat.colors`, `terrain.colors`, `topo.colors` and `cm.colors`.

You can change the colour of the other elements of the plot in a similar way, using the arguments given in Table 9.3.

Component	Argument
Plotting symbol, line or area	<code>col</code>
Foreground (axis, tick marks and other elements)	<code>fg</code>
Background	<code>bg*</code>
Title text	<code>col.main</code>
Subtitle text	<code>col.sub</code>
Axis label text	<code>col.lab</code>
Axis numbers	<code>col.axis</code>

**Table 9.3:** Arguments for changing the colour of plot components.<sup>38</sup> \*The background colour must be changed with the `par()` function.

Note that the background colour cannot be change from within the plotting function, but must be changed with the `par` function as shown below.

```
> par(bg="red")
> plot(dataset$variable)
```

Changes made with the `par` function apply to all plots for the remainder of the session, or until overwritten. See Section 9.11 for more details on using the `par` function.

## 9.4 Plotting symbols

This section applies only to plots where symbols are used to represent the data values, such as scatter plots.

You can change the plotting symbol with the `pch` argument, as shown below. The numbers 1 to 25 correspond to the symbols given in Table 9.4.

```
> plot(dataset$variable, pch=5)
```

Number	Symbol	Number	Symbol	Number	Symbol
1	○	10	⊕	19	●
2	△	11	⊗	20	•
3	+	12	⊞	21	◐
4	×	13	⊠	22	■
5	◇	14	⊡	23	◆
6	▽	15	■	24	▲
7	⊠	16	●	25	▼
8	*	17	▲		
9	⊕	18	◆		

**Table 9.4:** Plotting symbols for use with the `pch` argument

Symbol numbers 21 to 25 allow you to specify different colours for the symbol border and fill. The border colour is specified with the `col` argument and the fill colour with the `bg` argument.<sup>29</sup>

```
> plot(dataset$variable, pch=21, col="red", bg="blue")
```

Alternatively, you can select any single keyboard character to use as the plotting symbol by placing it between quotation marks, as shown below for the dollar symbol.

```
> plot(dataset$variable, pch="$")
```

You can adjust the size of the plotting symbol with the `cex` argument. The size is specified relative to the normal size. For example to make the plotting symbols 5 times their normal size, use the command:

```
> plot(dataset$variable, cex=5)
```

## 9.5 Plotting lines

This section applies only to plots with lines, such as scatter plots and basic plots created with the `plot` function and where the `type` argument is set to "l" or "b". It also applies to functions that add additional lines to existing plots, such as `abline`, `curve`, `segments` and `lines` (covered in Sections 9.7 and 9.8).

You can change the line type with the `lty` argument. The numbers 1 to 6 correspond to the line types given in Table 9.5.

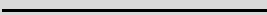





Number	Name	Line Style
1	solid	
2	dashed	
3	dotted	
4	dotdash	
5	longdash	
6	twodash	

Table 9.5: Plotting line types

For example, you can change the line type to dashed as shown.

```
> plot(dataset$variable, type="l", lty=2)
```

You can also select a line type by name.

```
> plot(dataset$variable, type="l", lty="dashed")
```

You can adjust the thickness of the line with the `lwd` argument. The line thickness is specified relative to the standard line thickness. For example to make the line three times its usual thickness, use the command:

```
> plot(dataset$variable, type="l", lwd=3)
```

## 9.6 Shaded areas

This section applies only to plots with shaded areas such as bar charts, pie charts and histograms.

The *density* of a shaded area is the number of lines per inch used to shade. You can change the density of the shaded areas with the `density` argument and the angle of the shading with the `angle` argument.

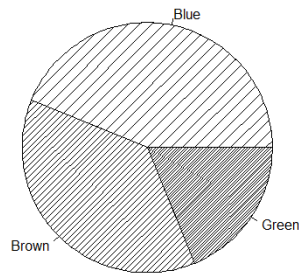
```
> barplot(tableobject, density=20, angle=30)
```

As a guideline, density values between 5 and 80 give discernible variation, while a value of 150 is indistinguishable from solid colour.

You can also give a list of densities and R will shade each section of a pie or bar chart with a different density from the list.

```
> pie(table(people2$Eye.Colour), density=c(10, 20, 40))
```

Figure 9.2 shows the result.



**Figure 9.2:** Pie chart with a different density of shading for each category

## 9.7 Adding additional items to plots

This section introduces some functions that add extra items to plots. You can use them with any type of plot. First create the plot using `plot` or another plotting function. While the plot is still displayed in the graphics device, enter the relevant command from this section. The item is added to the current plot.

### 9.7.1 Adding additional straight lines

You can add straight lines to your plot with the `abline` function.

To add a vertical line at  $x = 5$  use the command:

```
> abline(v=5)
```

To add a horizontal line at  $y = 2$  use:

```
> abline(h=2)
```

To add a diagonal line with intercept 2 and slope 3 (i.e. the line  $y = 2 + 3x$ ) use the command:

```
> abline(a=2, b=3)
```

To draw a line segment (a line that extends from one point to another), use the `segments` function. For example, the command below draws a straight line from coordinate (0,1) to coordinate (3,4).

```
> segments(0,1,3,4)
```

You can change the line colour, type, and thickness with the `col`, `lty` and `lwd` arguments as explained in Sections 9.3 and 9.5.

### 9.7.2 Adding a mathematical function curve

To add a mathematical function curve to your plot, use the `curve` function (introduced in Section 8.10). By default the `curve` function creates a new plot. To superimpose the curve over the current plot, set the `add` argument to `T` as shown.

```
> curve(x^2, add=T)
```

### 9.7.3 Adding labels and text

You can add text to your plot with the `text` function. For example, to add the text '*Text String*' centred at coordinates (3,4), use the command:

```
> text(3, 4, "Text String")
```

You can adjust the appearance of the text with `family`, `font`, `cex` and `col` arguments. For example to add a piece of red, italic text which is twice the default size, use the command:

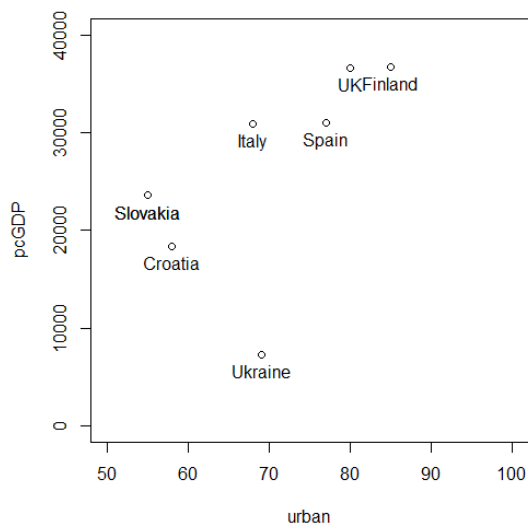
```
> text(3, 4, "Text String", col="red", font=3, cex=2)
```

The `text` function is useful if you want to add labels to all of the points in a scatter plot using text taken from a third variable or from the row names of your dataset. For example, the commands below create a scatter plot of per capita GDP against urban population, where each observation is labelled with the country name (using the `CIAdata` dataset shown on page 50).

```
> plot(pcGDP~urban, CIAdata, xlim=c(50, 100), ylim=c(0,40000))  
> text(pcGDP~urban, CIAdata, CIAdata$country, pos=1)
```

Figure 9.3 shows the results.

The `pos` argument tells R where to place the labels in relation to the coordinates: 1 is below; 2 is to the left, 3 is above; 4 is to the right. You can also use the `offset` argument to adjust the distance between the coordinate and the label.



**Figure 9.3:** Results of using the `text` function to add labels

### 9.7.4 Adding a grid

To add a grid to a plot (as shown in Figure 9.4a) use the `grid` function.

```
> grid()
```

By default, the grid lines are aligned with the axis tick marks. Alternatively, you can specify how many grid lines to display on each axis.

```
> grid(3, 3)
```

To add horizontal grid lines only (as shown in Figure 9.4b), use the command:

```
> grid(nx=NA, ny=NULL)
```

For vertical grid lines only, use:

```
> grid(ny=NA)
```

By default R uses grey dotted lines for the grid, but you can adjust the line style with the `col`, `lty`, and `lwd` arguments as explained in Sections 9.3 and 9.5.



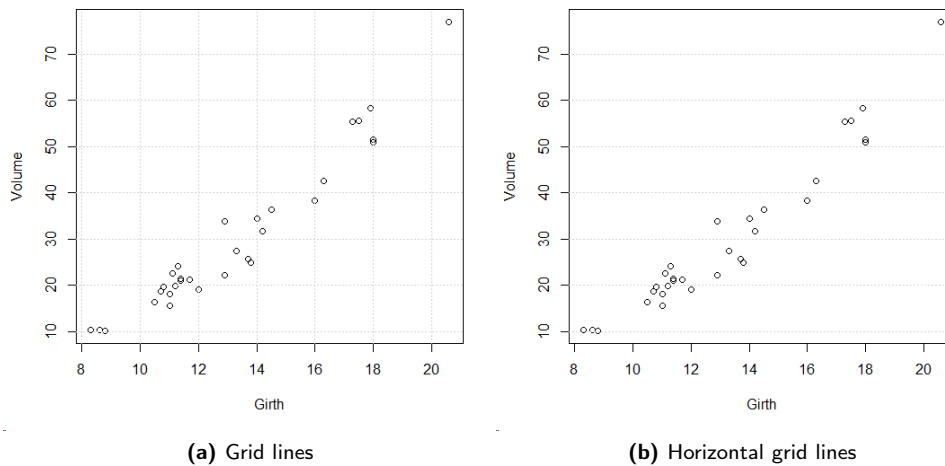


Figure 9.4: Plots with grid lines

### 9.7.5 Adding arrows

You can add an arrow to the current plot with the `arrows` function. For example, to draw an arrow from the point (0,1) to the point (4,5) use the command:

```
> arrows(0,1,4,5)
```

For a double ended arrow, set the `code` argument to 3.

```
> arrows(0,1,4,5, code=3)
```

You can adjust the line style using the `col`, `lty` and `lwd`, as explained in Sections 9.3 and 9.5. You can specify the length of the arrow head (in inches) with the `length` argument, and the angle between the arrow head and arrow body (in degrees) with the `angle` argument.

```
> arrows(0,1,4,5, angle=20, length=0.1)
```

## 9.8 Overlaying plots

R has two functions called `points` and `lines` that allow you to superimpose one set of data values over another.

The `points` function is very similar to the `plot` function. However instead of creating a new plot, it adds the data points to whichever plot is currently displayed in the graphics device. The `lines` function is very similar to the `points` function, except that it uses lines rather than symbols to plot the data (i.e. the default value for the `type` argument is `"l"`).

These functions are useful if you want to plot two or more variables on the same axis (as demonstrated in Example 9.1) or use different plotting symbols to represent different categories (as shown in Example 9.2).

**Example 9.1.** Overlay plot using `fiveyearreport` data

Consider the `fiveyearreport` dataset shown in Figure 9.5, which gives the UK sales of three supermarket chains over a five year period. Suppose you want to display the sales data for Tesco, Sainsburys and Morrisons in the same plot.

	Year	Tesco	Sainsburys	Morrisons
1	2007	35580	18518	12462
2	2008	37949	19287	12969
3	2009	41520	20383	14528
4	2010	42254	21421	15410
5	2011	44570	22943	16479

**Figure 9.5:** `fiveyearreport` dataset. See Appendix B for more details.

To create this plot, first plot the data for Tesco in the usual way, making sure to set the axis ranges so that they are wide enough to also accommodate the data for Sainsburys and Morrisons.

```
> plot(Tesco~Year, fiveyearreport, type="b", ylab="UK Sales (£M)",
      ylim=c(0, 50000))
```

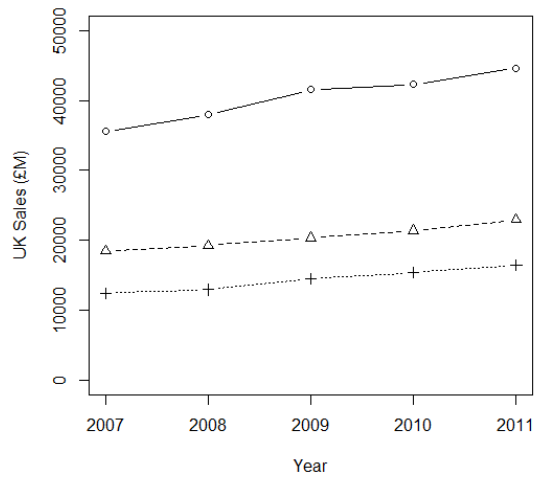
Once the plot is displayed in the graphics device, you can add the data for Sainsburys and Morrisons using the `lines` function, which superimposes the data over the current plot. Use the `pch` and `lty` arguments to change the symbol and line type, so that the data for the different chains can be distinguished. Alternatively you could use the `col` argument to give each chain a different colour.

```
> lines(Sainsburys~Year, fiveyearreport, type="b", pch=2, lty=2)
> lines(Morrisons~Year, fiveyearreport, type="b", pch=3, lty=3)
```

Figure 9.6 shows the results. The plot will require a legend to identify the chains, which you can add with the `legend` function as explained in Section 9.9.

**Example 9.2.** Overlay plot using `iris` data

Suppose you want to create a scatter plot of sepal length against sepal width for the `iris` data, using a different plotting symbol to represent each iris species. To create this plot, you will need



**Figure 9.6:** Overlay plot for `fiveyearreport` dataset

to plot the data for each species separately using the `points` function, overlaying them onto the same plot.

First plot the original (complete) data with the `plot` function, adding any labels or titles that you require. Set the `type` argument to `"n"` to prevent the data values from being added to the plot. This creates an empty axis which is the right size to accommodate all of the data for all three species.

```
> plot(Sepal.Length~Sepal.Width, iris, type="n")
```

Next, plot the data for each species separately with the `points` function. Use the `subset` argument to select each species in turn. Use the `pch` argument to select a different plotting symbol for each species. Alternatively you could use the `col` argument to give each species a different coloured symbol.

```
> points(Sepal.Length~Sepal.Width, iris, subset=Species=="setosa",
  pch=10)
> points(Sepal.Length~Sepal.Width, iris,
  subset=Species=="versicolor", pch=16)
> points(Sepal.Length~Sepal.Width, iris,
  subset=Species=="virginica", pch=1)
```

The result is shown in Figure 9.7. The legend is added with the `legend` function, as explained in Section 9.9.

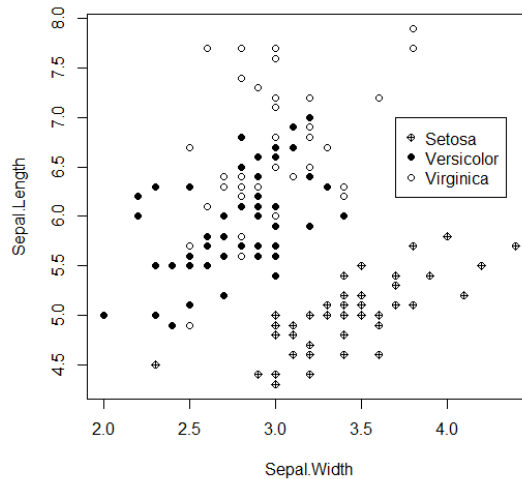


Figure 9.7: Overlay plot for the `iris` data

## 9.9 Adding a legend

You can add a legend to your plot with the `legend` function. The function adds a legend to whichever plot is currently displayed in the graphics device.

The following command creates the legend shown in Figure 9.7.

```
> legend(3.7, 7, legend=c("Setosa", "Versicolor", "Virginica"),
  pch=c(10, 16, 1))
```

The first two arguments (3.7 and 7) are the x and y coordinates for the top left-hand corner of the legend. Besides using coordinates, there are two other ways of specifying the position of the legend. The first is by using one of the location names: "top", "bottom", "left", "right", "center", "topright", "topleft", "bottomright" or "bottomleft".

```
> legend("topright", legend=c("Setosa", "Versicolor", "Virginica"),
  pch=c(10, 16, 1))
```

The second is by using `locator(1)`. This allows you to manually select the position for the top left-hand corner of the legend with your mouse.

```
> legend(locator(1), legend=c("Setosa", "Versicolor", "Virginica"),
  pch=c(10, 16, 1))
```

The `legend` argument gives the labels to be displayed on the legend. The `pch` argument gives the plotting symbols that correspond to each of the labels.

You can substitute the `pch` argument with the `lty`, `lwd`, `col`, `cex`, `fill`, `density` and `angle` arguments, according to what is relevant for your plot. You may need to use two or more of them. The examples below demonstrate how to create some different types of legend. The results are shown in Figure 9.8.

**Example 9.3.** Legend showing shaded areas of different densities

To create a legend for shaded areas of different densities as shown in Figure 9.8a, use the command:

```
> legend(locator(1), legend=c("Label1", "Label2", "Label3"),
        density=c(10,20,40))
```

**Example 9.4.** Legend showing different line and symbol types

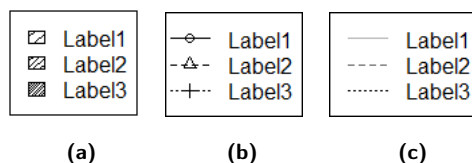
To create a legend for different line and symbol types as shown in Figure 9.8b, use the command:

```
> legend(locator(1), legend=c("Label1", "Label2", "Label3"),
        lty=1:3, pch=1:3)
```

**Example 9.5.** Legend with lines of different colours and types

To create a legend for lines of different types and colours as shown in Figure 9.8c, use the command:

```
> legend(locator(1), legend=c("Label1", "Label2", "Label3"),
        col=c("black", "grey40", "grey70"), lty=1:3)
```



**Figure 9.8:** Different types of legend

To display a legend over more than one column, use the `ncol` argument to specify the number of columns.

```
> legend(x, y, legend=c("Label1", "Label2", "Label3"),
        lty=c(1,2,3), ncol=2)
```

Note that you don't need to use the `legend` function to add a legend to a bar chart, as the `barplot` function has a built-in legend option (see Section 8.5).

## 9.10 Multiple plots in the plotting area

As well as creating images of single plots, R allows you to create an image composed of several smaller plots arranged in a grid formation.

To set up the graphics device to display multiple plots, use the command:

```
> par(mfrow=c(R,C))
```

where `R` and `C` are the number of row and columns in the grid. Any plots that you subsequently create will fill the slots in the top row from left to right, followed by those in the second row and so on.

For example, to arrange four plots in a two-by-two grid use the command:

```
> par(mfrow=c(2,2))
```

Then you can create up to four plots of any type to fill each of the slots, as shown in Figure 9.9.

```
> hist(iris$Sepal.Length)
> qqnorm(iris$Sepal.Length)
> pie(summary(iris$Species))
> plot(Petal.Length~Sepal.Length, iris)
```

When you create a fifth plot, a new image is started. The graphics device will continue to display multiple plots for the remainder of the session, or until you reverse it with the command:

```
> par(mfrow=c(1,1))
```

## 9.11 Changing the default plot settings

So far, this chapter has explained how you can make modifications to a specific plot. However, you may want to change the settings so that they apply to all plots.

The `par` function allows you to change the default plotting style. For example, to change the default plotting symbol to be a triangle and the default title colour to red, use the command:

```
> par(pch=2, col.main="red")
```

Any plots that you subsequently create will have red title text and use a triangle as the plotting symbol (if applicable). These changes are applied for the remainder of the session, or until they are overwritten.

R allows almost every aspect of the plots to be customised. Enter the command `help(par)` to see a full list of settings. Some arguments that you cannot change with the `par` function include `main`, `sub`, `xlab`, `ylab`, `xlim` and `ylim`, which apply to individual plots only.

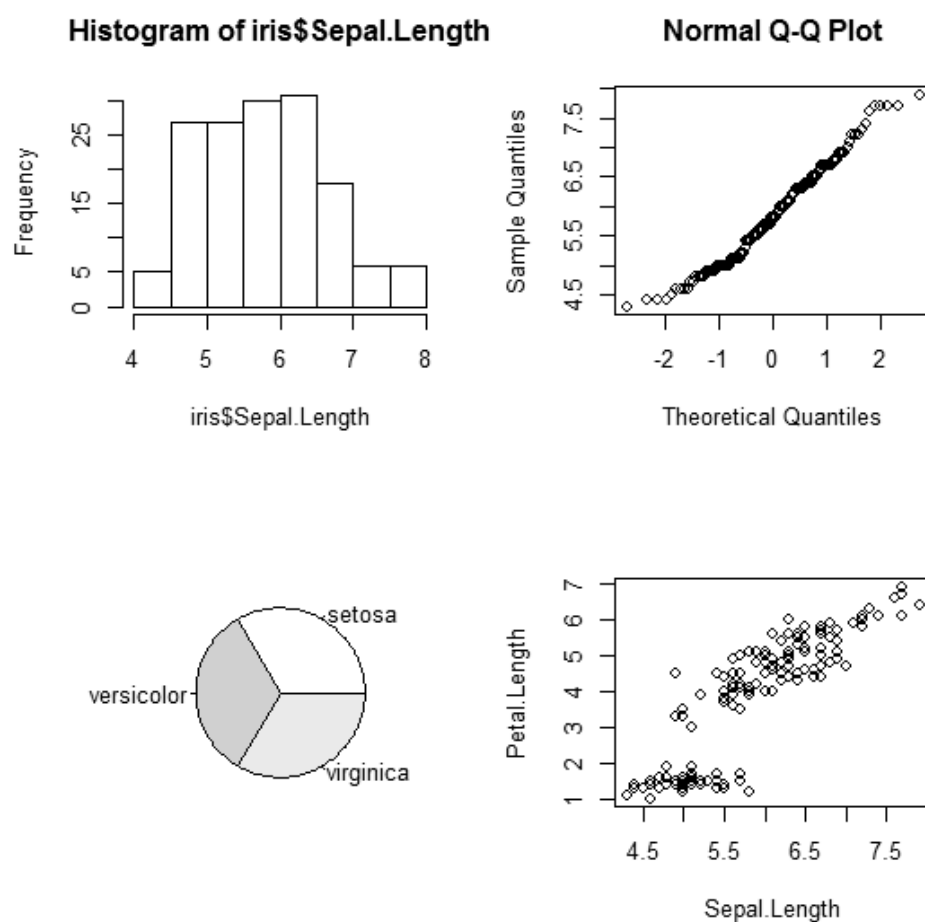


Figure 9.9: Multiple plots

## Chapter Summary: Customising your plots

Task	Argument or command
Add title	<code>main="Title Text"</code>
Add subtitle	<code>sub="Subtitle text"</code>
Add axis labels	<code>xlab="X axis label", ylab="Y axis label"</code>
Change axis limits	<code>xlim=c(xmin, xmax), ylim=c(ymin, ymax)</code>
Change plotting colour	<code>col="red"</code>
Change plotting symbol	<code>pch=2</code>
Change plotting symbol size	<code>cex=2</code>
Change line type	<code>lty=2</code>
Change line width	<code>lwd=2</code>
Change shading density	<code>density=20</code>
Add vertical line to plot	<code>abline(v=2)</code>
Add horizontal line to plot	<code>abline(h=2)</code>
Add straight line to plot	<code>abline(a=intercept, b=slope)</code>
Add line segment to plot	<code>segments(x1,y1,x2,y2)</code>
Add curve to plot	<code>curve(x^3)</code>
Add text to plot	<code>text(x, y, "Text String")</code>
Add grid to plot	<code>grid()</code>
Add arrow to plot	<code>arrows(x1, y1, x2, y2)</code>
Add points to plot	<code>points(dataset\$variable)</code>
Add lines to plot	<code>lines(dataset\$variable)</code>
Add legend to plot	<code>legend(x, y, legend=c("label1", "label2", "label3"), ...)</code>
Display multiple plots	<code>par(mfrow=c(rows, columns))</code>
Change default plot settings	<code>par(...)</code>



## Chapter 10

# Hypothesis tests

### Hypothesis tests

A hypothesis test uses a sample to test hypotheses about the population from which the sample is drawn. This helps you make decisions or draw conclusions about the population. A hypothesis test has the following components:

**Null hypothesis** (denoted  $H_0$ ) is a hypothesis about the population from which a sample or samples are drawn. It is usually a hypothesis about the value of an unknown parameter such as the population mean or variance, e.g.  $H_0$ : *The population mean is equal to five*. The null hypothesis is adopted unless proven false.<sup>15</sup>

**Alternative hypothesis** (denoted  $H_1$  or  $H_A$ ) is the hypothesis which will be accepted if there is enough evidence to reject the null hypothesis. This is generally the inverse of the null hypothesis, e.g.  $H_1$ : *The population mean is not equal to five*.<sup>15</sup>

**Test statistic** is a statistic calculated from the sample values, which has a known distribution under the null hypothesis. It varies depending on the type of test and has no direct interpretation.

**p-value** gives the probability of observing the test statistic or something more extreme, assuming that the null hypothesis were true. If this is very small then it suggests that the null hypothesis is not feasible, giving evidence in support of the alternative hypothesis.<sup>47</sup>

**Significance level** (denoted  $\alpha$ ) is the cut-off point at which the null hypothesis is rejected. The significance level should be determined before beginning the test. Usually a significance level of 0.05 is used, but 0.01 and 0.1 are also popular choices. If a significance level of 0.05 is used, then we reject the null hypothesis in favour of the alternative hypothesis only if the p-value is less than 0.05. Otherwise no conclusion is drawn. Choosing a significance level of 0.05 means that if the null hypothesis were true, there would be a 5% chance of incorrectly rejecting it (i.e. making a type I error).<sup>47</sup>

This chapter covers the topic of hypothesis testing, beginning with tests for comparing sample means. The t-test is covered Section 10.1 and the Wilcoxon rank-sum test (the non-parametric alternative) is covered in Section 10.2.

The analysis of variance, for comparing the means of three or more samples, is covered in Section 10.3. The Kruskal-Wallis test (the non-parametric alternative) is covered in Section 10.4.

Section 10.5 explains how to perform pairwise comparisons using multiple comparison methods such as Tukey's HSD test.

Finally, hypothesis tests for comparing sample variances are discussed in Section 10.6. These include the F-test and Bartlett's test.

There are also several hypothesis tests that are covered in other chapters. The hypothesis test for correlation and the Shapiro-Wilk and Kolmogorov-Smirnov tests for fit to a distribution are covered in Chapter 5. Tests for tabular data, such as the Chi-square test and Fisher's test, can be found in Chapter 6.

This chapter uses the `iris`, `PlantGrowth` and `sleep` datasets (included with R) and the `bottles`, `brains` and `grades1` datasets (available from the website). To view more details about the dataset included with R, enter `help(datasetname)`. For further details about all other datasets, see Appendix B.

## 10.1 Student's t-tests

### Student's t-test

The Student's t-test is used to test hypotheses about the mean value of a population or two populations. A t-test is suitable if the data is believed to be drawn from a normal distribution. If the samples are large enough (i.e. at least 30 values per sample<sup>11</sup>), then the t-test can be used even if the data is not normally distributed. If your data does not satisfy either of these criteria, then use the Wilcoxon rank-sum test instead (see Section 10.2).

There are three types of t-test:<sup>46</sup>

**One-sample t-test** is used to compare the mean value of a sample with a constant value denoted  $\mu_0$ . It has the null hypothesis that the population mean is equal to  $\mu_0$ , and the alternative hypothesis that it is not.

**Two-sample t-test** is used to compare the mean values of two independent samples, to determine whether they are drawn from populations with equal means. It has the null hypothesis that the two means are equal, and the alternative hypothesis that they are not equal.

**Paired t-test** is used to compare the mean values for two samples, where each value in one sample corresponds to a particular value in the other sample. It has the null hypothesis that the two means are equal, and the alternative hypothesis that they are not equal.

The t-test can also be performed with a one-sided alternative hypothesis, which is known as a one-tailed test. For a one-sample t-test, the one-sided alternative hypothesis is either that the mean is greater than  $\mu_0$  or that it is less than  $\mu_0$ . For a two-sample or paired t-test, the one-sided alternative hypothesis is that the mean of the first population is either greater or less than the mean of the second population.

### 10.1.1 One-sample t-test

You can perform a one-sample t-test with the `t.test` function. To compare a sample mean with a constant value `mu0`, use the command:

```
> t.test(dataset$sample1, mu=mu0)
```

The `mu` argument gives the value with which you to compare the sample mean. It is optional and has a default value of zero.

By default, R performs a two-tailed test. To perform a one-tailed test, set the `alternative` argument to "greater" or "less", as shown below.

```
> t.test(dataset$sample1, mu=mu0, alternative="greater")
```

A 95% confidence interval for the population mean is included with the output. To adjust the size of the interval, use the `conf.level` argument.

```
> t.test(dataset$sample1, mu=mu0, conf.level=0.99)
```

#### **Example 10.1.** One-tailed, one-sample t-test using the `bottles` data

A bottle filling machine is set to fill bottles with soft drink to a volume of 500 ml. The actual volume is known to follow a normal distribution. The manufacturer believes the machine is under-filling bottles. A sample of 20 bottles is taken and the volume of liquid inside is measured. The results are given in the `bottles` dataset, which is available from the website.

To calculate the sample mean, use the command:

```
> mean(bottles$Volume)
[1] 491.5705
```

Suppose you want to use a one-sample t-test to determine whether the bottles are being consistently under filled, or whether the low mean volume for the sample could be the result of random variation. A one-sided test is suitable because the manufacturer is specifically interested in knowing whether the volume is *less* than 500 ml. The test has the null hypothesis that the mean filling volume is

equal to 500 ml, and the alternative hypothesis that the mean filling volume is less than 500 ml. A significance level of 0.01 is to be used.

To perform the test, use the command:

```
> t.test(bottles$Volume, mu=500, alternative="less",  
        conf.level=0.99)
```

This gives the following output:

---

```
One Sample t-test  
  
data: bottles$Volume  
t = -1.5205, df = 19, p-value = 0.07243  
alternative hypothesis: true mean is less than 500  
99 percent confidence interval:  
 -Inf 505.6495  
sample estimates:  
mean of x  
 491.5705
```

---

From the output, we can see that the mean bottle volume for the sample is 491.6 ml. The one-sided 99% confidence interval tells us that mean filling volume is likely to be less than 505.6 ml. The p-value of 0.07243 tells us that if the mean filling volume of the machine were 500 ml, the probability of selecting a sample with a mean volume less than or equal to this one would be approximately 7%.

Since the p-value is not less than the significance level of 0.01, we cannot reject the null hypothesis that the mean filling volume is equal to 500 ml. This means that there is no evidence that the bottles are being under-filled.

### 10.1.2 Two-sample t-test

You can use the `t.test` function to perform a two-sample t-test using data in both stacked and unstacked forms. Your data is in stacked form if all the data values are stored in one variable, and a second variable gives the name or number of the sample that each observation belongs to. Your data is in unstacked form if the values for each sample are held in separate variables. If you are unsure which form your data is in, see Section 4.4 for some examples of data in stacked and unstacked forms.

To perform a two-sample t-test with data in stacked form, use the command:

```
> t.test(values~groups, dataset)
```

where *values* is the name of the variable containing the data values and *groups* is the variable containing the sample names. If the grouping variable has more than two levels then you must specify which two groups you want to compare, as shown below.

```
> t.test(values~groups, dataset, groups %in% c("Group1", "Group2"))
```

If your data is in unstacked form, use the command:

```
> t.test(dataset$sample1, dataset$sample2)
```

By default, R uses separate variance estimates when performing two-sample and paired t-tests. If you believe the variances for the two groups are equal, you can use the pooled variance estimate. An F-test or Bartlett's test (see Section 10.6) can help to determine whether this is the case. To use the pooled variance estimate, set the `var.equal` argument to `T` as shown.

```
> t.test(values~groups, dataset, var.equal=T)
```

To perform a one-tailed test, set the `alternative` argument to `"greater"` or `"less"`.

```
> t.test(values~groups, dataset, alternative="greater")
```

When the `alternative` argument is set to `"greater"`, the alternative hypothesis for the test is that the mean for the first group is greater than the mean for the second group. If you are using stacked data, you may need to use the `levels` function to check which are the first and second groups (see Section 3.6). Similarly, setting it to `"less"` gives an alternative hypothesis that the mean for the first group is less than the mean for the second group.

A 95% confidence interval for the difference in means is included with the output. You can adjust the size of this interval with the `conf.level` argument.

```
> t.test(values~groups, dataset, conf.level=0.99)
```

### Example 10.2. Two-sample t-test using the `iris` data

Suppose you wish to use a two-sample t-test to determine whether there is any real difference in mean sepal width for the *Versicolor* and *Virginica* species of iris. You can assume that sepal width is normally distributed, and that the variance for the two groups is equal. The null hypothesis for the test is that there is no real difference in mean sepal width for the two species, and the alternative hypothesis is that there is a difference.

The data is in stacked form, so perform the test with the command:

```
> t.test(Sepal.Width~Species, iris, Species %in% c("versicolor",  
  "virginica"), var.equal=T)
```

The output is shown below.

---

```
Two Sample t-test  
  
data: Sepal.Width by Species  
t = -3.2058, df = 98, p-value = 0.001819  
alternative hypothesis: true difference in means is not equal to 0  
95 percent confidence interval:
```

```

-0.33028246 -0.07771754
sample estimates:
mean in group versicolor   mean in group virginica
                2.770                2.974

```

---

The 95% confidence interval for the difference is -0.33 to -0.08, meaning that the mean sepal width for the Versicolor species is estimated to be between 0.08 and 0.33 cm less than for the Virginica species.

The p-value of 0.001819 is less than the significance level of 0.05, so we can reject the null hypothesis that the mean sepal width is the same for the Versicolor and Virginica species in favour of the alternative hypothesis that the mean sepal width is different for the two species.

This example is continued in Example 10.9 on page 142, where an F-test is used to check the assumption of equal variance.

### 10.1.3 Paired t-test

You can perform a paired t-test by setting the `paired` argument to `T`. If your data is in stacked form, use the command:

```
> t.test(values~groups, dataset, paired=T)
```

Your data must have the same numbers of observations in each group, so that there is a one-to-one correspondence between the samples. R matches the first value from the first sample with the first value from the second sample.

For data in unstacked form, use the command:

```
> t.test(dataset$sample1, dataset$sample2, paired=T)
```

As for the two-sample test, you can adjust the test using the `alternative`, `conf.level` and `var.equal` arguments. So to perform a test with the alternative hypothesis that the mean for the first group is less than the mean for the second group, and which uses a pooled variance estimate, use the command:

```
> t.test(values~groups, dataset, paired=T, alternative="less",
        var.equal=T)
```

#### Example 10.3. Paired t-test using the brains data

Consider the `brains` dataset shown in Figure 10.1, which gives brain volumes (in cubic centimetres) of the first and second-born twins for ten sets of monozygotic twins.

Suppose that you wish to use a t-test to determine whether brain volume is related to birth order. Brain volume is assumed to follow a normal distribution. The data is naturally paired because the

	Pair	Twin1	Twin2
1	1	1005	963
2	2	1035	1027
3	3	1281	1272
4	4	1051	1079
5	5	1034	1070
6	6	1079	1173
7	7	1104	1067
8	8	1439	1347
9	9	1029	1100
10	10	1160	1204

**Figure 10.1:** The `brains` dataset, giving brain volume data from the article *Brain Size, Head Size, and IQ in Monozygotic Twins*, by Tramo et al. See Appendix B for more details.

first twin from a birth corresponds to the second twin in the same birth, so a paired t-test is suitable. Since differences in either direction are of interest, a two-tailed test is used. The null hypothesis for the test is that there is no difference in mean brain volume for first and second-born twins, and the alternative hypothesis is that the mean brain volume for first-born twins is different to that of second-born twins.

To perform the test, use the command:

```
> t.test(brains$Twin1, brains$Twin2, paired=T)
```

The output is shown below.

---

```

Paired t-test

data:  brains$Twin1 and brains$Twin2
t = -0.4742, df = 9, p-value = 0.6466
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -49.04566  32.04566
sample estimates:
mean of the differences
          -8.5

```

---

The mean difference in brain size is estimated at -8.5, meaning that the brain size of the first born twins was an average of 8.5 cc less than that of their second born sibling. The confidence interval for the difference is -49 to 32 cc.

The p-value of 0.6466 tells us that if the mean brain size for first and second born twins is the same, the probability of observing a difference equal or greater to that in our sample is 65%. As the p-value is not less than the significance level of 0.05, the null hypothesis cannot be rejected. This means there is no evidence that brain size is related to birth order in twins.

## 10.2 Wilcoxon rank-sum test

### Wilcoxon rank-sum test

The Wilcoxon rank-sum test (also known as the Mann-Whitney U test) allows you test hypotheses about one or two sample means. It is a non-parametric alternative to the Student's t-test, which is suitable even when the distribution of the data is unknown and the samples are small.<sup>23,55</sup> In parallel with the t-test there are one-sample, two-sample and paired forms. The alternative hypothesis can be either two-sided or one-sided.

You can perform a Wilcoxon rank-sum test with the `wilcox.test` function. The commands are similar to those used to perform a t-test.

To perform a one-sample test, use the command:

```
> wilcox.test(dataset$sample1, mu=mu0)
```

To perform a two-sample test with data in stacked form, use the command:

```
> wilcox.test(values~groups, dataset)
```

If the grouping variable has more than two levels then you must specify which two you want to compare, as shown below.

```
> wilcox.test(values~groups, dataset, groups %in% c("Group1",  
  "Group2"))
```

If your data is in unstacked form (with the values for each sample held in separate variables), use the command:

```
> wilcox.test(dataset$sample1, dataset$sample2)
```

For a paired test, set the `paired` argument to `T`.

```
> wilcox.test(values~groups, dataset, paired=T)
```

To perform a one-tailed test, set the `alternative` argument to `"greater"` or `"less"`, as shown.

```
> wilcox.test(values~groups, dataset, alternative="greater")
```

By default, there is no confidence interval included with the output. To calculate a confidence interval for the population mean (for one-sample tests) or difference between means (for paired and two-sample tests), set the `conf.int` argument to `T`. The default size for the confidence intervals is 95%, but you can adjust this with the `conf.level` argument.



```
> wilcox.test(values~groups, dataset, conf.int=T, conf.level=0.99)
```

**Example 10.4.** Paired Wilcoxon rank-sum test using the sleep data

Consider the `sleep` dataset, which is included with R. The data gives the results of an experiment in which ten patients each took two different treatments and recorded the amount of additional sleep (compared to usual) that they experienced while receiving each of the treatments. The `extra` variable gives the increase in sleep (in hours per night), the `group` variable gives the treatment number (1 or 2) and the `ID` variable gives the patient number (1 to 10).

Suppose that you want to determine whether there is any real difference in the mean increase in sleep offered by the two treatments. A Wilcoxon rank-sum test is suitable because the distribution of additional sleep time is unknown, and the samples are small. A paired test is used because the additional sleep experienced by patient number  $x$  while taking drug 1 corresponds to the additional sleep experienced by patient number  $x$  while taking drug 2. The null hypothesis for the test is that there is no difference in mean additional sleep for the two treatments. The alternative hypothesis is that the mean additional sleep is different for the two treatments. A significance level of 0.05 will be used.

As the data is in stacked form, you can perform the test with the command:

```
> wilcox.test(extra~group, sleep, paired=T)
```

This gives the results:

---

```
Wilcoxon signed rank test with continuity correction

data:  extra by group
V = 0, p-value = 0.009091
alternative hypothesis: true location shift is not equal to 0

Warning messages:
1: In wilcox.test.default(x = c(0.7, -1.6, -0.2, -1.2, -0.1, 3.4, 3.7, :
   cannot compute exact p-value with ties
2: In wilcox.test.default(x = c(0.7, -1.6, -0.2, -1.2, -0.1, 3.4, 3.7, :
   cannot compute exact p-value with zeros
```

---

The p value of 0.009091 tells us that if the effect of both drugs were the same, there would be less than 1% chance of observing a difference in mean sleep increase as large as the one seen in this data. Since this is less than our significance level of 0.05, we can reject the null hypothesis of that the additional sleep is the same for the two treatments. This means that there is evidence of a difference in effectiveness between the two treatments.

R has given a warning that there are ties in the data. This means that some observations which have exactly the same value for the variable of interest, because the measurements have only been taken to one decimal place. For this reason, R is not able to calculate an exact p-value, and the results should be interpreted with caution. A more in-depth explanation of the effect of tied data on the Wilcoxon rank-sum test can be found on page 134 of *Nonparametric Statistics for the Behavioural Sciences*, by S. Siegel and N.J. Castellan.<sup>26</sup>

## 10.3 Analysis of variance

### Analysis of variance

An analysis of variance (or ANOVA) allows you to compare the means of three or more independent samples. It is suitable when the values are drawn from a normal distribution and when the variance is approximately the same in each group. You can check the assumption of equal variance with a Bartlett's test (see Section 10.6). The null hypothesis for the test is that the mean for all groups is the same, and the alternative hypothesis is that the mean is different for at least one pair of groups.<sup>14</sup>

More complex models such as the two-way analysis of variance or the analysis of covariance are covered in Chapter 11.

You can perform an analysis of variance with `aov` function. Since the ANOVA is a type of general linear model, you could also use the `lm` or `glm` functions as explained in Chapter 11. However the `aov` function presents the results more conveniently.

The command takes the form:

```
> aov(values~groups, dataset)
```

where `values` is the name of the variable that holds the data values and `groups` is the variable that identifies which sample each observation belongs to. If your data is in unstacked form (with the values for each sample held in separate variables) then you will need to stack the data beforehand, as explained in Section 4.4.

The results of the analysis consist of many components which R does not automatically display. If you save the results to an object as shown below, you can use further functions to extract the various elements of the output.

```
> aovobject<-aov(values~groups, dataset)
```

Once you have saved the results as an object, you can view the ANOVA table with the `anova` function.

```
> anova(aovobject)
```

To view the model coefficients, use the `coef` function.

```
> coef(aovobject)
```

To view confidence intervals for the coefficients, use the `confint` function.

```
> confint(aovobject)
```

**Example 10.5.** Analysis of variance using the `PlantGrowth` data

Consider the `PlantGrowth` dataset (included with R), which gives the dried weight of three groups of ten batches of plants, where each group of ten batches received a different treatment. The `weight` variable gives the weight of the batch and the `groups` variable gives the treatment received.

Suppose that you wish to perform an analysis of variance to determine whether there is any difference in plant growth (as measured by weight of batch) between the three groups. You can assume that plant growth is normally distributed, and that the variance is the same for all three treatments. A significance level of 0.05 is used.

To perform the ANOVA and save the results to an object named `plantanova`, use the command:

```
> plantanova<-aov(weight~group, PlantGrowth)
```

To view the ANOVA table, use the command:

```
> anova(plantanova)
```

This gives the following output:

---

Analysis of Variance Table

Response: weight

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
group	2	3.7663	1.8832	4.8461	0.01591 *
Residuals	27	10.4921	0.3886		

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

---

We can see that the p-value for the `group` effect is 0.01591. This means that if the effect of all three treatments were the same, we would have less than 2% chance of seeing differences between groups as large or larger than this. As the p-value is less than the significance level of 0.05, we can reject the null hypothesis that the mean growth is the same for all treatments, in favour of the alternative hypothesis that the mean growth is different for at least one pair of treatments.

To see the size of the treatment effects, use the command:

```
> coef(plantanova)
```

---

(Intercept)	grouptrt1	grouptrt2
5.032	-0.371	0.494

---

The output tells us that the control treatment gives an average weight of 5.032. The effect of treatment 1 (`trt1`) is to reduce weight by an average of -0.371 units compared to the control method, and the effect of treatment 2 (`trt2`) is to increase weight by an average of 0.494 units compared to the control method.

This example is continued in Example 10.7, where pairwise t-tests are used to further investigate the treatment differences, and in Example 10.10, where a Bartlett's test is used to check the assumption of equal variance.

## 10.4 Kruskal-Wallis test

### Kruskal-Wallis test

The Kruskal-Wallis test allows you to compare the mean values of three or more samples. It is a non-parametric alternative to the analysis of variance, which can be used when the distribution of the values is unknown.<sup>9,52</sup>

You can perform a Kruskal-Wallis test with the `kruskal.test` function. To perform the test with stacked data, use the command:

```
> kruskal.test(values~groups, dataset)
```

where the `values` variable contains the data values and the `groups` variable indicates which sample each observation belongs to.

For unstacked data (with samples in separate variables), nest the variables inside the `list` function as shown below.

```
> kruskal.test(list(dataset$sample1, dataset$sample2,
  dataset$sample3))
```

### Example 10.6. Kruskal-Wallis test using `grades1` data

Consider the `grades1` dataset, shown in Figure 4.5 on page 54. It gives the test results (out of 100) of 15 students belonging to three different classes.

Suppose you want to use a Kruskal-Wallis test to determine whether there are any differences in the effectiveness of the teaching methods used by each of the three classes, as measured by the mean test results of the students. A significance level of 0.05 is used.

As the data is in unstacked form, you can perform the test with the command:

```
> kruskal.test(list(grades1$ClassA, grades1$ClassB, grades1$ClassC))
```

This gives the following output:

```
Kruskal-Wallis rank sum test

data: list(grades1$ClassA, grades1$ClassB, grades1$ClassC)
Kruskal-Wallis chi-squared = 6.6796, df = 2, p-value = 0.03544
```

From the output you can see that the p-value is 0.03544. As this is less than the significance level of 0.05, we can reject the null hypothesis that the mean score is equal for all classes. This means that there is evidence of a difference in effectiveness between the teaching methods used by the three classes.

## 10.5 Multiple comparison methods

### Multiple comparison methods

After performing an analysis of variance and finding that there are some differences between group means, you may want to perform a series of pairwise t-tests to identify differences between specific pairs of groups. However, when performing a large number of t-tests, the overall probability of incorrectly rejecting the null hypothesis for at least one of the tests (the type I error) is greater than the significance level used for the individual tests. Multiple comparison methods allow you to perform pairwise t-tests on three or more samples, while controlling the overall type I error.<sup>12</sup>

There are a number of methods for performing multiple comparisons. Some of the most popular are the Tukey's honestly significant difference (HSD) test and the Bonferroni test.

### 10.5.1 Tukey's HSD test

You can perform a Tukey's HSD test with the `TukeyHSD` function. If you have previously performed an analysis of variance using the `aov`, `lm` or `glm` functions and saved the results to an object (as explained in Section 10.3), use the function directly.

```
> TukeyHSD(aovobject)
```

If you don't have an `aov` object and are using raw data, nest the `aov` function inside the `TukeyHSD` function, as shown below.

```
> TukeyHSD(aov(values~groups, dataset))
```

The default overall confidence level is 0.95, but you can adjust it with the `conf.level` argument.

```
> TukeyHSD(aovobject, conf.level=0.99)
```

#### **Example 10.7.** Tukey's HSD test using the `PlantGrowth` data

In Example 10.5, an analysis of variance was used to help determine whether there are any differences in mean plant growth (measured by weight of batch) between the three treatment groups. The conclusion was that there is a difference in plant growth for at least one pair of treatments.

Suppose you wish to continue this analysis by using pairwise t-tests to determine which treatment groups have differences in plant growth. An overall significance level of 0.05 is used.

If you still have the `plantanova` aov object created in Example 10.5, you can perform the test with the command:

```
> TukeyHSD(plantanova)
```

If you no longer have the model object, you can perform the test from the raw data as shown.

```
> TukeyHSD(aov(weight~group, PlantGrowth))
```

The output is shown below.

---

```
Tukey multiple comparisons of means
 95% family-wise confidence level

Fit: aov(formula = weight ~ group, data = PlantGrowth)

$group
      diff      lwr      upr    p adj
trt1-ctrl -0.371 -1.0622161 0.3202161 0.3908711
trt2-ctrl  0.494 -0.1972161 1.1852161 0.1979960
trt2-trt1  0.865  0.1737839 1.5562161 0.0120064
```

---

The `diff` column shows the difference in sample means for the two groups, while the `lwr` and `upr` columns give a 95% confidence interval for the difference. The `p adj` column gives the p-value for a t-test between the two groups, adjusting for multiple t-tests.

Comparing the p-values to our significance level of 0.05, we can see that the comparison `trt2-trt1` is statistically significant. This means that the plant growth for treatment 1 is significantly different from the growth for treatment 2. Treatment 1 and treatment 2 are not significantly different from the control.

Treatment 2 is estimated to give 0.865 units more growth than treatment 1. The 95% confidence interval for the difference in growth is 0.174 to 1.556.

### 10.5.2 Other pairwise t-tests

The `pairwise.t.test` function allows you to perform pairwise t-tests using a number of other multiple comparison methods. To perform pairwise t-tests using the Bonferroni adjustment, use the command:

```
> pairwise.t.test(dataset$values, dataset$groups,
  p.adj="bonferroni")
```

Other possible options for the `p.adj` argument include `holm` (the default method), `hochberg`, and `hommel`.<sup>30</sup> Enter `help(p.adjust)` to view a complete list.

**Example 10.8.** Pairwise comparisons with Bonferroni adjustment using the `PlantGrowth` data

Continuing the `PlantGrowth` example, suppose you wish to try performing pairwise t-tests using the Bonferroni adjustment for comparison.

```
> pairwise.t.test(PlantGrowth$weight, PlantGrowth$group,  
  p.adj="bonferroni")
```

The results are shown below.

---

```
      Pairwise comparisons using t tests with pooled SD  
  
data:  PlantGrowth$weight and PlantGrowth$group  
  
      ctrl  trt1  
trt1 0.583 -  
trt2 0.263 0.013  
  
P value adjustment method: bonferroni
```

---

The output shows p-values for each of the comparisons. From the output we can see that the comparison of treatment 1 (`trt1`) and treatment 2 (`trt2`) has a p-value of 0.013, which is statistically significant at the 0.05 level. The comparison between the the control and treatment 1, and between the control and treatment 2 were not statistically significant. This is consistent with the results of the Tukey's HSD test in the previous example.

### 10.5.3 Pairwise Wilcoxon rank-sum tests

There is also a function called `pairwise.wilcox.test` which allows you to perform pairwise Wilcoxon rank-sum tests. This is useful for identifying differences between individual pairs of groups after performing a Kruskal-Wallis test as described in Section 10.4.

To perform the test, use the command:

```
> pairwise.wilcox.test(dataset$values, dataset$groups)
```

As when performing pairwise t-tests, you can adjust the multiple comparison method with the `p.adj` argument.

## 10.6 Hypothesis tests for variance

### Hypothesis tests for variance

A hypothesis test for variance allows you to compare the variance of two or more samples to determine whether they are drawn from populations with equal variance. The tests have the null hypothesis that the variances are equal and the alternative hypothesis that they are not.<sup>20</sup> These tests are useful for checking the assumptions of a t-test or analysis of variance.

Two types of test for variance are covered in this section:

**F-test** allows you to compare the variance of two samples. It is suitable for normally distributed data.

**Bartlett's test** allows you to compare the variance of two or more samples. It is suitable for normally distributed data.<sup>20</sup>

### 10.6.1 F-test

You can perform an F-test with the `var.test` function. If your data is in stacked form, use the command:

```
> var.test(values~groups, dataset)
```

If the `groups` variable has more than two levels then you must specify which two you want to compare, as shown below.

```
> var.test(values~groups, dataset, groups %in% c("Group1", "Group2"))
```

For data in unstacked form (with the samples in separate variables), use the command:

```
> var.test(dataset$sample1, dataset$sample2)
```

#### Example 10.9. F-test using the `iris` dataset

In Example 10.2 we used a t-test to compare the mean sepal width for the *Versicolor* and *Virginica* species of iris, using a pooled variance estimate. One of the assumptions made for this test was that the variance in sepal width is the same for both species.

Suppose you want to use an F-test to help determine whether the variance in sepal width is the same for the two species. A significance level of 0.05 will be used.

To perform the test, use the command:



```
> var.test(Sepal.Width~Species, iris, Species %in% c("versicolor",
"virginica"))
```

The output is shown below.

---

```
      F test to compare two variances

data:  Sepal.Width by Species
F = 0.9468, num df = 49, denom df = 49, p-value = 0.849
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.5372773 1.6684117
sample estimates:
ratio of variances
      0.9467839
```

---

The p-value of 0.849 is not less than the significance level of 0.05, so we cannot reject the null hypothesis that the variance for the two groups is equal. There is no evidence to suggest that the variance in sepal width is different for the Versicolor and Virginica species.

### 10.6.2 Bartlett's test

You can perform the Bartlett's test with the `bartlett.test` function. If your data is in stacked form, use the command:

```
> bartlett.test(values~groups, dataset)
```

Unlike the F-test, the Bartlett's test allows you to compare the variance of more than two groups. However if required, you can still select a subset of groups to compare, as shown below.

```
> bartlett.test(values~groups, dataset, groups %in% c("Group1",
"Group2"))
```

If you have data in unstacked form (with the samples in separate variables), nest the variables inside the `list` function as shown.

```
> bartlett.test(list(dataset$sample1, dataset$sample2,
dataset$sample3))
```

#### Example 10.10. Bartlett's test using the PlantGrowth data

In Example 10.5 we used an analysis of variance to compare the mean weight of plant batches for the three treatment groups. One of the assumptions made for this test was that the variance in weight is the same for all treatment groups.

Suppose you want to use a Bartlett's test to determine whether the variance in weight is the same for all treatment groups. A significance level of 0.05 will be used.

To perform the test, use the command:

```
> bartlett.test(weight~group, PlantGrowth)
```

This gives the output:

---

```
Bartlett test of homogeneity of variances  
  
data: weight by group  
Bartlett's K-squared = 2.8786, df = 2, p-value = 0.2371
```

---

From the output we can see that the p-value of 0.2371 is not less than the significance level of 0.05. This means we cannot reject the null hypothesis that the variance is the same for all treatment groups. This means that there is no evidence to suggest that the variance in plant growth is different for the three treatment groups.

## Chapter Summary: Hypothesis tests

Test type	Command
One-sample t-test	<code>t.test(dataset\$sample1, mu=mu0)</code>
Two-sample t-test	<code>t.test(values~groups, dataset)</code> <code>t.test(dataset\$sample1, dataset\$sample2)</code>
Paired t-test	<code>t.test(values~groups, dataset, paired=T)</code> <code>t.test(dataset\$sample1, dataset\$sample2, paired=T)</code>
One-sample Wilcoxon rank-sum test	<code>wilcox.test(dataset\$sample1, mu=mu0)</code>
Two-sample Wilcoxon rank-sum test	<code>wilcox.test(values~groups, dataset)</code> <code>wilcox.test(dataset\$sample1, dataset\$sample2)</code>
Paired Wilcoxon rank-sum test	<code>wilcox.test(values~groups, dataset, paired=T)</code> <code>wilcox.test(dataset\$sample1, dataset\$sample2, paired=T)</code>
Analysis of variance	<code>aov(values~groups, dataset)</code>
Kruskal-Wallis test	<code>kruskal.test(values~groups, dataset)</code> <code>kruskal.test(list(dataset\$sample1, dataset\$sample2, dataset\$sample3))</code>
Tukey's HSD test	<code>tukeyHSD(aovobject)</code> <code>tukeyHSD(aov(values~groups, dataset))</code>
Pairwise t-tests	<code>paired.t.test(dataset\$values, dataset\$groups, method="bonferroni")</code>

Continued...

Test type	Command
Pairwise Wilcoxon rank-sum tests	<code>paired.wilcox.test(dataset\$values, dataset\$groups)</code>
F-test	<code>var.test(values~groups, dataset)</code> <code>var.test(dataset\$sample1, dataset\$sample2)</code>
Bartlett's test	<code>bartlett.test(values~groups, dataset)</code> <code>bartlett.test(dataset\$sample1, dataset\$sample2)</code>

## Chapter 11

# Regression and general linear models

---

### General linear models

A general linear model is used to predict the value of a continuous variable (known as the *response* variable) from one or more explanatory variables. A general linear model takes the form:

$$y = \beta_0 + \beta_1x_1 + \beta_2x_2 + \dots + \beta_nx_n + \epsilon$$

where  $y$  is the response variable,  $x_i$  are the explanatory variables,  $\beta_i$  are coefficients to be estimated and  $\epsilon$  represents the random error.<sup>19</sup>

The explanatory variables can be either continuous or categorical, and they can include cross products, polynomials and transformations of other variables.

The random errors are assumed to be independent, to follow a normal distribution with a mean of zero, and to have the same variance for all values of the explanatory variables.<sup>13</sup>

Simple linear regression, multiple linear regression, polynomial regression, analysis of variance, two-way analysis of variance, analysis of covariance and experimental design models are all types of general linear model.

This chapter explains how to build a general linear model, interpret the results, check the validity of the model and use the model to make predictions for new data.

This chapter uses the `trees` dataset (included with R) and the `powerplant`, `concrete` and `people2` datasets (which are available from the website and described in Appendix B).

## 11.1 Building the model

You can build a regression model or general linear model with either the `lm` function or the `glm` function. When used for this purpose, these functions do the same thing and give similar output.

In this book we will use the `lm` function for building general linear models, but be aware that you may also see the `glm` function in use.

### 11.1.1 Simple linear regression

To build a simple linear regression model with an explanatory variable named `var1` and a response variable named `resp`, use the command:

```
> lm(resp~var1, dataset)
```

You don't need to specify an intercept term (or constant term) in your model because R includes one automatically. To build a model without an intercept term, use the command:

```
> lm(resp~-1+var1, dataset)
```

When you build a model with the `lm` function, R displays the coefficient estimates. However, there are more components to the output that are not displayed, such as summary statistics, residuals and fitted values. You can save the all of the output to an object, as shown below. Later in the chapter, you will learn how to access the various components of the model output.

```
> modelname<-lm(resp~var1, dataset)
```

**Example 11.1.** Simple linear regression using the `trees` data

In Section 5.3.2 we saw that there is a correlation between tree girth and tree volume. Suppose you want to build a simple linear regression model to predict a tree's volume from its girth. To build the model, use the command:

```
> lm(Volume~Girth, trees)
```

---

```
Call:
lm(formula = Volume ~ Girth, data = trees)
```

```
Coefficients:
(Intercept)      Girth
   -36.943       5.066
```

---

From the output you can see that the model formula is:

$$\text{Volume} = -36.943 + 5.066 \times \text{Girth}$$

To save the model output as an object, use the command:

```
> treemodel<-lm(Volume~Girth, trees)
```

The `treemodel` object will be used later in the chapter.

### 11.1.2 Multiple linear regression

To include several explanatory variables in a model, separate them with the plus sign as shown below.

```
> modelname<-lm(resp~var1+var2+var3, dataset)
```

**Example 11.2.** Multiple linear regression using the powerplant data

Consider the `powerplant` dataset, which is described in Appendix B and is available from the website. The `Output` variable gives the output of a gas electrical turbine in megawatts. The `Pressure` and `Temp` variables give temperature and pressure measurements inside the turbine.

To build a multiple linear regression model that predicts the output from the pressure and temperature, use the command:

```
> lm(Output~Pressure+Temp, powerplant)
```

---

Call:

```
lm(formula = Output ~ Pressure + Temp, data = powerplant)
```

Coefficients:

(Intercept)	Pressure	Temp
-32.8620	0.1858	-0.9916

---

From the output you can see that the model is:

$$\text{Volume} = -32.8620 + 0.1858 \times \text{Pressure} - 0.9916 \times \text{Temperature}$$

### 11.1.3 Interaction terms

To add an interaction term to a model, use a colon (:). For example, `var1:var2` denotes the interaction between the two variables `var1` and `var2`. The command below builds a model with terms for two variables and their interaction.

```
> modelname<-lm(resp~var1+var2+var1:var2, dataset)
```

You can also use a colon to express third-order and higher interactions.

```
> modelname<-lm(resp~var1+var2+var3+var1:var2+var1:var3
+var2:var3+var1:var2:var3, dataset)
```

As you can see, this notation becomes lengthy for models with many variables. R has two useful shorthand notations for interaction terms, which are the asterisk (\*) notation and the hat (^) notation.

Use the asterisk notation to include a group of variable and all their possible interactions. For example, the command:

```
> modelname<-lm(resp~var1*var2*var3, dataset)
```

builds a model with terms for the three variable main effects (*var1*, *var2*, *var3*), the three second-order interactions (*var1:var2*, *var1:var3*, *var2:var3*) and the third-order interaction (*var1:var2:var3*). This is equivalent to the previous very lengthy formula.

The hat notation includes a set of variable as well as all the possible interactions up to a given order. For example, to include all main effects and second-order interactions (but not the third-order interaction), use the command:

```
> modelname<-lm(resp~(var1+var2+var3)^2, dataset)
```

This is equivalent to the command below.

```
> modelname<-lm(resp~var1+var2+var3+var1:var2+var1:var3 +var1:var2,
  dataset)
```

### Example 11.3. Factorial experiment using the concrete data

Consider the `concrete` dataset, which is shown in Figure 11.1 and available from the website. It gives the results of an experiment to determine the effect of cement type (I or II), additive type (A or B) and additive dose (0.3%, 0.4% or 0.5%) on the density of concrete.

	Cement	Additive	Additive.Dose	Density
1	I	A	0.3	2.43
2	II	A	0.3	2.431
3	I	B	0.3	2.43
4	II	B	0.3	2.418
5	I	A	0.4	2.419
6	II	A	0.4	2.436
7	I	B	0.4	2.435
8	II	B	0.4	2.414
9	I	A	0.5	2.419
10	II	A	0.5	2.425
11	I	B	0.5	2.422
12	II	B	0.5	2.41

**Figure 11.1:** The `concrete` dataset. See Appendix B for more details.

To build a model to predict density that includes terms for all of the explanatory variables (cement type, additive type and additive dose) and their interactions (second and third-order), use the command:



```
> concmodel<-lm(Density~Cement*Additive*Additive.Dose, concrete)
```

The `concmodel` object will be used later in the chapter.

### 11.1.4 Polynomial terms

To build a polynomial regression model with terms for  $\text{var1}$ ,  $\text{var1}^2$  and  $\text{var1}^3$ , use the command:

```
> modelname<-lm(resp~var1+I(var1^2)+I(var1^3), dataset)
```

Notice that the terms  $\text{var1}^2$  and  $\text{var1}^3$  are nested inside an `I()`. This is because the symbols  $\wedge$ ,  $*$  and  $+$  have special meanings when used in a model formula, which can be confused with their usual arithmetic meanings of power, multiplication and addition. If you want to use these symbols for their usual arithmetic meanings, you must nest them inside the `I` function.

**Example 11.4.** Polynomial regression using the `trees` dataset

To build a model to predict tree volume that has terms for `girth` and  $\text{girth}^2$ , use the command:

```
> lm(Volume~Girth+I(Girth^2), trees)
```

---

Call:

```
lm(formula = Volume ~ Girth + I(Girth^2), data = trees)
```

Coefficients:

(Intercept)	Girth	I(Girth^2)
10.7863	-2.0921	0.2545

---

From the result you can see that the model formula is:

$$\text{Volume} = 10.7863 - 2.0921 \times \text{Girth} + 0.2545 \times \text{Girth}^2$$

To save the model as an object, use the command:

```
> polytrees<-lm(Volume~Girth+I(Girth^2), trees)
```

The `polytrees` object will be used later in the chapter.

### 11.1.5 Transformations

Simple variable transformations such as the log or square root transformations can be applied to the response or explanatory variables directly in the formula using the relevant function, as shown below.

```
> modelname<-lm(log(resp)~var1, dataset)
```

For transformations that use the asterisk, hat or plus symbols, nest them inside the `I` function as shown.

```
> modelname<-lm(I(resp^2)~var1, dataset)
```

### 11.1.6 The intercept term

There may be occasions when you want to build a model without an intercept term. To do this, add `-1` as a term in the model, as shown below. This tells R not to include an intercept term, which would otherwise be included automatically.

```
> modelname<-lm(resp~-1+var1+var2+var3, dataset)
```

You can also create a model which contains only the intercept term, known as the *null model*.

```
> modelname<-lm(resp~1, dataset)
```

### 11.1.7 Including factor variables

You can include categorical variables in your model in the same way as continuous variables. Before including a categorical variable, use the `class` function to check that it has the `factor` variable class as explained in Section 3.6.

When you include a factor variable in a model, R treats the first level of the factor as the reference level. So for a factor with  $n$  level, the model will have  $n - 1$  coefficients that express the effect of the remaining  $n - 1$  levels relative to the reference level.

To check which is the first level for a factor variable, use the `levels` function.

```
> levels(dataset$variable)
```

You can change the reference level for a factor variable with the `relevel` function:

```
> dataset$variable<-relevel(dataset$variable, "reflevel")
```

It is possible to change the way that R uses the coefficients to express the effect of the factor by changing the *contrasts* for the factor variable. Every factor variable has a set of contrasts associated with it, which you can view and change with the `contrasts` function. Enter `help(contrasts)` for more details on how to change the contrasts for a factor variable, and enter `help(contr.treatment)` for a list of contrast options.

**Example 11.5.** Model with factor variable, using the `people2` dataset

Suppose that you want to use the `people2` dataset to build a model to predict a person's height from their hand span and eye colour.

You can build the model with the command:

```
> lm(Height~Hand.Span+Eye.Colour, people2)
```

---

Call:

```
lm(formula = Height ~ Hand.Span + Eye.Colour, data = people2)
```

Coefficients:

(Intercept)	Hand.Span	Eye.ColourBrown	Eye.ColourGreen
82.8902	0.4456	-3.6233	-4.1924

---

From the output we can see that formula has two coefficients that express the effect of brown and green eyes on height relative to blue eyes. So for people with blue eyes, the model formula is:

$$\text{Height} = 82.8902 + 0.4456 \times \text{HandSpan}$$

The formula for people with brown eyes is:

$$\text{Height} = 82.8902 + 0.4456 \times \text{HandSpan} - 3.6233$$

The formula for people with green eyes is:

$$\text{Height} = 82.8902 + 0.4456 \times \text{HandSpan} - 4.1924$$

**11.1.8 Updating a model**

Once you have built a model, you may want to add or remove a term from the model to see how the new model compares with the previous one. The `update` function allows you build a new model by adding or removing terms from an existing model. This is useful when working with models that have many terms, as it means that you don't have to retype the entire model specification every time you add or remove a term.

Suppose that you have build a model and saved it to an object named `modell`, as shown below.

```
> modell<-lm(resp~var1+var2+var3+var4, dataset)
```

To create a new model named `model2` by adding an additional term to `model1` (such as the interaction `var1:var2`), use the command:

```
> model2<-update(model1, ~.+var1:var2)
```

Similarly you can remove a term from the model as shown.

```
> model2<-update(model1, ~.-var4)
```

To check that the new model has the formula you expect, use the `formula` function.

```
> formula(model2)
```

### Example 11.6. Updating the `concmode1` model

In Example 11.3, we build the `concmode1` model with the command:

```
> concmodel<-lm(Density~Cement*Additive*Additive.Dose, concrete)
```

To create a new model by removing the three-way interaction from the `concmode1` model, use the command:

```
> concmodel2<-update(concmode1, ~.-Cement:Additive:Additive.Dose)
```

Check the model formula for the new model with the command:

```
> formula(concmode12)
```

---

```
Density ~ Cement + Additive + Additive.Dose + Cement:Additive +  
Cement:Additive.Dose + Additive:Additive.Dose
```

---

## 11.1.9 Stepwise model selection procedures

### Stepwise model selection procedures

Stepwise model selection procedures are algorithms designed to simplify the process of finding a model that explains a large amount of variation while including as few terms as possible. They are useful when dealing with large models with many potential terms. Popular stepwise selection procedures include forward selection, backward elimination and general stepwise selection.<sup>45</sup>

The `step` function allows you to perform forward, backward and stepwise model selection. The function takes an `lm` or `glm` model object as input, which should be the full model from which you want to select a subset of terms.

Suppose that you have created a large model such as the one shown below, which includes a total of fifteen terms: four main effects; six second-order interactions; four third-order interactions and one fourth-order interaction.

```
> model1<-lm(resp~var1*var2*var3*var4, dataset)
```

Once you have create the model, perform the stepwise selection procedure with the command:

```
> model2<-step(model1)
```

By default, the `step` function uses the general stepwise selection method. To use the backward or forward methods, set the `direction` argument to "backward" or "forward" as shown.

```
> model2<-step(model1, direction="backward")
```

The newly created model object can be used in just the same way as any other model object. To see which terms have been kept in the new model, use the `formula` function.

```
> formula(model2)
```

### Example 11.7. Stepwise selection using the concrete data

In Example 11.3, we build the `concomod` model with the command:

```
> conmodel<-lm(Density~Cement*Additive*Additive.Dose, concrete)
```

To perform the stepwise selection procedure on this model, use the command:

```
> conmodel3<-step(conmodel)
```

To view the formula of the resulting model, use the command:

```
> formula(conmodel3)
```

---

```
Density ~ Cement + Additive + Additive.Dose + Cement:Additive
```

---

From the output you can see that the stepwise selection procedure has removed the three-way interaction and two of the two-way interaction terms from the model, leaving the main effects of cement type, additive type and additive dose, and the interaction between cement type and additive type.

## 11.2 Summarising the model

### Model summary statistics

**Coefficient of determination ( $R^2$ )** gives an indication of how well the model is likely to predict future observations. It measures the portion of the total variation in the data that the model is able to explain. It takes values between 0 and 1. A value close to 1 suggests that the model will give good predictions, while a value close to 0 suggests that the model will make poor predictions.<sup>39</sup>

**Adjusted R-squared** is similar to  $R^2$ , but makes an adjustment for the number of terms in the model.

**Significance test for model coefficients** tells you whether individual coefficient estimates are significantly different from 0, and hence whether the coefficients are contributing to the model. Consider removing coefficients with p-values greater than 0.05.<sup>17</sup>

**F-test** tells you whether the model is significantly better at predicting compared with using the overall mean value as a prediction. For good models, the p-value will be less than 0.05.<sup>16</sup>

To view some summary statistics for a model, use the `summary` function:

```
> summary(lmobject)
```

The `summary` function displays:

- the model formula
- quantiles for the residuals
- coefficient estimates with the standard error and a significance test for each
- the residual standard error and degrees of freedom
- the  $R^2$  (multiple and adjusted)
- an F-test for model fit.

Note that if you built your model with the `glm` function instead of the `lm` function, the output will be slightly different and will use the generalized linear model terminology.

To view an ANOVA table for the model, use the `anova` function.

```
> anova(lmobject)
```

**Example 11.8.** Model summary statistics for the `treemodel` model

In Example 11.1 we created a simple linear regression model to predict tree volume from tree girth and saved the output to an object named `treemodel`.

To view summary statistics for the model, use the command:

```
> summary(treemodel)
```

This gives the following output.

---

```
Call:
lm(formula = Volume ~ Girth, data = trees)

Residuals:
    Min       1Q   Median       3Q      Max
-8.065 -3.107  0.152  3.495  9.587

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -36.9435     3.3651  -10.98 7.62e-12 ***
Girth         5.0659     0.2474   20.48 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.252 on 29 degrees of freedom
Multiple R-squared:  0.9353,    Adjusted R-squared:  0.9331
F-statistic: 419.4 on 1 and 29 DF,  p-value: < 2.2e-16
```

---

The  $R^2$  value of 0.9353 tells us that the model explains approximately 94% of the variation in tree volume. This suggests the model would be very good at predicting tree volume.

The hypothesis tests for the model coefficients tell us that the intercept and girth coefficients are significantly different from zero.

The p-value for the F-test is less than 0.05, which tells us that the model explains a significant amount of the variation in tree volume.

To view the ANOVA table, use the command:

```
> anova(treemodel)
```

---

```
Analysis of Variance Table

Response: Volume
      Df Sum Sq Mean Sq F value    Pr(>F)
Girth   1  7581.8   7581.8   419.36 < 2.2e-16 ***
Residuals 29   524.3     18.1
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

---

## 11.3 Coefficient estimates

Coefficient estimates (or parameter estimates) are provided with the summary output, but you can also view them with the `coef` function, or the identical `coefficients` function.

```
> coef(lmobject)
```

To view confidence intervals for the coefficient estimates, use the `confint` function.

```
> confint(lmobject)
```

The function gives 95% intervals by default, but you can adjust this with the `level` argument.

```
> confint(lmobject, level=0.99)
```

**Example 11.9.** Coefficients estimates for the `treemodel` model

To view coefficient estimates for the `treemodel`, use the command:

```
> coef(treemodel)
```

---

(Intercept)	Girth
-36.943459	5.065856

---

To view confidence intervals for the coefficient estimates, use the command:

```
> confint(treemodel)
```

---

	2.5 %	97.5 %
(Intercept)	-43.825953	-30.060965
Girth	4.559914	5.571799

---

From the output you can see that the 95% confidence interval for the intercept is -43.83 to -30.06, and the 95% confidence interval for the `Girth` coefficient is 4.56 to 5.57.

## 11.4 Plotting the line of best fit

For simple linear regression models, you can create a scatter plot with a line of best fit superimposed over the data to help visualise the model. Use the `plot` and `abline` functions as shown.

```
> plot(resp~var1, dataset)
> abline(coef(lmobject))
```



See Section 8.7 for more details about creating scatter plots with the `plot` function, and Section 9.7.1 for more about adding straight lines with the `abline` function.

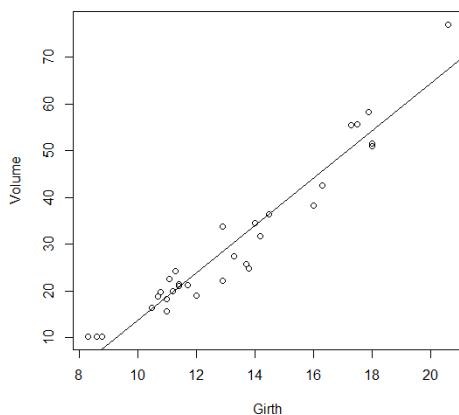
For polynomial regression models, you can superimpose a polynomial curve over the data. Use the `curve` function, as explained in Section 8.10.

**Example 11.10.** Scatter plot with line of best fit for the `treemodel` model

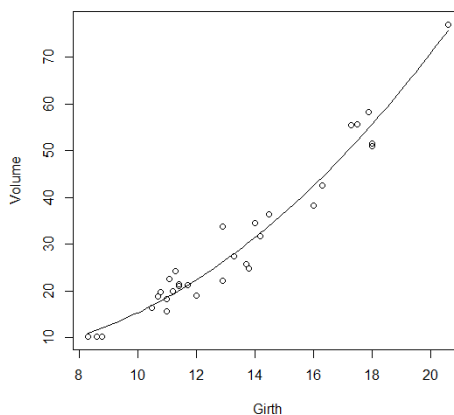
To create a scatter plot with line of best fit for the `treemodel`, use the commands:

```
> plot (Volume~Girth, trees)
> abline(coef(treemodel))
```

The result is shown in Figure 11.2a.



(a) Simple linear regression



(b) Polynomial regression

**Figure 11.2:** Scatter plots with models superimposed

**Example 11.11.** Scatter plot with polynomial curve for the `polytrees` model

In Example 11.4 we created a polynomial regression model named `polytrees` to predict tree volume from tree girth. To view the model coefficients for the `polytrees` model, use the `coef` function.

```
> coef(polytrees)
```

---

(Intercept)	Girth	I(Girth^2)
10.7862655	-2.0921396	0.2545376

---

To create a scatter plot with curve superimposed, use the `curve` function as shown.

```
> plot(Volume~Girth, trees)
> curve(10.7863-2.0921*x+0.2545*x^2, add=T)
```

Figure 11.2b shows the result.

## 11.5 Model diagnostics

This section discusses the methods used to check the suitability of the model for the data and the reliability of the coefficient estimates. These include examining the model residuals and the influence and Cook's distances for each of the observations.

### 11.5.1 Residual analysis

#### Residuals

The residuals for a given model are the set of differences between the observed values of the response variable, and the values predicted by the model (the fitted values). Standardised residuals and studentised residuals are types of residuals that have been adjusted to have a variance of one.<sup>4</sup>

Examining the set of residuals for a model helps you to determine whether the model is appropriate. If the assumptions of a general linear model are met, then the residuals will be normally distributed and have constant variance. They will also be independent and will not follow any observable patterns. Residuals also help you to identify any observations that heavily influence the model.<sup>1</sup>

To calculate raw residuals for a model, use the `residuals` function. There is also an identical function called `resid`. To calculate standardised residuals, use the `rstandard` function and for studentised residuals, use the `rstudent` function.

```
> residuals(lmobject)
```

You can save the residuals to a new variable in your dataset, as shown below. This is useful if you want to plot the residuals against the response or explanatory variables.

```
> dataset$resids<-rstudent(lmobject)
```

Similarly, you can create a new variable containing fitted values with the `fitted` function.

```
> dataset$fittedvals<-fitted(lmobject)
```

To can use the `plot` function to create residual plots for a model object.

```
> plot(lmobject, which=1)
```

Use the `which` argument to select from the following plots:

1. Residuals against fitted values
2. Normal probability plot of residuals
3. Scale-location Plot
5. Residuals against leverage

The numbers 4 and 6 select plots of influence measures, which you will learn about in the next section.

Some other useful plots of residuals include a histogram of residuals, for checking the assumption of normality:

```
> hist(dataset$resids)
```

A plot of residuals against the response variable:

```
> plot(resids~resp, dataset)
```

A plot of the residuals against the explanatory variable:

```
> plot(resids~var1, dataset)
```

### **Example 11.12.** Residual plots for the `treemodel` model

Suppose that you wish to analyse the residuals for the `treemodel` model, to check that the assumptions of the model are met.

First, save the studentised residuals to the `trees` dataset as shown.

```
> trees$Resids<-rstudent(treemodel)
```

Next, set up the graphics device to display six plots, as explained in Section 9.10.

```
> par(mfrow=c(3,2))
```

Next use the relevant commands to create the following plots: histogram of residuals; normal probability plot of the residuals; residuals against fitted values; residuals against response (`Volume`); residuals against explanatory variable (`Girth`); residuals against other variable of interest (`Height`).

```
> hist(trees$Resids)
> plot(treemodel, which=2)
> plot(treemodel, which=1)
> plot(Resids~Volume, trees)
> plot(Resids~Girth, trees)
> plot(Resids~Height, trees)
```

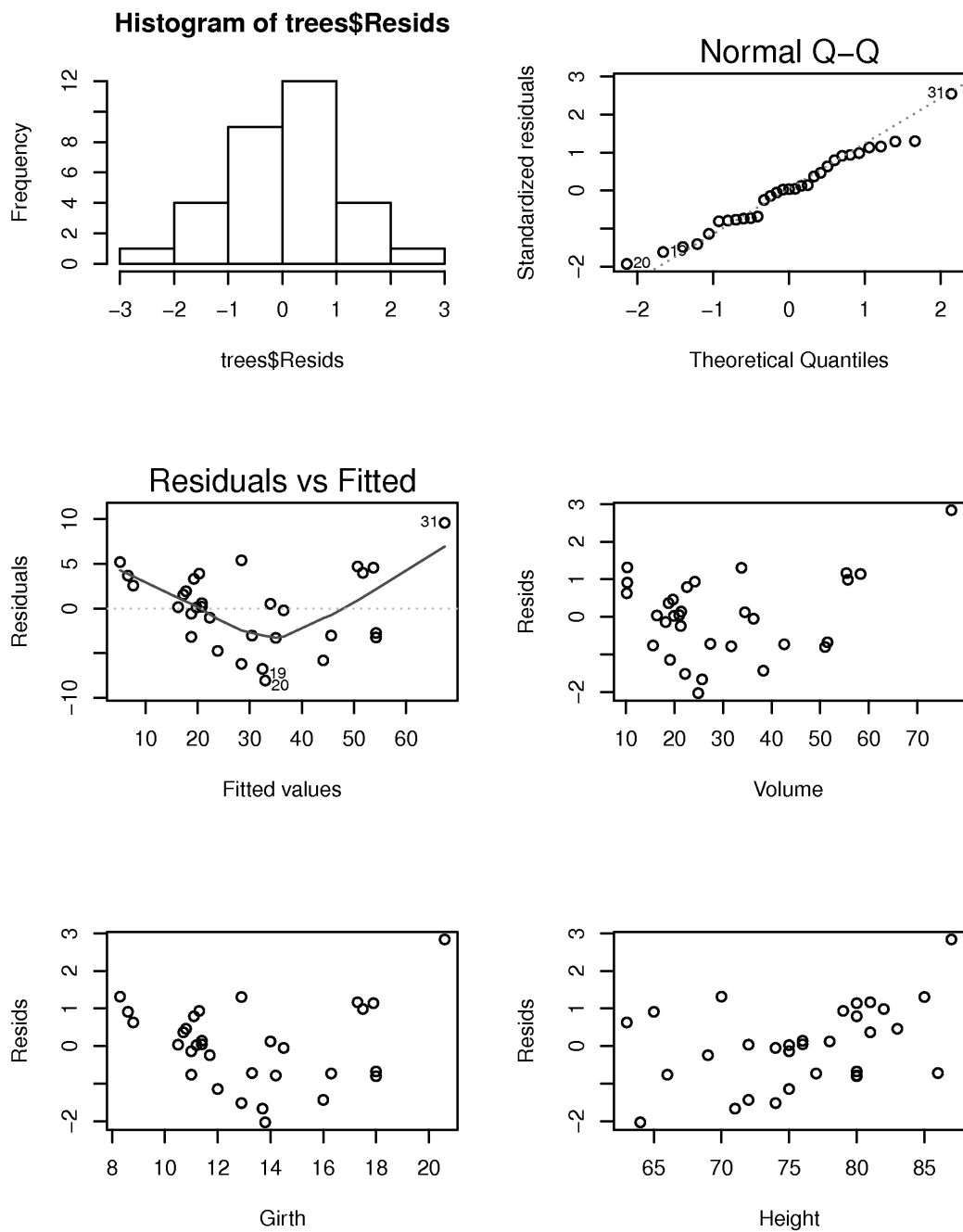
**Figure 11.3:** Residual plots for the `trees` model

Figure 11.3 shows the result.

From the histogram and normal probability plot, we can see that the residuals are approximately normally distributed.

In the plot of residuals against fitted values and the plot of residuals against girth, we can see that there is a slight pattern in the residuals. The residuals tend to be negative for trees with medium girth, and positive for trees with very small or very large girth. This suggests that adding polynomial terms to the model may improve the fit.

There are no obvious patterns in the plots of residual against volume and height. There are also no obvious outliers in any of the plots.

### 11.5.2 Leverage

#### Leverage

The leverage helps to identify observations that have outlying values or unusual combinations for the explanatory variables. A large leverage value indicates that the observation may have a big influence on the model.<sup>8</sup>

To calculate the leverage of each observation for a given model, use the `hatvalues` function.

```
> hatvalues(lmobject)
```

To create a plot of the residuals against the leverage, use the command:

```
> plot(lmobject, which=5)
```

To create a plot of the Cook's distances against the leverage, use the command:

```
> plot(lmobject, which=6)
```

### 11.5.3 Cook's distances

#### Cook's distances

The Cook's distance for an observation is a measure of how much the model parameters change if the observation is removed before estimation. Large values indicate that the observation has a big influence on the model.<sup>24</sup>

To calculate the Cook's distances for a model, use the `cooks.distance` function.

```
> cooks.distance(lmobject)
```

You can create a plot of Cook's distance against observation number with the command:

```
> plot(lmobject, which=4)
```

and a plot of Cook's distance against leverage with the command:

```
> plot(lmobject, which=6)
```

**Example 11.13.** Leverage and Cook's distances for the `treemodel`

Suppose you want to create a set of plots for the `treemodel` regression model, to help identify any observations which may have a large influence on the model.

First set up the graphics device to hold four plots, as explained in Section 9.10.

```
> par(mfrow=c(2,2))
```

Then create the three plots showing leverage and Cook's distances, as shown below.

```
> plot(treemodel, which=4)
> plot(treemodel, which=5)
> plot(treemodel, which=6)
```

The result is shown in Figure 11.4. The plots suggest that observation number 31 has a large influence on the model.

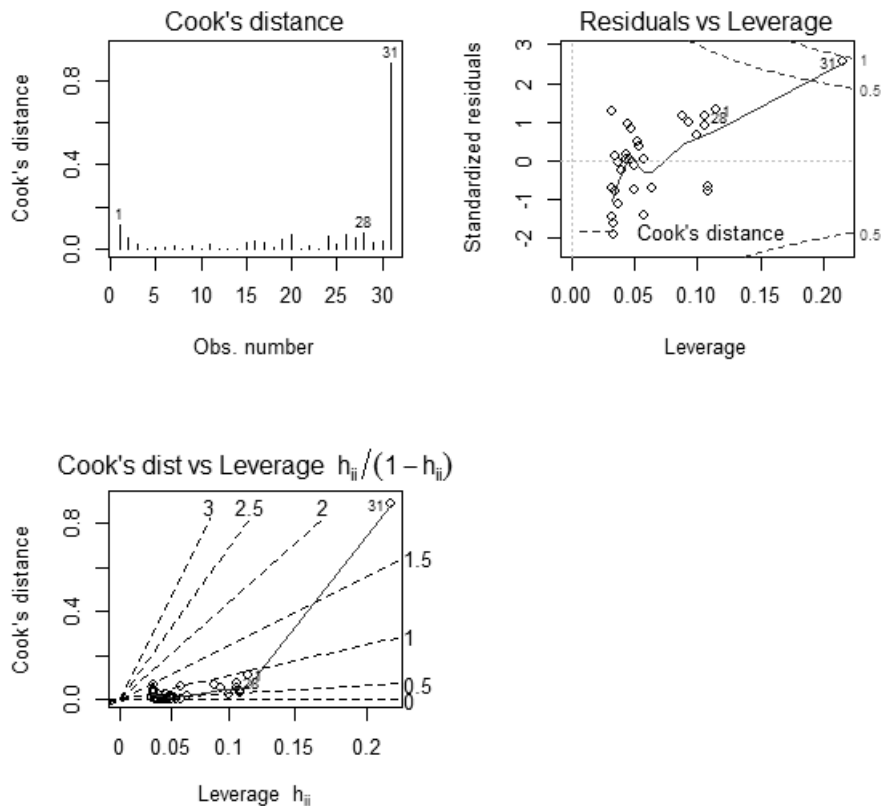
## 11.6 Making predictions

Once you have built a model that you are happy with, you may want to use it to make predictions for new data. R has a convenient function for making predictions, called `predict`. To use this function, the new data must be arranged in a data frame (see Section 2.1 for how to create data frames). The explanatory variables in the new dataset should be given identical names to those in the original dataset from which the model was built. It does not matter if the order of the variables is different, or if there are additional variables present.

Once your new data is arranged in a data frame, you can use the `predict` function as shown.

```
> predict(lmobject, newdata)
```

The command creates a vector of predictions which correspond to the rows of the data frame. You can attach it to the data frame as a new variable, as shown below:



**Figure 11.4:** Influence plots for `treemodel` model

```
> newdata$predictedvalues<-predict(lmobject, newdata)
```

You can also use the `predict` function to calculate confidence or prediction intervals for your predictions. Recall that confidence intervals only account for the uncertainty of the model estimation, while prediction interval also account for natural random variation in the response variable.

To calculate a confidence interval, set the `interval` argument to "confidence" and for a prediction interval, set it to "prediction". You can adjust the size of the interval with the `level` argument.

```
> predict(lmobject, newdata, interval="confidence", level=0.99)
```

**Example 11.14.** Making predictions using `treemodel`

Suppose you want to use the `treemodel` regression model to estimate the volume of three trees with girths of 17.2, 12.0 and 11.4 inches.

First put the new data into a data frame, as shown below.

```
> newtrees<-data.frame(Girth=c(17.2, 12.0, 11.4))
```

To make the predictions and add them to the `newtrees` data frame, use the command:

```
> newtrees$predictions<-predict(treemodel, newtrees,
  interval="prediction")
```

Then view the contents of the dataset.

```
> newtrees
```

---

	Girth	prediction.fit	prediction.lwr	prediction.upr
1	17.2	50.18927	41.13045	59.24809
2	12.0	23.84682	14.98883	32.70481
3	11.4	20.80730	11.92251	29.69210

---

From the output you can see that for a tree with a girth of 17.2 inches, the predicted volume is 50.2 cubic feet with a prediction interval of 41.1 to 59.2.



### Chapter Summary: General linear models

Task	Command
Build simple linear regression model	<code>lm(<i>resp~var1</i>, <i>dataset</i>)</code>
Build multiple regression model	<code>lm(<i>resp~var1+var2+var3</i>, <i>dataset</i>)</code>
Build model with interaction term	<code>lm(<i>resp~var1+var2+var1:var2</i>, <i>dataset</i>)</code>
Build model with interaction terms to a given order	<code>lm(<i>resp~(var1+var2+var3)^2</i>, <i>dataset</i>)</code>
Build factorial model	<code>lm(<i>resp~var1*var2*var3</i>, <i>dataset</i>)</code>
Build polynomial regression model	<code>lm(<i>resp~var1+I(var1^2)+I(var1^3)</i>, <i>dataset</i>)</code>
Build model with log-transformed response variable	<code>lm(log(<i>resp</i>)~<i>var1+var2+var3</i>, <i>dataset</i>)</code>
Build model without intercept term	<code>lm(<i>resp~-1+var1+var2+var3</i>, <i>dataset</i>)</code>
Build null model	<code>lm(<i>resp~1</i>, <i>dataset</i>)</code>
Update a model	<code>update(<i>lmobject</i>, ~.+<i>var4</i>)</code>
Stepwise selection	<code>step(<i>lmobject</i>)</code>
Summarise a model	<code>summary(<i>lmobject</i>)</code>
Coefficient estimates	<code>coef(<i>lmobject</i>)</code>
Confidence interval for coefficient estimate	<code>confint(<i>lmobject</i>)</code>
Plot line of best fit	<code>plot(<i>resp~var1</i>, <i>dataset</i>)</code> <code>abline(coef(<i>lmobject</i>))</code>
Raw residuals	<code>residuals(<i>lmobject</i>)</code>
Standardised residuals	<code>rstandard(<i>lmobject</i>)</code>
Studentised residuals	<code>rstudent(<i>lmobject</i>)</code>

**Chapter Summary: General linear models**

Task	Command
Fitted values	<code>fitted(lmobject)</code>
Residuals and influence plots	<code>plot(lmobject, which=1)</code>
Leverage	<code>hatvalues(lmobject)</code>
Cook's distances	<code>cooks.distance(lmobject)</code>
Predictions	<code>predict(lmobject, newdata)</code>

# Chapter 12

## Basic programming

---

As well being a piece of statistical software, R is also a programming language. The topic of R programming is beyond the scope of this book. However, this chapter gives a brief introduction to basic programming and writing your own functions.

When programming, use a script file so that you can edit your code easily. Remember to add plenty of comments with the hash symbol (`#`), so that your script can be understood by others or by yourself at a later date.

### 12.1 Creating new functions

To create a new function, the script takes the following general form:

---

```
functionname<-function(arg1name, arg2name, ..., argNname) {  
  
    command(s)  
  
    return(outputvalue)  
}
```

---

All of the indentation and additional blank lines are optional, but they help to show the hierarchy of the program and are considered good programming practise.

The first line determines the name of the function and the names of the input arguments. You can include as many arguments as required, or none at all. Optionally you can assign default values to the arguments, as shown below.

---

```
functionname<-function(arg1name=value1, arg2name=value2, ..., argNname=valueN) {  
  
    command(s)  
  
    return(outputvalue)  
}
```

---

This first line of the script ends with an opening curly bracket. After this is where the main content of the function begins. Generally this is a set of command that manipulate the input arguments in some way, or perform calculations from them, in order to create an output. The commands can make use of existing functions, including any that you have written yourself. Note that any objects that you create within a function exist only inside the function and are not saved to the workspace.

The `return` function determines the output of the function, which is either displayed in the console or assigned to an object whenever the function is used. It can either be a literal value such as a number or character string, or it can be a vector, data frame or any other type of object.

Instead of using the `return` function in the final command, you can use the `print` function, which always displays the output in the console window even if the output is assigned to an object. Alternatively you may want to create a function that produces a plot instead of giving an output value. In this case the final command would be `plot` or another plotting function.

Finally, the script ends with a closing curly bracket.

Once you have written your function and run the script, the function is saved to the workspace as an object and is available to use for as long as the current workspace is loaded. You can use it just like a built-in function.

```
> functionname(value1, value2, ..., valueN)
```

### Example 12.1. A function for cubing a value

The script below creates a simple function which takes a single value as input and calculates the cube root.

---

```
cube.root<-function(x) {  
  y<-x^(1/3)  
  return(y)  
}
```

---

Once you have written the function and run the script, the `cube.root` function is available to use as shown.

```
> cube.root(5)  
[1] 1.709976
```

### Example 12.2. A function for calculating the hypotenuse of a triangle

The script below creates a simple function which calculates the length of the hypotenuse of a triangle, given the lengths of the other two sides (recall that the formula is  $c^2 = a^2 + b^2$ ).

---

```
hypot<-function(a=2, b=3) {  
  c=sqrt(a^2+b^2)  
  return(c)  
}
```

---

Once the function is written and the script run, you can use it as shown.

```
> hypot(6, 7)  
[1] 9.219544
```

As the arguments have been given default values, the function still works if one of both of them are missing.

```
> hypot(b=2)  
[1] 2.828427
```

**Example 12.3.** A function that creates a histogram of random numbers

The script below creates a function that generates a specified amount of random numbers from a standard normal distribution and plots them in a histogram.

---

```
randhist<-function(n) {  
  vector1<-rnorm(n)      # Generate n random numbers  
  hist(vector1)           # Plot histogram  
}
```

---

Once the function is written and the script has been run, you can use it as shown.

```
> randhist(50)
```

## 12.2 Conditional statements

A *conditional statement* allows you to perform a command or set of commands only under certain circumstances (called the *condition*). Conditional statements add flexibility to your functions, as they allow the function to behave in different ways depending on the input.

There are a few types of conditional statement which allow you to do slightly different things. Before looking at these, you need to understand conditions and how they are constructed in R.

### 12.2.1 Conditions

In programming, a condition is an expression that can be either true or false. For example, the condition `'4<5'` (4 is less than 5) is true, whilst the condition `'5==8'` (5 is equal to 8) is false.

If you enter a condition at the command prompt, R tells you whether it is true or false:

```
> 6>8
[1] FALSE
```

A condition must contain at least one *comparison operator* (also known as a *relational operator*). In the example above, the comparison operator is `>` (greater than). Table 12.1 gives a list of comparison operators that you can use to form conditions.

Operator	Meaning
<code>==</code>	Equal to
<code>&lt;</code>	Less than
<code>&lt;=</code>	Less than or equal to
<code>&gt;</code>	Greater than
<code>&gt;=</code>	Greater than or equal to
<code>%in%</code>	In

**Table 12.1:** Comparison operators

The `%in%` operator compares a single value with all members of a vector, as shown below.

```
> vector1<-c(4,2,1,6)
> 2 %in% vector1
[1] TRUE
```

Conditions containing only constant values are not very useful because we already know in advance whether they are true or false. More useful are conditions that include objects, such as `'object1<5'` (the value of `object1` is less than 5). Whether or not this condition is true depends on the value of `object1`.

```
> object1<-4
> object1<5
[1] TRUE
```

You can join two or more conditions to form a larger one, using the OR and AND operators.

The AND operator is denoted `&`. When two expressions are joined with the AND operator, both must be true in order for the whole condition to be true. For example, the condition below is false because only one of the expressions is true.

```
> 3<5 & 7<5
[1] FALSE
```

The statement below is true because both of the expressions are true.

```
> 3<5 & 7>5
[1] TRUE
```

The OR operator is denoted `|`. When two expressions are joined with the OR operator, the overall condition is true if either one or both of the expressions are true. For example, the condition below is true because one of the expressions is true.

```
> 3<5 | 7<5
[1] TRUE
```

The condition is also true when both expressions are true.

```
> 3<5 | 7>5
[1] TRUE
```

You can negate a condition with the `!` operator. This reverses the result of the condition.

```
> !3<5
[1] FALSE
```

If the condition is complex, you can use brackets to negate the entire condition as shown below.

```
> !(3<5 & 7<5)
[1] TRUE
```

### 12.2.2 If statement

The simplest form of conditional statement is the *if* statement. The *if* statement consists of a condition and a command. When R runs an *if* statement, it first checks whether the condition is true or false. If the condition is true, it runs the command and if it is false it does not. The general form for the statement is shown below.<sup>49</sup>

---

```
if (condition) command
```

---

You can also include a group of several commands in an *if* statement, by placing them between curly brackets as shown.

---

```
if (condition) {
  commands to be performed if condition is true
}
```

---

The *if* statement is very useful as part of a function, as illustrated in the following example.

**Example 12.4.** A function for calculating body mass index

A person's body mass index (BMI) is calculated from his or her height in metres and weight in kilograms using the formula:<sup>21</sup>

$$\text{BMI} = \frac{\text{Weight}}{\text{Height}^2}$$

If imperial measurements are used (height in inches and weight in pounds), the formula is:

$$\text{BMI} = \frac{\text{Weight} \times 702}{\text{Height}^2}$$

The script below creates a function to calculate the BMI from a height and weight. By default the function calculates the BMI assuming that the metric measurements have been supplied. If the user sets the `units` argument to "imperial", the function makes the appropriate adjustment for imperial measurements. Notice the use of the *if* statement to control whether the adjustment is made.

---

```
bmi<-function(height, weight, units="metric") {  
  bmi=weight/height^2  
  if (units=="imperial") bmi<-bmi*702  
  return(bmi)  
}
```

---

Once the script has been run, you can use the function to calculate BMI using metric measurements:

```
> bmi(1.7, 70)  
[1] 24.22145
```

or imperial measurements:

```
> bmi(66, 125, "imperial")  
[1] 20.17332
```



### 12.2.3 If/else statement

The *if/else* statement extends the *if* statement to include a second command (or set of commands) to be performed if the condition is false. The general form is shown below.<sup>49</sup>

---

```
if (condition) command1 else command2
```

---

You can include groups of commands between curly brackets as shown.

---

```
if (condition) {  
  commands to be performed if condition is true  
} else {  
  commands to be performed if condition is false  
}
```

---

You can also extend the *if/else* statement to accommodate three or more possible outcomes, as shown below.

---

```
if (condition1) command1 else if (condition2) command2 else command3
```

---

When running the statement, R begins by checking whether the first condition is true or false. If it is true, then the first command is run. If the first condition is false, then R proceeds to check the second condition. If the second condition is true then the second command is run. Otherwise, the final command is run.

**Example 12.5.** A function for classifying dates

The script below creates a function for classifying a date (given in the format *ddmmmyyyy*) as a weekend or a weekday.

---

```
day.type<-function(date) {  
  date1<-as.Date(date, "%d%b%Y")  
  if (weekdays(date1) %in% c("Saturday", "Sunday")) return ("Weekend") else  
    return("Weekday")  
}
```

---

The function gives a different output depending on whether the input date is a weekend or a weekday.

```
> day.type("30AUG2012")  
[1] "Weekday"
```

```
> day.type("01SEP2012")  
[1] "Weekend"
```

**Example 12.6.** A function for classifying heights

The script below creates a function that takes a height in centimetres as input, and gives a height category as output. Heights below 140 cm are classified as 'Short', heights between 140 cm and 180 cm as 'Medium' and heights over 180 cm as 'Tall'.

```
heightcat<-function(height) {  
  if (height<140) return("Short") else if (height<180) return("Medium") else  
    return("Tall")  
}
```

Once you have run the script, you can use the function as shown.

```
> heightcat(136)  
[1] "Short"  
  
> heightcat(187)  
[1] "Tall"
```

### 12.2.4 The switch function

In some circumstances, you can use the `switch` function as a compact alternative to using *if/else* statements with many possible outcomes. The `switch` function selects between a list of alternative commands, each of which must return a single value. R compares the input with a list of options, and if it finds a match then it performs the corresponding command. The final command (which is optional) is performed if the input does not match any of the options.<sup>37</sup>

```
> switch(input, option1=command1, option2=command2, option3=command3, command4)
```

There is another use of the `switch` function which takes an integer value as input, and outputs the corresponding value from a list of values.

```
> switch(input, value1, value2, value3, value4)
```

**Example 12.7.** Function to perform a selected calculation with two numbers

The script below creates a function which allows the user to give two numbers and a calculation type. The `switch` function is used to select from several commands, depending on which option the user selects.

---

```
func1<-function(number1, number2, calctype="add") {  
  result<-switch(calctype,  
    "multiply"=number1*number2,  
    "divide"=number1/number2,  
    "add"=number1+number2,  
    "subtract"=number1-number2,  
    "exponent"=number1^number2,  
    "Invalid calculation type"  
  )  
  
  return(result)  
}
```

---

Once the script has been run, you can use the function as shown below.

```
> func1(2, 3, "divide")  
[1] 0.6666667  
  
> func1(2, 3, "mean")  
[1] "Invalid calculation type"
```

**Example 12.8.** Function for giving the name of the day of the week

The script below creates a function named `week.day` that takes a number from 1 to 7 as input, and returns a character string giving the corresponding day of the week.

---

```
week.day<-function(daynum) {  
  dayname<-switch(daynum,  
    "Monday",  
    "Tuesday",  
    "Wednesday",  
    "Thursday",  
    "Friday",  
    "Saturday",  
    "Sunday"  
  )  
  
  return(dayname)  
}
```

---

Once you have run the script, you can use it as shown below.

```
> week.day(2)  
[1] "Tuesday"
```

## 12.3 Loops

Loops allow you to repeat a command (or set of commands) a number of times. The two most important types of loop are the *for* loop and the *while* loop.

### 12.3.1 For loop

The *for* loop allows you want to repeat a command or set of commands a prespecified number of times. The *for* loop takes the following general form:<sup>49</sup>

---

```
for (i in startvalue:endvalue) command
```

---

You do not have to use *i* for the repetition number as shown above. Any valid object name can be used, however *i* is conventional. The *startvalue* and *endvalue* can be either constant values or object names.

You can also include a set of commands to be repeated by enclosing them within curly brackets, as shown below.

---

```
for (i in startvalue:endvalue) {  
    commands to be repeated  
}
```

---

#### Example 12.9. Function for calculating times tables

The script below creates a function that takes a single number as input, multiplies it by each of the numbers 1 to 10 and displays the results in the console window. The *for* loop is used to repeat the calculation for each of the numbers 1 to 10.

---

```
times.table<-function(x) {  
    for (i in 1:10) {  
        result=x*i  
        text=paste(x, "times", i, "equals", result)  
        print(text)  
    }  
}
```

---

Once the script has been run, you can use the function as shown.

```
> times.table(5)
```

---

```
[1] "5 times 1 equals 5"
[1] "5 times 2 equals 10"
[1] "5 times 3 equals 15"
[1] "5 times 4 equals 20"
[1] "5 times 5 equals 25"
[1] "5 times 6 equals 30"
[1] "5 times 7 equals 35"
[1] "5 times 8 equals 40"
[1] "5 times 9 equals 45"
[1] "5 times 10 equals 50"
```

---

### 12.3.2 While loop

The *while* loop is suitable when you want to repeat a command or set of commands until a given condition is satisfied, and you don't know in advance how many repetitions will be required in order to achieve this. The general form for the *while* loop is shown below.<sup>49</sup>

---

```
while (condition) command
```

---

To include a group of commands, use curly brackets as shown.

---

```
while (condition) {
  commands to be repeated
}
```

---

The commands within the loop should do something that will affect whether or not the condition is true, otherwise R will keep processing the commands infinitely. Consider the example below:

---

```
b<-2
while (b<3) a<-4
```

---

As the command within the loop is unrelated to the condition, the condition continues to be true each time the loop is repeated. This causes R to keep repeating the loop indefinitely and to stop responding. If you find R is stuck repeating a loop, press the escape key to cancel the commands.

#### **Example 12.10.** Function for simulating die rolls

The following script creates a function which simulates die rolls. The function keeps rolling imaginary dice until a six is rolled.

```
die.rolls<-function() {  
    roll<-0          # Create the variable before using it in the loop  
    while (roll!=6) {  
        roll<-sample(1:6, 1) # Generate a random number between 1 and 6  
        print(roll)  
    }  
}
```

---

Once the script is run, you can use the function as shown.

```
> die.rolls()  
[1] 1  
[1] 1  
[1] 5  
[1] 6  
  
> die.rolls()  
[1] 1  
[1] 3  
[1] 3  
[1] 1  
[1] 3  
[1] 6
```

**Chapter Summary: Basic programming**

Task	General form
Create a function	<pre>functionname&lt;-function(arguments) {   command(s)   return(output) }</pre>
<i>if</i> statement	<pre>if (condition) command</pre>
<i>if/else</i> statement	<pre>if (condition) command1 else command2</pre>
<i>if/else</i> statement (extended)	<pre>if (condition1) command1 else if   (condition2) command2 else command3</pre>
switch function	<pre>switch(input, value1, value2, valueN)  switch(input, option1=command1,   option2=command2, optionN=commandN)</pre>
<i>for</i> loop	<pre>for (i in startval:endval) command</pre>
<i>while</i> loop	<pre>while (condition) command</pre>





# Appendix A

## Add-on packages

---

Over three thousand add-on packages are available for R, which serve a wide variety of purposes. An add-on package contains additional functions and sometimes objects such as example datasets. This appendix explains how to find a package that serves your purpose and install it.

### Viewing a list of available add-on packages

To view a list of available add-on packages, follow the instructions below:

1. Go to the R project website at *www.r-project.org*.
2. Follow the link to 'CRAN' (on the left-hand side).
3. You will be taken to a list of sites that host the R installation files (mirror sites). Select a site close to your location.
4. Select 'Packages' from the menu on the left-hand side.
5. Select 'Table of available packages, sorted by name'.

A list of packages with descriptions of their purpose is displayed. You can use the browser tools to search the list, usually by entering Ctrl+F or Cmd+F.

On selecting a suitable package, a manual is available in pdf format. You will notice that the package is available to download, but you do not need to do this as it is simpler to install the package from within the R environment.

### Installing and loading add-on packages

To use an add-on package you must first install it, which only needs to be done once. There are a number of packages that are included with the R base installation, which do not need to be installed.

Once a package is installed, it must be loaded before you can use the functions within. The functions will be available for the remainder of the session, so you will need to load the package during each session that you intend to use it.

You can install and load packages from within the R environment, which is explained separately for Windows, Mac and Linux users.

## Windows users

To install a package:

1. Select 'Install package(s)' from the 'Package' menu.
2. The first time you install a package, you will be prompted to select a mirror site. Select a site close to your location.
3. When prompted, select the required package from the list.

To load a package:

1. Select 'Load package' from the 'Packages' menu.
2. When prompted, select the required package from the list.

## Mac users

To install a package:

1. Select 'R Package Installer' from the 'Packages & Data' menu.
2. Press 'Get List'
3. A list of packages is displayed. Select the required package and press 'Install Selected'.
4. Close the window.

To load a package:

1. Select 'R Package Manager' from the 'Packages & Data' menu.
2. Tick the status box next to the required package so that the status changes to 'loaded'.

## Linux users

To install a package:

1. Enter the command:

```
download.packages(packagename, "/home/Username/folder").
```

The file path gives the location in which to save the package.

2. When prompted, select a mirror site close to your location.

To load a package:

1. Enter the command:

```
install.packages("packagename")
```



# Appendix B

## Datasets

---

### apartments

<b>Description</b>	Details of 32 one-bedroom apartments advertised for rent within a 5 mile radius of Bishops Stortford, Hertfordshire, UK in October 2012.	
<b>Variables</b>	Town	Location of apartment.
	Furnished	Provided furnished (Yes or No)
	Price.Cat	Rental price category (per calendar month)
<b>Source</b>	www.rightmove.co.uk	

### bigcats

<b>Description</b>	Average weights for four big cat species	
<b>Variables</b>	Name	Name of species. Note that first instance of Leopard refers to <i>Pantera Pardus</i> and the second instance refers to <i>Unica unica</i> (Snow Leopard).
	Weight	Mean weight in kilograms for male of species
<b>Source</b>	dialspace.dial.pipex.com/agarman/facts1.htm	

### bottles

<b>Description</b>	Dataset giving the volume of liquid within thirty bottles of soft drink randomly selected from a production line. This is fictional data.	
<b>Variables</b>	Volume	Volume (millilitres)

**brains**

**Description** Brain volume for ten pairs of monozygotic twins, measured using magnetic resonance imaging and computer-based image analysis techniques. This data is taken from the article *Brain Size, Head Size, and IQ in Monozygotic Twins* by Tramo, M.J. et al and published by Neurology 1998; 50:1246-1252. Reproduced with permission.

**Variables**

<code>Pair</code>	Pair identifier
<code>Twin1</code>	Total brain volume of first-born twin (cubic centimetres)
<code>Twin2</code>	Total brain volume of second-born twin (cubic centimetres)

**Source** [lib.stat.cmu.edu/datasets/IQ\\_Brain\\_Size](http://lib.stat.cmu.edu/datasets/IQ_Brain_Size)

**CIAdata1, CIAdata2**

**Description** Statistical data for seven countries, collected from the CIA World Factbook on the 5th August 2012

**Variables**

<code>country</code>	Country name
<code>lifeExp</code>	Life expectancy (years)
<code>urban</code>	Living in urban areas (%)
<code>pcGDP</code>	Per capita gross domestic product (\$US)

**Source** [www.cia.gov/library/publications/the-world-factbook/](http://www.cia.gov/library/publications/the-world-factbook/)

**coffeeshop**

**Description** Total sales at a coffee shop over a five day period. This is fictional data.

**Variables**

<code>date</code>	Date in format dd/mmm/yyyy
<code>sales</code>	Sales for the day (£)

**concrete**

**Description** The results of an experiment to determine the best concrete mix. The experiment was conducted in Santiago, Chile in 2007.

<b>Variables</b>	Cement	Cement type (I or II)
	Additive	Additive (A or B)
	Additive.Dose	Additive dose (0.3%, 0.4% or 0.5%)
	Density	Density (grams per cubic centimetre)

**CPIdata**

**Description** Consumer price index data for six countries (2012).

<b>Variables</b>	country	Country name
	CPI	Consumer price index (relative to New York at 100)

**Source** [www.numbeo.com/cost-of-living/rankings\\_by\\_country.jsp](http://www.numbeo.com/cost-of-living/rankings_by_country.jsp)

**customers**

**Description** The names and addresses of five customers living in the area of Reading, Berkshire, UK. This is fictional data.

<b>Variables</b>	Name	Character string giving customer's full name
	Address	Character string giving customer's full address

**endangered**

**Description** Conservation status of four big cat species

**Variables**

Name	Name of species
Status	Conservation status

**Source** <http://www.bigcats.com/redlist.php>

**fiveyearreport**

**Description** UK Sales (including VAT) for the years 2007 to 2011 for the Tesco, Sainsburys and Morrisons supermarket chains.

**Variables**

Year	Year (2007-2011)
Tesco	UK sales including VAT (£M) for Tesco
Sainsburys	UK sales including VAT (£M) for Sainsburys
Morrisons	UK sales including VAT (£M) for Morrisons

**Source**

[www.tescopl.com/files/pdf/reports/tesco\\_annual\\_report\\_2011.pdf](http://www.tescopl.com/files/pdf/reports/tesco_annual_report_2011.pdf) (p106)  
[www.tescopl.com/files/pdf/reports/annual\\_report\\_2010.pdf](http://www.tescopl.com/files/pdf/reports/annual_report_2010.pdf) (p16)  
[www.tescopl.com/files/pdf/reports/annual\\_report\\_2009.pdf](http://www.tescopl.com/files/pdf/reports/annual_report_2009.pdf) (p34)  
[www.tescopl.com/files/pdf/reports/annual\\_report\\_2008.pdf](http://www.tescopl.com/files/pdf/reports/annual_report_2008.pdf) (p5)  
[www.tescopl.com/files/pdf/reports/annual\\_report\\_2007.pdf](http://www.tescopl.com/files/pdf/reports/annual_report_2007.pdf) (p3)  
[www.j-sainsbury.co.uk/about-us/financial-performance/5-year-summary/](http://www.j-sainsbury.co.uk/about-us/financial-performance/5-year-summary/)  
[www.morrisons.co.uk/corporate/2011/annualreport/investor-information/five-year-summary-results/](http://www.morrisons.co.uk/corporate/2011/annualreport/investor-information/five-year-summary-results/)



**flights**

**Description** Flight data for seven flights departing from Southampton Airport on the 12th and 13th of January 2012.

**Variables**

Date	Date of flight in format dd/mm/yyyy
Time	Time of flight in format hh:mm
Flight.number	Alphanumeric flight number
Destination	Name of destination city

**Source** [www.southamptonairport.com](http://www.southamptonairport.com)

**fruit**

**Description** Dataset of UK fruit prices at August 2012.

**Variables**

Product	Product name
Price	Sale price (£)
Unit	Sale unit

**Source** [www.sainsburys.co.uk](http://www.sainsburys.co.uk)

**grades1**

**Description** Fictional dataset giving the grades of fifteen students belonging to three classes labelled A, B and C.

**Variables**

ClassA	Grades of students in class A (%).
ClassB	Grades of students in class B (%).
ClassC	Grades of students in class C (%).

**people**

**Description** Physical characteristics for a sample of sixteen people. Sample selected using non-random methods. Data is self-reported.

<b>Variables</b>	<code>Subject</code>	Respondent number
	<code>Eye.colour</code>	Eye colour (Blue, Green or Brown)
	<code>Height</code>	Height (centimetres)
	<code>Hand.span</code>	Hand span of left hand (millimetres)
	<code>Handedness</code>	Handedness (L=left-handed, R=right-handed)
	<code>Sex</code>	Sex (1=Male, 2=Female)

**people2**

**Description** This dataset is a clean version of the `people` dataset.

<b>Variables</b>	<code>Subject</code>	Respondent number
	<code>Eye.colour</code>	Eye colour (Blue, Green, Brown)
	<code>Height</code>	Height (centimetres)
	<code>Hand.span</code>	Hand span of left hand (millimetres)
	<code>Handedness</code>	Handedness (Left or Right)
	<code>Sex</code>	Sex (Male or Female)
	<code>Height.cat</code>	Height category (Tall, Medium or Short)

**powerplant**

**Description** Thirty measurements of pressure, temperature and output collected from a gas electrical turbine at a UK power station in 2010.

<b>Variables</b>	<code>Pressure</code>	Pressure inside turbine (millibars)
	<code>Temperature</code>	Temperature inside turbine (degrees centigrade)
	<code>Output</code>	Output of turbine (megawatts)

**pulserates**

**Description** Pulse data for four people. Sample selected using non-random methods. Data is self-reported.

**Variables**

<code>patient</code>	Patient identifier
<code>pulse1</code>	First pulse reading (beats per minute)
<code>pulse2</code>	Second pulse reading (beats per minute)
<code>pulse3</code>	Third pulse reading (beats per minute)

**resistance**

**Description** Gives the results of a simple experiment to compare the cubic resistance of four concrete formulations, at three, seven and fourteen days after setting. Experiment conducted in Santiago, Chile in 2007.

**Variables**

<code>Formula</code>	A (Huechuraba Aggregate + Additive A) B (Huechuraba Aggregate + Additive B) C (Mauro Aggregate + Additive A) D (Mauro Aggregate + Additive B)
<code>Day3</code>	Cubic resistance (kilograms per square metre), measured three days after setting
<code>Day7</code>	Cubic resistance (kilograms per square metre), measured seven days after setting
<code>Day14</code>	Cubic resistance (kilograms per square metre), measured fourteen days after setting

**supermarkets**

**Description** Data for four UK supermarket chains. Data for number of stores and total sales area is collected from the respective 2011 annual reports and Market Share is collected from Kantar Worldpanel.

**Variables**

Chain	Name of chain
Stores	Number of stores in the UK
Sales.Area	Sales area (1,000 square feet)
Market.Share	Market share (%)

**Source**

[www.tescopl.com/media/417/tesco\\_annual\\_report\\_2011\\_final.pdf](http://www.tescopl.com/media/417/tesco_annual_report_2011_final.pdf)  
[www.j-sainsbury.co.uk/investor-centre/reports/2011/annual-report-and-financial-statements-2011/](http://www.j-sainsbury.co.uk/investor-centre/reports/2011/annual-report-and-financial-statements-2011/)  
[www.morrisons.co.uk/Documents/Morrisons-Annual-Report-2011.pdf](http://www.morrisons.co.uk/Documents/Morrisons-Annual-Report-2011.pdf)  
[www.kamcity.com/namnews/mktshare/2011/kantar-march11.htm](http://www.kamcity.com/namnews/mktshare/2011/kantar-march11.htm)

**vitalsigns**

**Description** Measurements of systolic blood pressure, diastolic blood pressure and pulse rate for four patients. This is fictional data.

**Variables**

subject	Patient identifier
test	Name of parameter (SysBP, DiaBP, Pulse)
result	Systolic blood pressure (mmHg), diastolic blood pressure (mmHg) or pulse (beats per minute)

**WHOdata**

**Description** Data on alcohol consumption and mortality rate for five countries, collected from the WHO website.

**Variables**

alcohol	Alcohol consumption per adult over 15 years (litres of pure alcohol per person per year)
mortality	Adult mortality rate (probability of dying between 15 and 60 years, per 1000 of population)

**Source** [apps.who.int/ghodata](https://apps.who.int/ghodata)



# Bibliography

---

- [1] Atkinson, A.C., 1985. *Plots, Transformations and Regression*. New York: Oxford University Press. p3.
- [2] Devore, J. and Peck, R., 2001. *Statistics: The Exploration and Analysis of Data*. Pacific Grove, CA: Brooks/Cole. p266.
- [3] Everitt, B.S., 1992. *The Analysis of Contingency Tables*. London: Chapman & Hall. p39.
- [4] Everitt, p285.
- [5] Fisher, R. A., 1922. *On the interpretation of  $\chi^2$  from contingency tables, and the calculation of P*, Journal of the Royal Statistical Society [e-journal] **85** (1) pp.87-94, Available through: JSTOR database [Accessed 18 August 2012].
- [6] Gibilisco, S., 2004. *Statistics Demystified*. McGraw-Hill. p202-6.
- [7] Hinton, P.R., 2004. *Statistics Explained*. 2nd ed. Routledge. p248-51.
- [8] Keith, T.Z., 2006. *Multiple Regression And Beyond*. Pearson. p197.
- [9] Kruskal, W. H. and Wallis, W. A., 1952. *Use of Ranks in One-Criterion Variance Analysis*, Journal of the American Statistical Association [e-journal] **47** (260) pp.583-621, Available through: JSTOR database [Accessed 18 August 2012].
- [10] Lewis, J.P. and Traill, A., 1998. *Statistics Explained*. Addison-Wesley. p419.
- [11] Lim, K., 2006. *Lecture 2: t-tests and the Central Limit Theorem*, [online] available at: <[http://www.stats.ox.ac.uk/~lim/Teaching\\_files/Lecture2/Lecture2\(4up\).pdf](http://www.stats.ox.ac.uk/~lim/Teaching_files/Lecture2/Lecture2(4up).pdf)> [Accessed 1 November 2012].
- [12] McClave, J.T. and Sincich, T., 2002. *Statistics*. 9th ed. Prentice Hall. p463-4.
- [13] McClave, p583.
- [14] McClave, p403.
- [15] McClave, p305.
- [16] Mendenhall, W., Wackerly, D.D. and Scheaffer, R.L., 1990. *Mathematical Statistics with Applications*. Belmont, CA: Duxbury Press. p534-8.
- [17] Mendenhall, p519-21.
- [18] Mendenhall, p237.

- [19] Mendenhall, p495-7.
- [20] Montgomery, D.C. and Runger, G.C., 2007. *Applied Statistics and Probability for Engineers* 4th ed. John Wiley & Sons.
- [21] NHS Choices, 2010. *How can I work out my body mass index (BMI)?* [online] Available at: <<http://www.nhs.uk/chq/Pages/how-can-i-work-out-my-bmi.aspx?CategoryID=51&SubCategoryID=165>> [Accessed 30 August 2012].
- [22] Ott, R.L. and Longnecker, M.T., 2000. *An Introduction to Statistical Methods and Data Analysis*. 5th ed. Florence, KY: Duxbury Press. p504.
- [23] Ott, p289.
- [24] Rawlings, J.O. and Pantula, S.G., 1998. *Applied Regression Analysis: A Research Tool*. Springer. p362.
- [25] Shapiro, S. S. and Wilks, M. B., 1965. *An analysis of variance test for normality (complete samples)*, *Biometrika* [e-journal] **52** (3-4) pp.591-611, Available through: JSTOR database [Accessed 18 August 2012].
- [26] Siegel, S. and Castellan, N.J. Jr., 1988. *Nonparametric Statistics for the Behavioural Sciences*, Singapore: McGraw-Hill. p134.
- [27] Spearman, C., 1904. *The proof and measurement of association between two things*, *American Journal of Psychology* [e-journal] **15** pp.72-101, Available through: JSTOR database [Accessed 18 August 2012].
- [28] The R Core Team, 2012. *Package 'foreign'* [online] Available at: <<http://cran.r-project.org/web/packages/foreign/foreign.pdf>> [Accessed 21 August 2012].
- [29] The R Core Team, 2012. *R Documentation: Add Points to a Plot* [online] Available at: <<http://127.0.0.1:29279/library/graphics/html/points.html>> [Accessed 23 August 2012].
- [30] The R Core Team, 2012. *R Documentation: Adjust P-values for Multiple Comparisons* [online] Available at: <<http://127.0.0.1:24075/library/stats/html/p.adjust.html>> [Accessed 12 September 2012].
- [31] The R Core Team, 2012. *R Documentation: Correlation, Variance and Covariance (Matrices)* [online] Available at: <<http://127.0.0.1:29279/library/stats/html/cor.html>> [Accessed 22 August 2012].
- [32] The R Core Team, 2012. *R Documentation: Date-time Conversion Functions to and from Character* [online] Available at: <<http://127.0.0.1:29279/library/base/html/strptime.html>> [Accessed 21 August 2012].
- [33] The R Core Team, 2012. *R Documentation: Distributions in the stats package* [online] Available at: <<http://127.0.0.1:17047/library/stats/html/Distributions.html>> [Accessed 21 August 2012].



- 
- [34] The R Core Team, 2012. *R Documentation: Generic X-Y Plotting* [online] Available at: <<http://127.0.0.1:29279/library/graphics/html/plot.html>> [Accessed 23 August 2012].
  - [35] The R Core Team, 2012. *R Documentation: Get and Set Contrast Matrices* [online] Available at: <<http://127.0.0.1:29279/library/stats/html/contrast.html>> [Accessed 28 August 2012].
  - [36] The R Core Team, 2012. *R Documentation: Remove Objects from a Specified Environment* [online] Available at: <<http://127.0.0.1:10620/library/base/html/rm.html>> [Accessed 01 September 2012].
  - [37] The R Core Team, 2012. *R Documentation: Select One of a List of Alternatives* [online] Available at: <http://127.0.0.1:13657/library/base/html/switch.html> [Accessed 28 September 2012].
  - [38] The R Core Team, 2012. *R Documentation: Set or Query Graphical Parameters* [online] Available at: <<http://127.0.0.1:29279/library/graphics/html/par.html>> [Accessed 21 August 2012].
  - [39] Upton, G. and Cook, I., 2008. *Oxford Dictionary of Statistics*. 2nd ed. Oxford: OUP. p79.
  - [40] Upton, p84.
  - [41] Upton, p308.
  - [42] Upton, p355.
  - [43] Upton, p204.
  - [44] Upton, p376.
  - [45] Upton, p387.
  - [46] Upton, p181-2.
  - [47] Upton, p180.
  - [48] Upton, p310.
  - [49] Venables, W. N., Smith, D. M. and the R Core Team, 2012. *An Introduction to R* [online] Available at: <<http://cran.r-project.org/doc/manuals/R-intro.pdf>> [Accessed 21 August 2012]. p40.
  - [50] Weiss, N.A., 2004. *Introductory Statistics*. 7th ed. Addison-Wesley. p662.
  - [51] Weiss, p776-8.
  - [52] Weiss, p829.
  - [53] Weisstein, E.W., *Spearman Rank Correlation Coefficient*. From MathWorld—A Wolfram Web Resource. [online] Available at: <<http://mathworld.wolfram.com/SpearmanRankCorrelationCoefficient.html>> [Accessed 4 October 2012].

- [54] Weisstein, E.W., *Distribution Function*. From MathWorld—A Wolfram Web Resource. [online] Available at: <<http://mathworld.wolfram.com/DistributionFunction.html>> [Accessed 4 October 2012].
- [55] Wilcoxon, F., 1945. *Individual comparisons by ranking methods*, Biometrics Bulletin [e-journal] **1** (6) pp.80-83, Available through: JSTOR database [Accessed 18 August 2012].

# Index

---

- abline function, 101, 116, 158
- addmargins function, 76
- aggregate function, 62
- analysis of variance, 136
- AND operator, 46, 172
- anova function, 136, 156
- aov function, 136
- appending, 49
- apply function, 35
- arrows function, 119
- as.character function, 35
- as.data.frame function, 76
- as.Date function, 41
- as.factor function, 34, 37
- as.numeric function, 34
- assignment operator, 6
- association, 63
- axis labels, 109
- axis limits, 111
  
- bar charts, 98
- barplot function, 98
- bartlett.test function, 143
- bitmap image, 106
- Bonferroni adjustment, 140
- boxplot function, 104, 111
- bracket notation, 10, 43, 46
  
- c function, 8, 19
- cbind function, 50
- character variables, 38
- chisq.test function, 77
- class function, 33
- coef function, 136, 158, 159
- coefficient estimates, 158
- coefficient of determination, 156
- coefficients function, 158
- colours, 112
- colours function, 112
- command prompt, 3
  
- comments, 16
- comparison operators, 172
- concatenating, 49
- conditional statements, 171
- conditions, 172
- confidence intervals, 70
- confint function, 136, 158
- contingency table, 74
- contrasts function, 152
- cooks.distance function, 164
- copying variables, 35
- cor function, 64, 65
- cor.test function, 66
- correlation, 63, 64
- cov function, 64
- covariance, 63, 64
- csv files, 21, 28
- cumulative distribution function, 84, 86
- curve function, 95, 105, 117, 159
- cut function, 36
  
- data editor, 13, 43
- data frames, 9
- data.frame function, 20, 166
- date function, 5
- dbinom function, 85
- DIF files, 21, 23
- difftime function, 42
- dnorm function, 86
- dpois function, 85
- duplicated function, 44
  
- error messages, 15
- Excel files, 24
- exp function, 8
- exporting datasets, 28
  
- F-test for equality of variance, 142
- F-test for model fit, 156
- factor variables, 37

- `fisher.test` function, 81
- `fitted` function, 160
- `fix` function, 13, 43
- for* loop, 178
- foreign package, 27
- `format` function, 39, 43
- `formula` function, 154
- `ftable` function, 75
- `function` function, 169
- functions, 5
  
- `getwd` function, 28
- `glm` function, 147
- graphics device, 93
- `grep` function, 40
- `grid` function, 118
  
- `hatvalues` function, 163
- `help` function, 6
- `hist` function, 95, 161
- histograms, 95
  
- if* statement, 173
- if/else* statement, 175
- intercept, 148, 152
- inverse cumulative distribution function, 84
- inverse cumulative distribution function, 88
- IQR function, 60
  
- jpeg image, 106
  
- Kolmogorov-Smirnov test, 68
- Kruskal-Wallis test, 138
- `kruskal.test` function, 138
- `ks.test` function, 68
  
- `legend` function, 122
- `length` function, 8, 60
- `levels` function, 37, 152
- leverage, 163
- line of best fit, 101, 158
- line thickness, 115
- line type, 114
- `lines` function, 119
- `lm` function, 147, 148
- `log` function, 151
- loops, 178
  
- Mann-Whitney U test, 134
- `max` function, 60
- `mean` function, 60
- `median` function, 60
- `merge` function, 51
- `min` function, 60
- multiple linear regression, 149
  
- `names` function, 33
- normal probability plots, 96
- numeric variables, 35
  
- objects, 6
- `objects` function, 14
- OR operator, 46, 173
- `order` function, 47
  
- paired t-test, 132
- `pairs` function, 102
- `pairwise.t.test` function, 140
- `pairwise.wilcox.test` function, 141
- `par` function, 113, 124
- `paste` function, 38
- `pbinom` function, 87
- pdf file, 106
- `pexp` function, 87
- `pie` function, 100, 110
- pie charts, 100
- `plot` function, 93, 98, 101, 161
- png image, 106
- `pnorm` function, 86
- `points` function, 119
- polynomial regression, 151
- precedence, 4
- `index` function, 164
- `predict` function, 70
- prediction intervals, 70
- probability density function, 84
- probability mass function, 84
- `prop.table` function, 75
  
- `qexp` function, 89
- `qnorm` function, 88
- `qpois` function, 89
- `qqline` function, 96
- `qqnorm` function, 96

- quantile function, 60
- quartz device, 93
- rainbow function, 113
- random numbers, 84, 89
- random sample, 47
- range function, 60
- rbind function, 49
- rbinom function, 90
- read.csv function, 21
- read.csv2 function, 23
- read.delim function, 21
- read.delim2 function, 23
- read.DIF function, 23
- read.table function, 24
- rearranging variables, 32
- relational operators, 45, 172
- relative file paths, 28
- relevel function, 38, 152
- removing variables, 32
- renaming variables, 33
- reshape function, 56
- resid function, 160
- residuals function, 160
- rexp function, 91
- rm function, 14, 20
- rnorm function, 90
- rotating, 56
- round function, 5, 43, 75
- rpois, 91
- rstandard function, 160
- rstudent function, 160
- sample function, 47, 90
- sapply function, 34, 60
- savePlot function, 106
- scatter plots, 101
- scatterplot matrices, 102
- script file, 15
- sd function, 60
- segments function, 116
- setwd function, 28
- shading density, 115
- shapiro.test function, 67
- simple linear regression, 148
- sorting, 47
- Spearman's correlation, 63, 65
- SPSS files, 26
- stack function, 54
- Stata files, 27
- stem function, 97
- stem-and-leaf plots, 97
- step function, 155
- stepwise model selection, 155
- strptime function, 42
- subset function, 32, 44
- substring function, 39
- subtitle, 109
- summary function, 59, 79, 156
- switch function, 176
- symbols, 114
- Sys.Date function, 43
- t.test function, 70, 129
- tab-delimited files, 21, 29
- table function, 73
- table object, 73
- tapply function, 61
- text function, 117
- title, 109
- transformations, 151
- TukeyHSD function, 139
- unique function, 44
- unstack function, 55
- update function, 153
- var function, 60
- var.test function, 142
- variable classes, 33
- vectors, 8
- warning messages, 15
- weekdays function, 43
- while loop, 179
- wilcox.test function, 134
- Wilcoxon rank-sum test, 134, 141
- workspace, 14
- write.csv function, 28
- write.table function, 29
- xtabs function, 76